Host Efficient Networking Stack
Utilizing NIC DRAM

APNet ’23
2023.06.29 (Thu) 10:15
HKUST

Byeongkeon Lee, Donghyeon Lee, Jisu Ok, Wonsup Yoon, and Sue Moon
KAIST
CPU Speed Has Stalled Since 2000s

- End of Dennard scaling and Moore’s laws: CPU speed has not improved.
- Performance improvement depends on multicore scalability or alternative hardware.
Yet Networking Speed Still Growing

- Nowadays, enterprise and data center networks are using 10 GbE ~ 100 GbE by default.
- Ethernet Alliance forecasts the Ethernet speed will be 800 GbE in 10 years.

https://ethernetalliance.org/technology/ethernet-roadmap/
Precious Host Resources

• For 100 Gbps data transfer, host CPU burden is heavy.
  - 1,500-byte-long MTU translates to 8.33 Mpps.
  - Host CPU has to copy 12.2 GB/s from application buffer to socket buffer.

• Host memory access for copy is also not negligible.
  - 12.2 GB/s for read from app buffer and 12.2 GB/s to write to intermediate buffer.

• End-host resources become scarce.
  - We have to reduce the CPU usage and memory access caused by networking.
Existing Efforts to Save Host Resources

1. Eliminate payload copy by avoiding intermediate buffer
   - Memory pool-based I/O: Netmap, DPDK, MAIO
   - Application buffer direct I/O
     • Copy-on-write: ZCopy, zIO
     • Completion notification: MSG_ZEROCOPY, io_uring

2. Exploit special-purpose hardware for copy
   - Memory copy offloading to I/OAT engine

3. Offload network protocol to NIC
   - Common operation offloading to commodity NIC: checksum, TSO, header-split, etc.
   - TCP protocol offloading to programmable NIC: Verilog-Ethernet, Limago, Tonic, AccelTCP, etc.
Pros and Cons of #1 Eliminate payload copy

+ Payload memory copy that takes up significant CPU cycles is eliminated.

- Memory pool-based I/O
  - Memory pool-based zero-copy should use provided API causing application modification.

- Application buffer direct I/O – Copy-on-write
  - Copy-on-write protects the buffer in the unit of pages and has to handle protection faults.

- Application buffer direct I/O – Completion notification
  - Completion notification approach requires application modification and has a long buffer holding time.
Pros and Cons of #1 Eliminate payload copy

+ Payload memory copy that takes up significant CPU cycles is eliminated.

• Memory pool-based I/O
  - Memory pool-based zero-copy should use provided API causing application modification.

• Application buffer direct I/O – Copy-on-write
  - Copy-on-write protects the buffer in the unit of pages and has to handle protection faults.

• Application buffer direct I/O – Completion notification
  - Completion notification approach requires application modification and has a long buffer holding time.
Pros and Cons of #1 Eliminate payload copy

- Payload memory copy that takes up significant CPU cycles is eliminated.

- **Memory pool-based I/O**
  - Memory pool-based zero-copy should use provided API causing application modification.

- **Application buffer direct I/O – Copy-on-write**
  - Copy-on-write protects the buffer in the unit of pages and has to handle protection faults.

- **Application buffer direct I/O – Completion notification**
  - Completion notification approach requires **application modification** and has a **long buffer holding time**.
MSGZEROCOPY Mechanism

Sender Application

"Hello world"

TCP Stack

Receiver Application

TCP Stack

ACK
Application expects receiver to get “Hello world”, but receiver gets “Goodbye world”

Data corrupt!
MSG_ZEROCOPY Mechanism – Completion Notification

1. Application has to handle completion notification to safely modify/free the buffer.
2. Buffer holding time is at least 1 RTT.
Pros and Cons of #2 Exploit special-purpose H/W

+ Payload copy from application buffer to socket buffer does not involve host CPU.

− It still wastes **significant memory bandwidth**.
Pros and Cons of #3 Offload network protocol to NIC

+ Many complex TCP operations are done without involving host CPU.

− Protocols in the NIC cannot be modified easily: poor flexibility.
Observations

• Buffer sharing between application and networking stack
  → Application modification and long buffer holding time.

• The intermediate buffer located on the host
  → Significant memory access and bandwidth usage

• Protocol processing in NIC
  → Inflexible networking stack.

Recent NICs have evolved to have powerful core and plentiful DRAM.
Our Solution

- Buffer sharing between application and networking stack
  → Application modification and long buffer holding time.

- The intermediate buffer located on the host
  → Significant memory access and bandwidth usage

So why not put the payload buffer in NIC DRAM via NIC DMA Engine?

- Protocol processing in NIC
  → Inflexible networking stack

and leave protocol processing on the host?
Design Proposal and Limitations

- Our design is a user-space TCP library.

- Socket APIs for active open (socket, connect, write, close) are implemented as proof of concept

- Other APIs (bind, listen, accept, read) and part of TCP (window control) are for future work.
Implementation

- **Alveo U200 FPGA Module**
  
  Reference: Xilinx University Program Opensource
  
  Design Flow: Vitis HLS with v++ compiler 2021.2
  
  Block Implementation: Vitis HLS (High Level Synthesis)

- **TCP Stack (user space)**
  
  Reference: About 2,500 C++ LoC from scratch
  
  TCP Stack: Half TCP Stack (TCP Active Open)
  
  Library: XRT (Xilinx Runtime Library) 2.12.427
Evaluation Setup

Machine Configuration

CPU : Intel Xeon Gold 6226R CPU@2.90 GHz 16 Cores
   hyperthreading disabled
Memory : 192 GiB Samsung DDR4 2933 MHz
NUMA : Single
PCle : Gen 3.0 x 16 lanes
Sender : Xilinx Alveo U200 Data Center Acceleration Card including 4 x 16GB DDR4 2400 MT/s + Our stack
Receiver : Intel E810-CQDA2 100 Gbps + DPDK-based simple TCP receiver stack
Evaluation Benchmark

We compare four I/O schemes on the same hardware, Alveo U200 FPGA NIC:

(1) Baseline

(2) I/OAT

(3) Zero-copy

(4) Our work
Evaluation - I/O Schemes

1. Baseline
   - **CPU** copies application buffer into host memory accessible by FPGA.

2. I/OAT
   - **I/OAT engine** copies to host memory accessible by FPGA after the buffer pinning.
   - We modified SPDK I/OAT source code to support general application environment.

3. Zero-copy
   - Application buffers are pinned as **MSG_ZEROCOPY** does.
   - After the transmission, TCP stack notifies application via completion message.

• Above three I/O schemes concatenate header and payload from host (No FPGA DRAM).
Evaluation Main Points

We compare four I/O schemes on the same hardware, Alveo U200 FPGA NIC:

(1) Baseline, (2) I/OAT, (3) Zero-copy, (4) Our work

Does our work show:

1. Socket API intact?
2. Less CPU usage than the baseline?
3. Comparable throughput with other I/O schemes?
4. Less memory access and bandwidth usage than the baseline and the I/OAT?
5. Shorter buffer holding time than the zero-copy?
Socket API Intact

```c
int main(int argc, char const* argv[]){
    // Application logic...
    char tmp;
    if(strcmp(argv[2]) == 0){
        printf("payload 0. Terminating program\n");
        return 0;
    }
    sleep(0);
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(12345); // Example port number
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    socklen_t addrlen = sizeof(serv_addr);
    connect(fd, (const struct sockaddr*)&serv_addr, addrlen);
    int length = strlen(argv[2]);
    char hello = payload[0];
    printf("Hello = payload[0]\n");
    write(fd, hello, length);
    printf("hello = payload[0]\n");
    write(fd, hello, length);
    close(fd);
    sleep(0);
    return 0;
}
```

- Our approach is compatible with socket APIs.
- We **eliminate** completion notification handling in application logic!
- Application can use our TCP stack without modification by: `LD_PRELOAD=./libtcp.so`
- Our work shows 38.6% fewer CPU cycles than baseline.

- Our work does not involve host CPU for data transfer.

- 3.6 MB buffers * 1,125 times = about 4 GB

Lower is better
Throughput

- Baseline and our work show similar throughput.
- Handling completion notification lowers throughput.

Higher is better
- Tool: Intel PCM (Performance Counter Monitor)
- Both one read and one write are removed.
- Our work accesses memory 61.7% lower than I/OAT.
- Our work saves mem b/w by 17.3 GB/s than I/OAT.
- It is equal to 12.3% system-wide mem b/w save.

(system-wide theoretical max b/w: 140.8 GB/s)

Lower is better
Buffer Holding Time

Our work does not require 1 RTT buffer holding time.

Note that two servers are directly connected.
Future Works

• Retransmission of the data that has been transferred into the NIC DRAM

• Receiver-side CPU-based copy elimination

• Various protocol support
Conclusion

Existing solutions to save host CPU have limitations:
- application modification, long buffer holding time, memory bandwidth, hardware flexibility, etc.

We have proposed new I/O architecture utilizing NIC DRAM that
- Leaves the protocol processing to the host to **preserve flexibility**.
- **Reduces CPU usage** than baseline and **host memory access/bandwidth usage** than I/OAT.
- Has **shorter buffer holding time** than zero-copy.
- Does **not require application modification**.
Thank you