NetTLP: A Development Platform for PCIe Devices in Software Interacting with Hardware

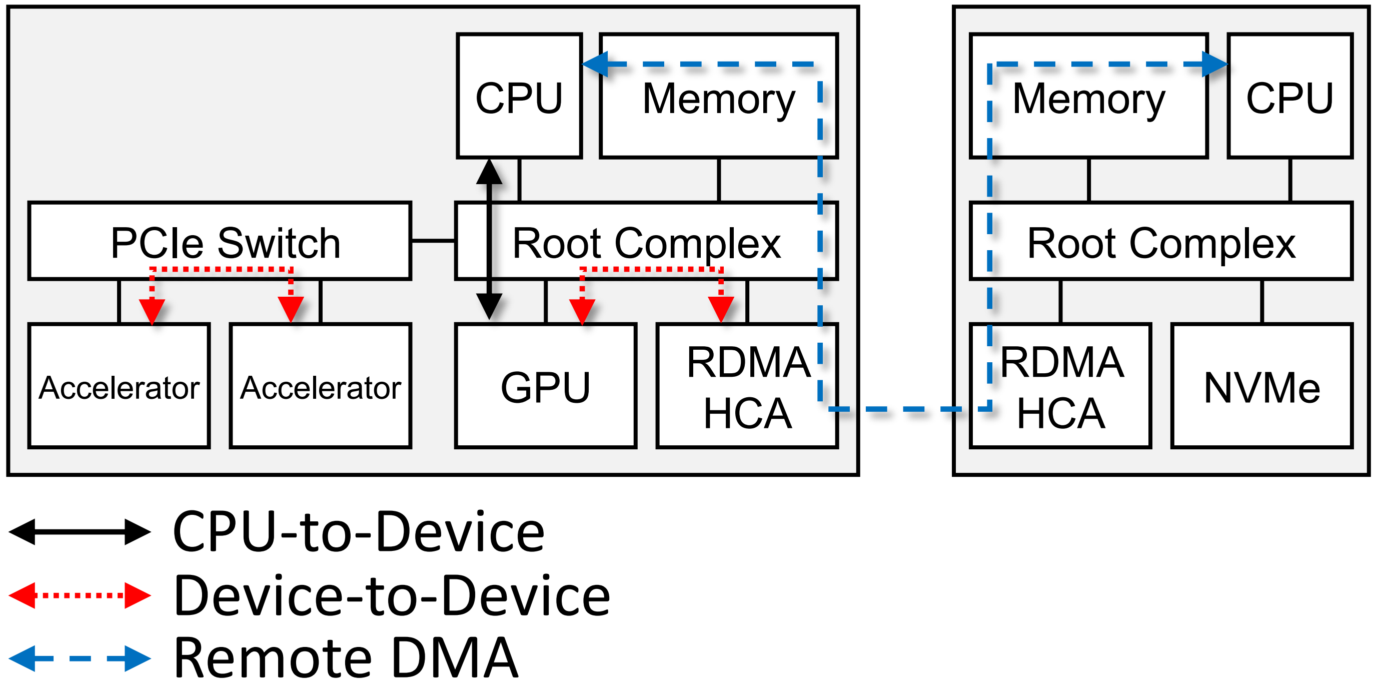
Yohei Kuga (*The University of Tokyo)*

Ryo Nakamura (*The University of Tokyo)*

Takeshi Matsuya (*Keio University)*

Yuji Sekiya (*The University of Tokyo)*

PCI Express-Based Heterogeneous Computing



- PCI Express (PCIe) is the most popular interconnect standard for communicating between accelerator, storage, and network devices
- PCIe is a packet-based protocol
  - PCIe topology is flexible
  - PCIe switch and root complex forward PCIe packets to other PCIe devices
  - PCIe devices can communicate directly by using the PCIe switch

Problem: Lack of Productivity and Observability on PCIe

- Why can't we develop PCIe the same way as IP networking
  - Although both PCIe and IP are packet-based data communication standards
- Prototyping a PCIe device by FPGA still requires significant effort
  - Such as in the NetFPGA project
- Observing PCIe transactions is also difficult
  - Because they are confined in hardware and require special analyzers

	IP networks	PCI Express
Type of data communication	Packet-based	Packet-based
Components	**Software** and hardware	Hardware
Analyzing by	**tcpdump, Wireshark, etc**	FPGA, special hardware

Gap between Software and Hardware

Goal

- Bridge the gap between hardware and software for PCIe
    - QEMU performs everything in **software** but without actual PCIe protocols
    - FPGA and ASIC handle actual PCIe transactions in **hardware**, but developing them is still hard compared with software-based platforms

NetTLP provides high productivity and observability for PCIe developments by connecting **software PCIe devices** to **hardware root complexes**

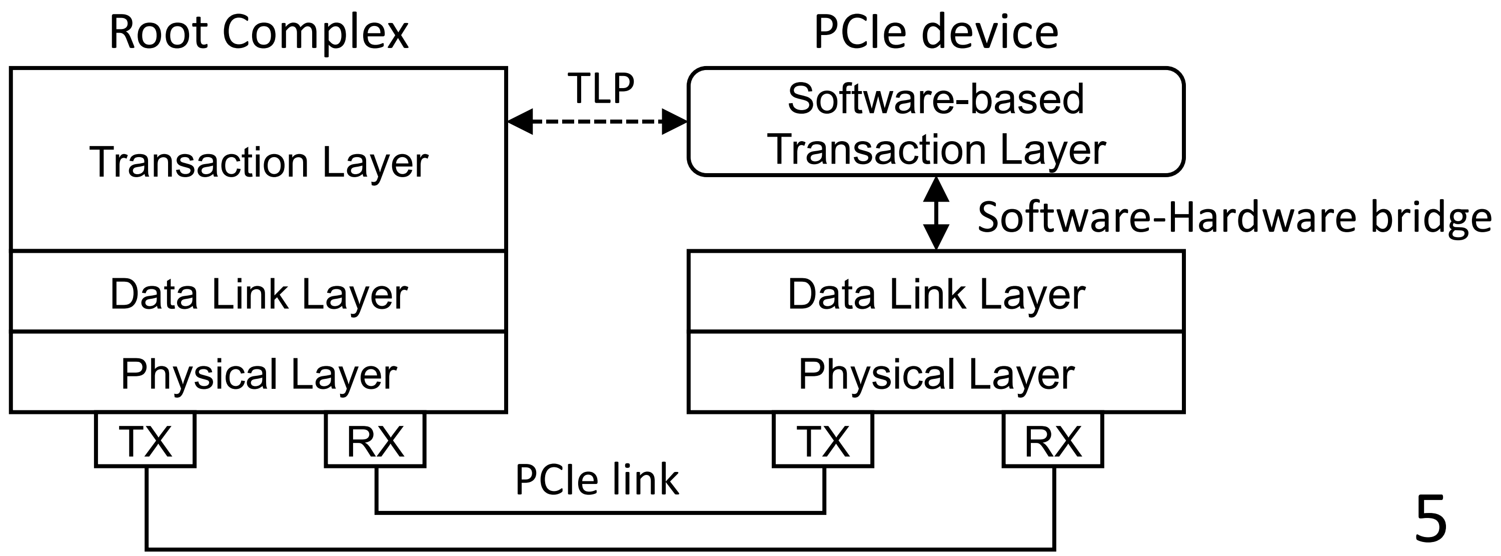
		PCIe device	
		Software	Hardware
Root complex	Software	QEMU	-
	Hardware	**NetTLP**	FPGA/ASIC

NetTLP approach

- Separating the PCIe transaction layer into software
  - Software PCIe devices communicate with hardware root complexes on the PCIe transaction layer
- Bridging the software transaction layer with hardware data link layer by delivering TLPs over Ethernet
  - It is possible because both use packet-based data communication

TLP manipulation platform
- [ExpEther HOTI'06]
- [Thunderclap NDSS'19]

NetTLP target
- Software PCIe device

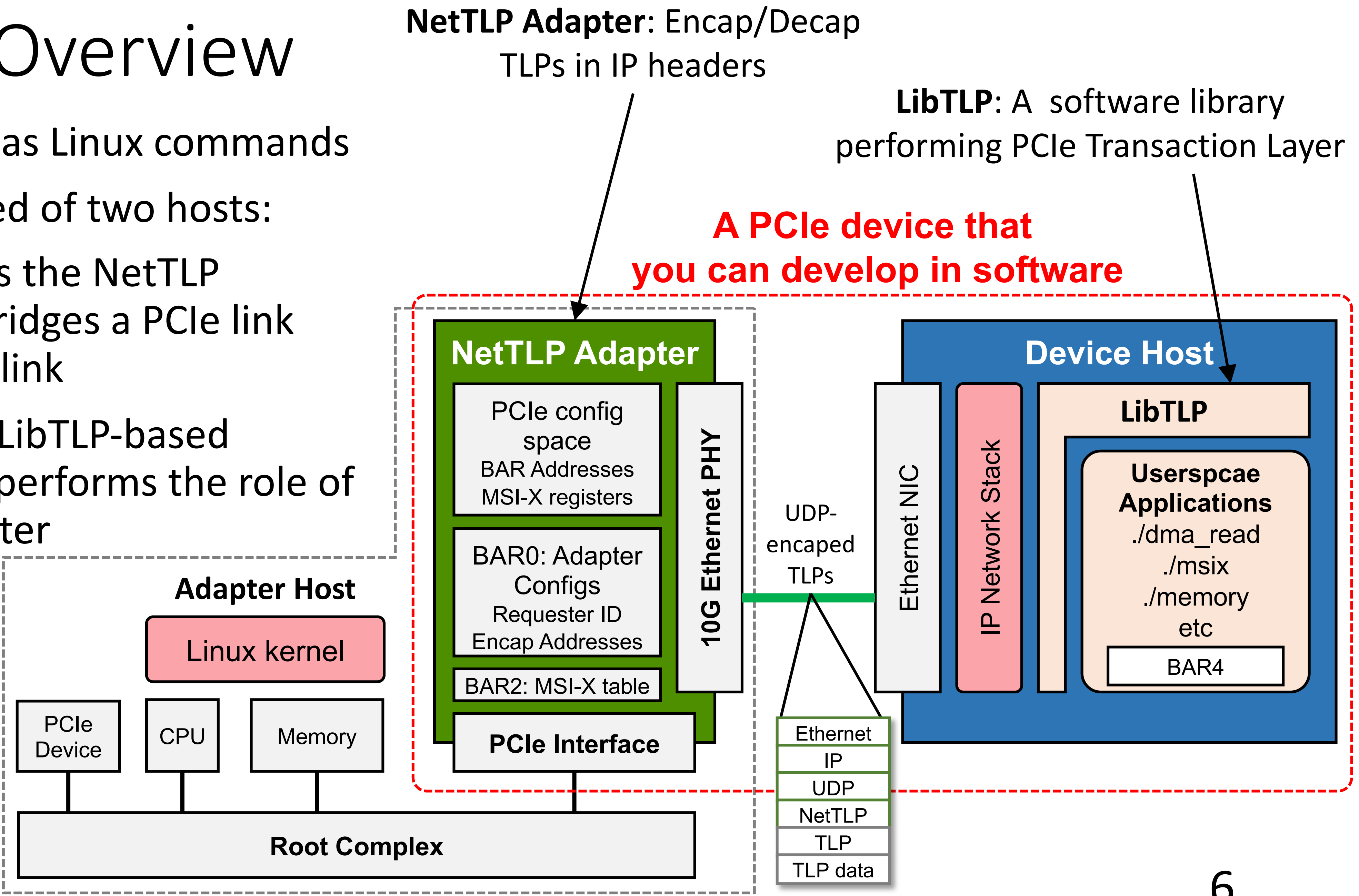


NetTLP Overview

PCIe devices work as Linux commands

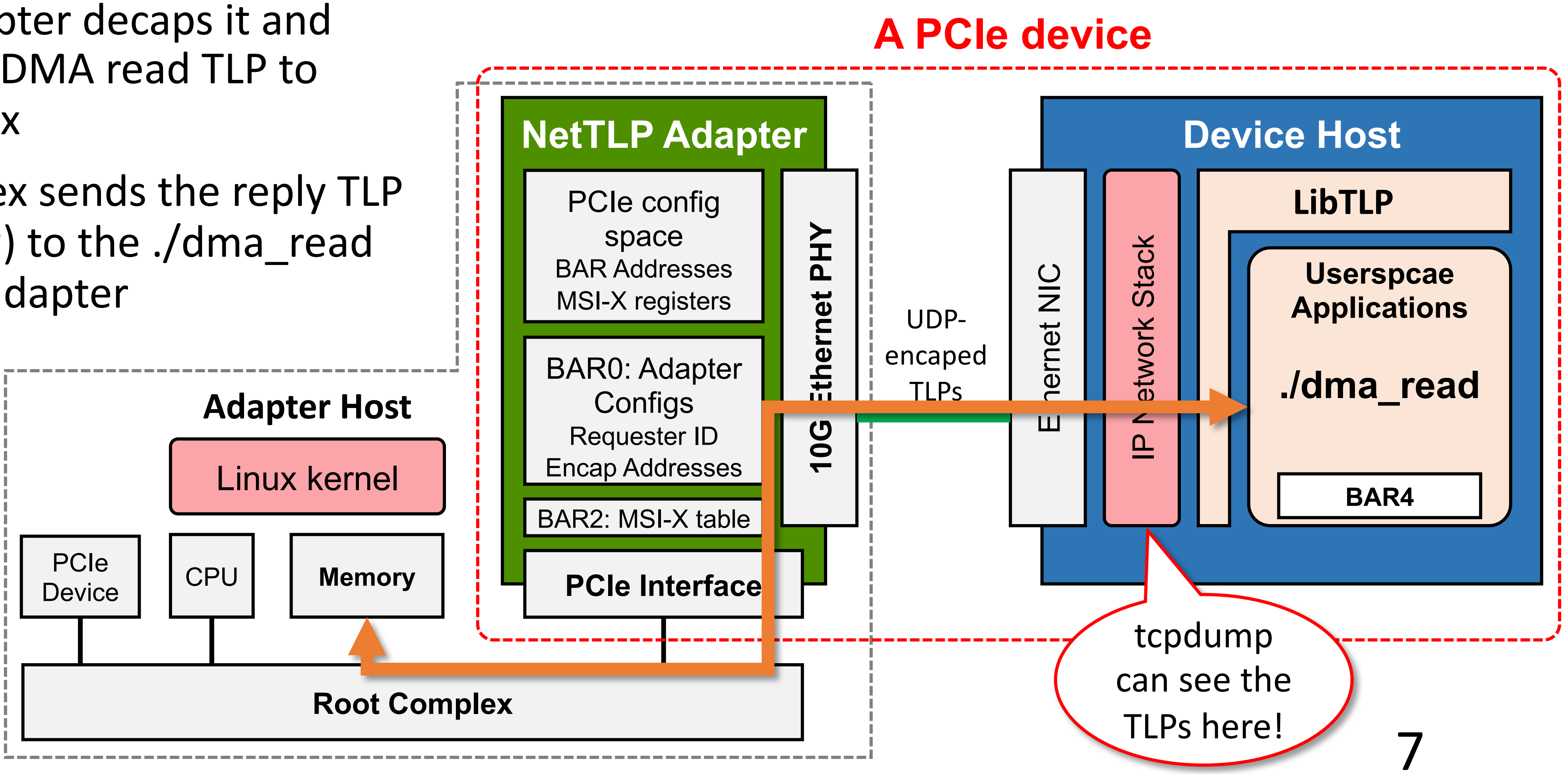
NetTLP is composed of two hosts:

- **Adapter host** has the NetTLP adapter which bridges a PCIe link and an Ethernet link
- **Device host** has LibTLP-based application that performs the role of the NetTLP adapter



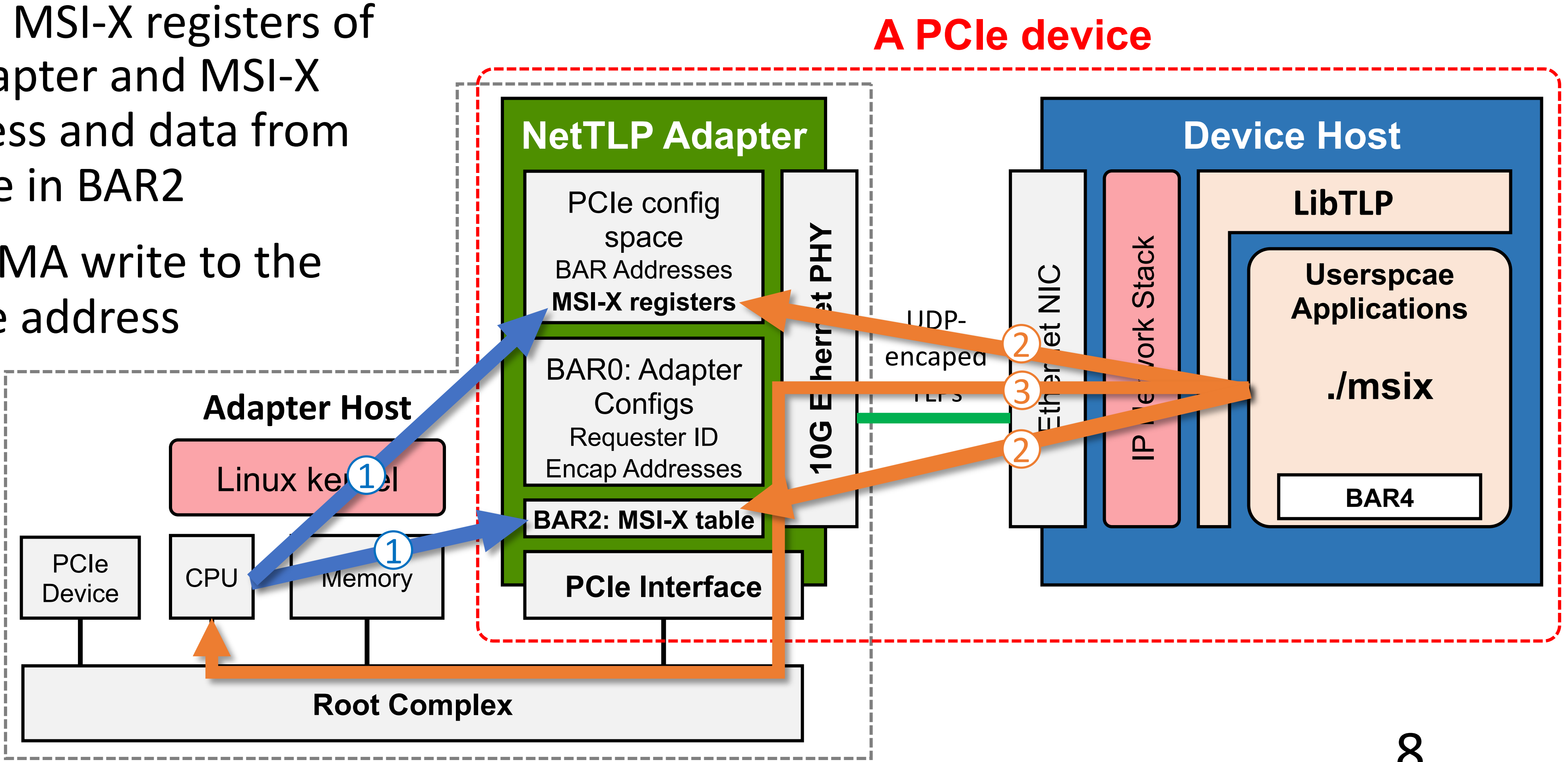
Example 1: DMA Read by Software from the Device Host

1. ./dma_read sends a DMA read TLP over UDP
2. The NetTLP adapter decaps it and sends the inner DMA read TLP to the root complex
3. The root complex sends the reply TLP (completion TLP) to the ./dma_read via the NetTLP adapter



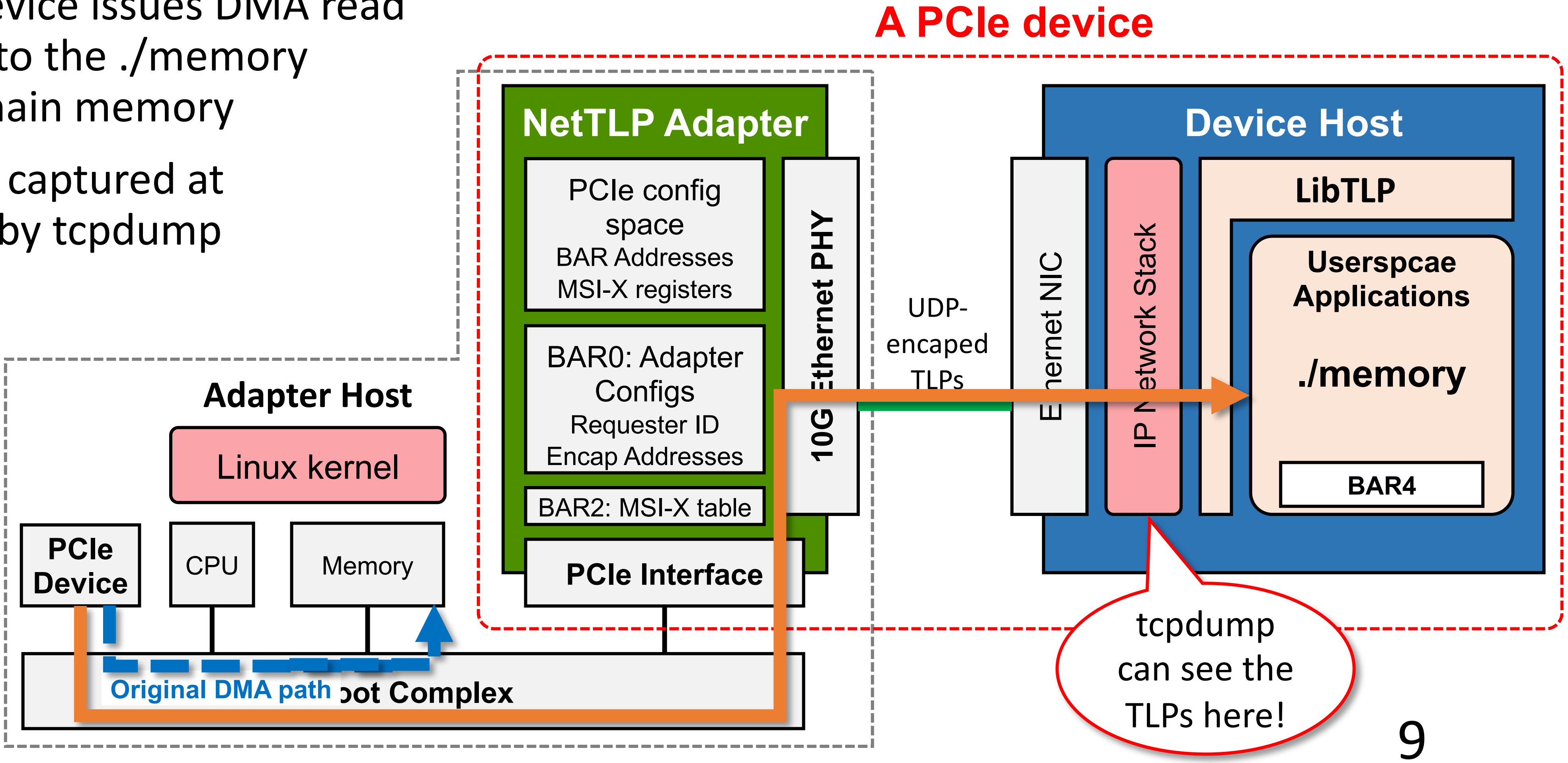
Example 2: Generating MSI-X Interrupts in NetTLP Platform

1. Interrupt controller sets MSI-X table data
2. ./msix gets the MSI-X registers of the NetTLP adapter and MSI-X message address and data from the MSI-X table in BAR2
3. ./msix sends DMA write to the MSI-X message address



Example 3: Capturing TLPs from Other PCIe Devices

1. ./memory performs a memory region associating with BAR4 of the NetTLP adapter
2. Another PCIe device issues DMA read and DMA write to the ./memory instead of the main memory
3. The TLPs can be captured at the device host by tcpdump



LibTLP Design: DMA APIs

```
ssize_t dma_read(struct nettlp *nt, uintptr_t addr, void *buf, size_t count);
ssize_t dma_write(struct nettlp *nt, uintptr_t addr, void *buf, size_t count);
```

- DMA APIs are inspired by read(2) and write(2) system calls
  - dma_read() attempts to read up to `` `**count**` `` bytes into `` `**buf**` ``
  - dma_write() writes up to `` `**count**` `` bytes from `` `**buf**` ``
  - `` `**addr**` `` indicates a target address of DMA transaction
  - The return values of the functions
    - Success: the number of bytes read or written
    - Error: returns -1 and sets errno

LibTLP Design: PIO APIs

```
struct nettlp_cb {
        int (*mrd)(struct nettlp *nt, struct tlp_mr_hdr *mh, …);
        int (*mwr)(struct nettlp *nt, struct tlp_mr_hdr *mh, …);
        int (*cpl)(struct nettlp *nt, struct tlp_cpl_hdr *ch, …);
        int (*cpld)(struct nettlp *nt, struct tlp_cpl_hdr *ch, …);
        int (*other)(struct nettlp *nt, struct tlp_hdr *tlp, …);
};
```

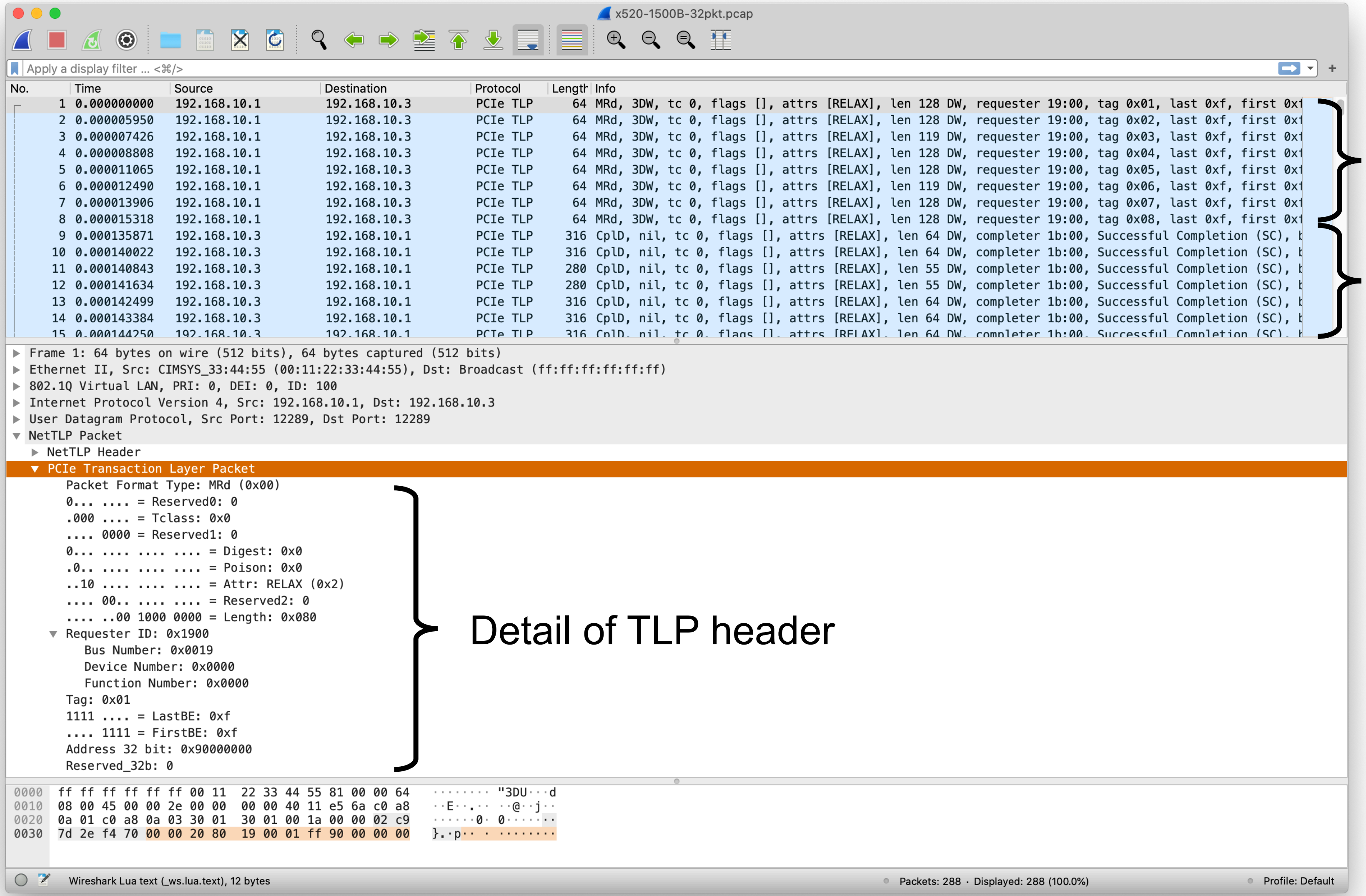
- Register the functions receiving the request TLPs using callback API
- Call nettlp_run_cb() / nettlp_stop_cb() to start/stop the software device

Example) dma_read.c

- Programing PCIe devices in the same manner as IP packet processing with Linux
  1. Set IP packet parameters
  2. Set TLP header parameters
  3. Call the DMA read API
  4. Output DMA read results

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <libtlp.h>

int main(int argc, char **argv) {
        uintptr_t addr = 0x0;
        struct nettlp nt;
        char buf[128];
        int ret;

        inet_pton(AF_INET, "192.168.10.1", &nt.remote_addr);
①       inet_pton(AF_INET, "192.168.10.3", &nt.local_addr);
        nt.requester = (0x1a << 8 | 0x00);
②       nt.tag = 0;

        nettlp_init(&nt);

        ret = dma_read(&nt, addr, buf, sizeof(buf));
        if (ret < 0) {
③               perror("dma_read");
                return ret;
        }

④       printf("DMA read: %d bytes from 0x%lx¥n", ret, addr);
        return 0;
}
```

Observing Actual TLPs with Tcpdump and Wireshark!



x520-1500B-32pkt.pcap

Apply a display filter ... <⌘/>

```
No.  Time         Source        Destination   Protocol  Length Info
  1 0.000000000  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 128 DW, requester 19:00, tag 0x01, last 0xf, first 0xf
  2 0.000005950  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 128 DW, requester 19:00, tag 0x02, last 0xf, first 0xf
  3 0.000007426  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 119 DW, requester 19:00, tag 0x03, last 0xf, first 0xf
  4 0.000008808  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 128 DW, requester 19:00, tag 0x04, last 0xf, first 0xf
  5 0.000011065  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 128 DW, requester 19:00, tag 0x05, last 0xf, first 0xf
  6 0.000012490  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 119 DW, requester 19:00, tag 0x06, last 0xf, first 0xf
  7 0.000013906  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 128 DW, requester 19:00, tag 0x07, last 0xf, first 0xf
  8 0.000015318  192.168.10.1  192.168.10.3  PCIe TLP    64 MRd, 3DW, tc 0, flags [], attrs [RELAX], len 128 DW, requester 19:00, tag 0x08, last 0xf, first 0xf
  9 0.000135871  192.168.10.3  192.168.10.1  PCIe TLP   316 CplD, nil, tc 0, flags [], attrs [RELAX], len 64 DW, completer 1b:00, Successful Completion (SC), b
 10 0.000140022  192.168.10.3  192.168.10.1  PCIe TLP   316 CplD, nil, tc 0, flags [], attrs [RELAX], len 64 DW, completer 1b:00, Successful Completion (SC), b
 11 0.000140843  192.168.10.3  192.168.10.1  PCIe TLP   280 CplD, nil, tc 0, flags [], attrs [RELAX], len 55 DW, completer 1b:00, Successful Completion (SC), b
 12 0.000141634  192.168.10.3  192.168.10.1  PCIe TLP   280 CplD, nil, tc 0, flags [], attrs [RELAX], len 55 DW, completer 1b:00, Successful Completion (SC), b
 13 0.000142499  192.168.10.3  192.168.10.1  PCIe TLP   316 CplD, nil, tc 0, flags [], attrs [RELAX], len 64 DW, completer 1b:00, Successful Completion (SC), b
 14 0.000143384  192.168.10.3  192.168.10.1  PCIe TLP   316 CplD, nil, tc 0, flags [], attrs [RELAX], len 64 DW, completer 1b:00, Successful Completion (SC), b
 15 0.000144250  192.168.10.3  192.168.10.1  PCIe TLP   316 CplD, nil, tc 0, flags [], attrs [RELAX], len 64 DW, completer 1b:00, Successful Completion (SC), b
```

```
▶ Frame 1: 64 bytes on wire (512 bits), 64 bytes captured (512 bits)
▶ Ethernet II, Src: CIMSYS_33:44:55 (00:11:22:33:44:55), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▶ 802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 100
▶ Internet Protocol Version 4, Src: 192.168.10.1, Dst: 192.168.10.3
▶ User Datagram Protocol, Src Port: 12289, Dst Port: 12289
▼ NetTLP Packet
  ▶ NetTLP Header
  ▼ PCIe Transaction Layer Packet
        Packet Format Type: MRd (0x00)
        0... .... = Reserved0: 0
        .000 .... = Tclass: 0x0
        .... 0000 = Reserved1: 0
        0... .... .... .... = Digest: 0x0
        .0.. .... .... .... = Poison: 0x0
        ..10 .... .... .... = Attr: RELAX (0x2)
        .... 00.. .... .... = Reserved2: 0
        .... ..00 1000 0000 = Length: 0x080
      ▼ Requester ID: 0x1900
           Bus Number: 0x0019
           Device Number: 0x0000
           Function Number: 0x0000
        Tag: 0x01
        1111 .... = LastBE: 0xf
        .... 1111 = FirstBE: 0xf
        Address 32 bit: 0x90000000
        Reserved_32b: 0
```

```
0000  ff ff ff ff ff ff 00 11  22 33 44 55 81 00 00 64   ········ "3DU··d
0010  08 00 45 00 00 2e 00 00  00 00 40 11 e5 6a c0 a8   ··E··.·· ··@··j··
0020  0a 01 c0 a8 0a 03 30 01  30 01 00 1a 00 00 02 c9   ······0· 0·······
0030  7d 2e f4 70 00 00 20 80  19 00 01 ff 90 00 00 00   }.·p·· · ········
```

Wireshark Lua text (_ws.lua.text), 12 bytes   Packets: 288 · Displayed: 288 (100.0%)   Profile: Default

Captured the DMA read TLPs from the physical NIC

./memory replied with the Completion TLPs to the NIC

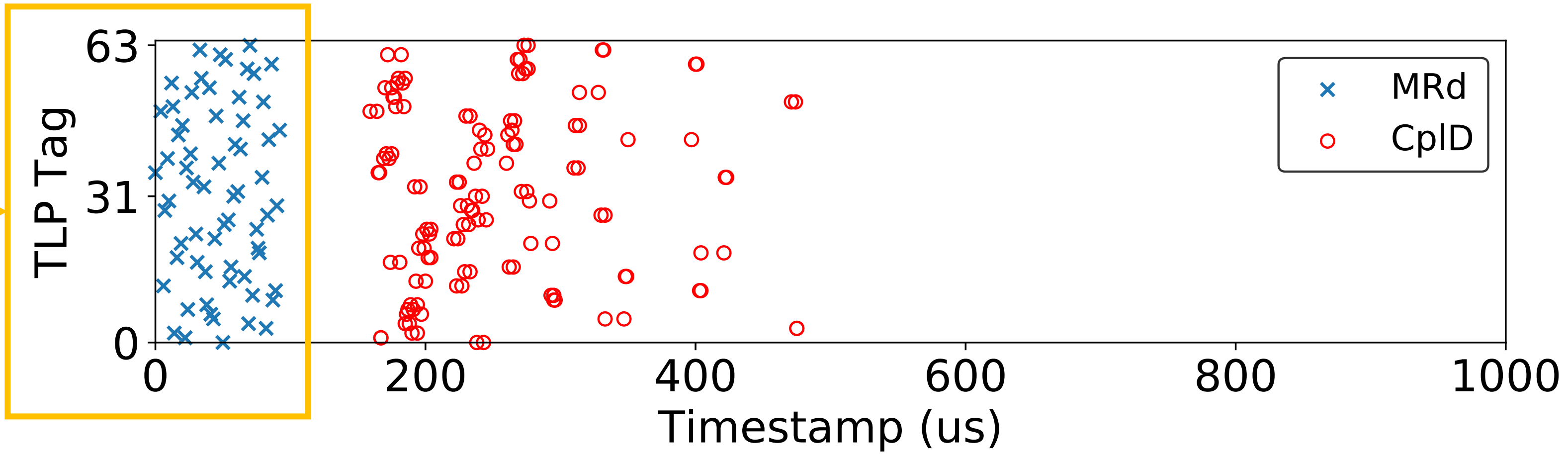
Detail of TLP header

We've implemented an FPGA-based NetTLP adapter with 10Gbps Ethernet and PCIe Gen2 interface

Challenge 1: Receiving Burst TLPs

- PCIe could momentarily send TLPs at Ethernet wire-speed
  - PCIe endpoints use different TLP tag values to send consecutive DMA read requests (split-transaction)
  - The encapsulated DMA read TLP is 64 bytes = Ethernet short packet size
- LibTLP needs to receive such burst TLPs

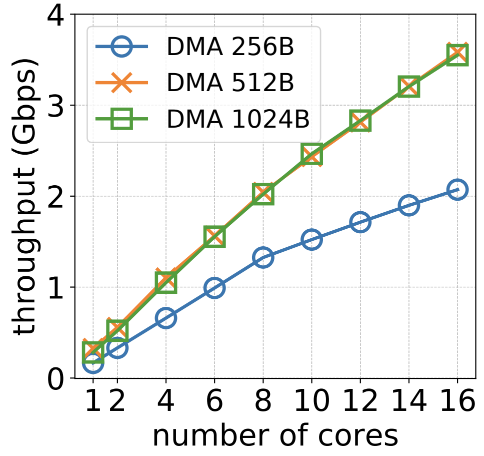
This NVMe sends 64 DMA read requests at a time in this experiment



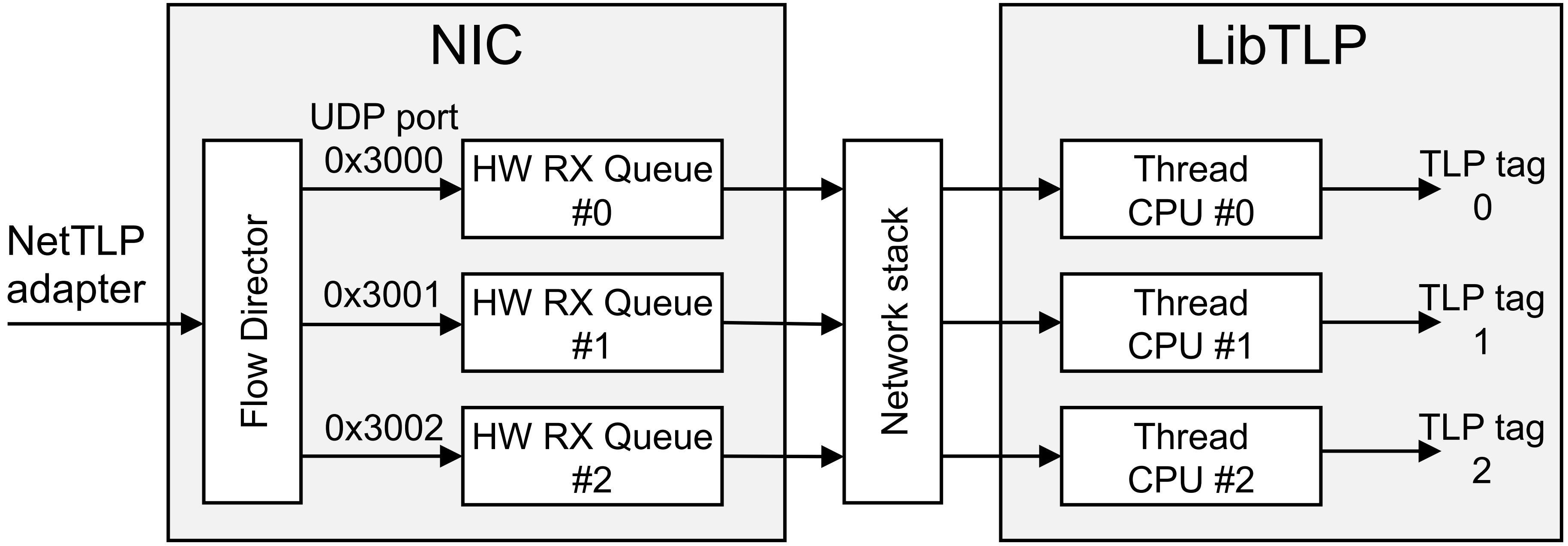
DMA Read Requests for writing 8 blocks issued from Samsung PM1725a NVMe (captured by NetTLP)

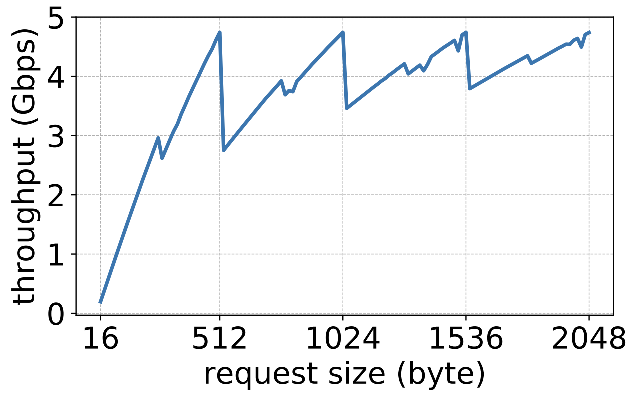
Challenge 1: Receiving Burst TLPs

- Exploiting multi-cores and multi-queues for PCIe transactions from software
- NetTLP adapter maps TLP tag values to UDP port numbers for encapsulation
    - TLPs are delivered through different UDP flows based on the tag field
    - LibTLP receives the flows by different NIC queues and CPU cores
- Our implementation with 16 core: DMA read 3.6 Gbps



DMA read throughput from LibTLP to the NetTLP adapter





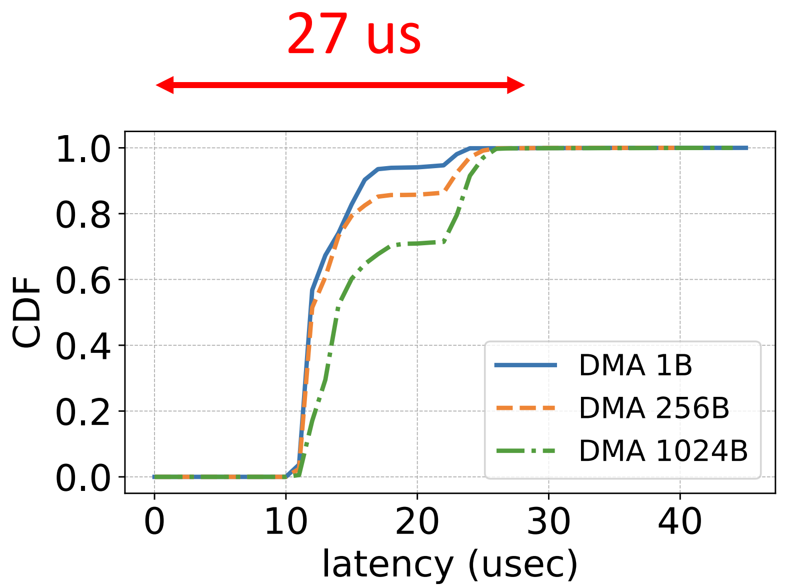
DMA read throughput from NetTLP adapter to LibTLP

Challenge 2: Completion Timeout

- PCIe specification defines the completion timeout
  - Minimal range is 50 us to 10 ms
  - PCIe specification recommends that PCIe devices do not expire in less than 10 ms
  - Intel X520 NIC sets the range from 50 us to 50 ms
- Our software implementation result:
  - 99% DMA read latency is less than 27 us

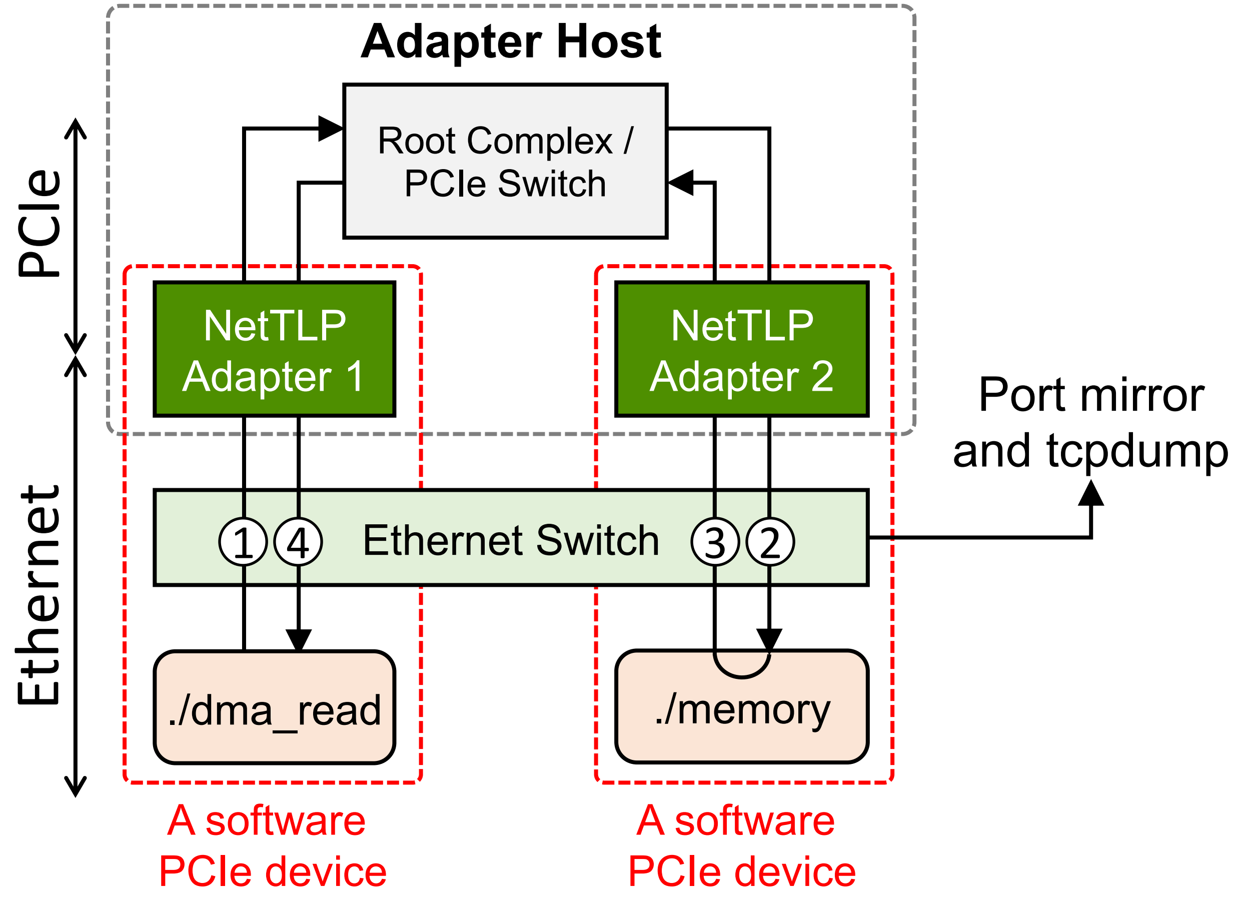
```
$ sudo lspci -vv
01:00.0 Ethernet controller: Intel Corporation 82599ES
DevCtl: MaxPayload 128 bytes, MaxReadReq 512 bytes
DevCtl2: Completion Timeout: 50us to 50ms,
```

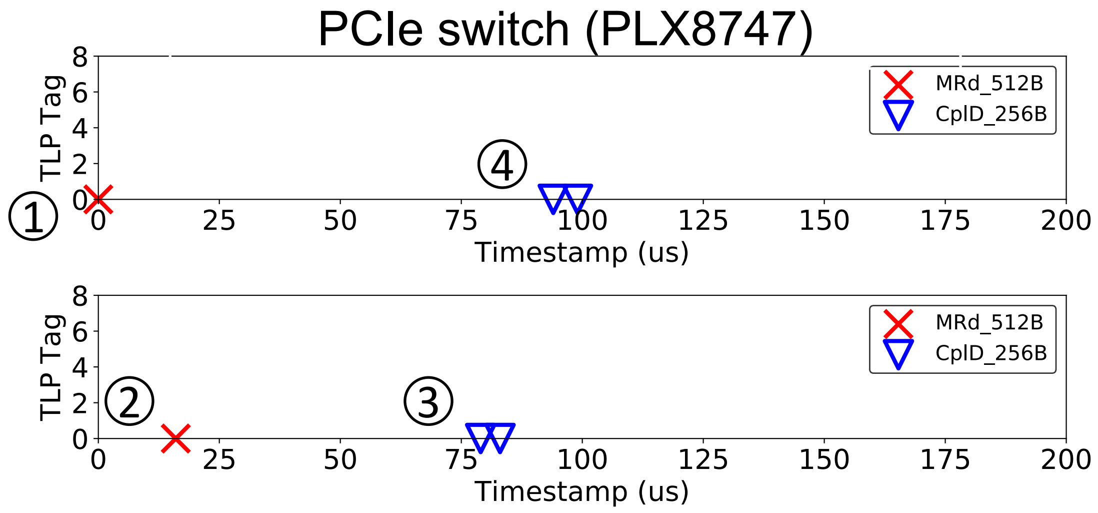
Completion timeout of Intel X520 NIC

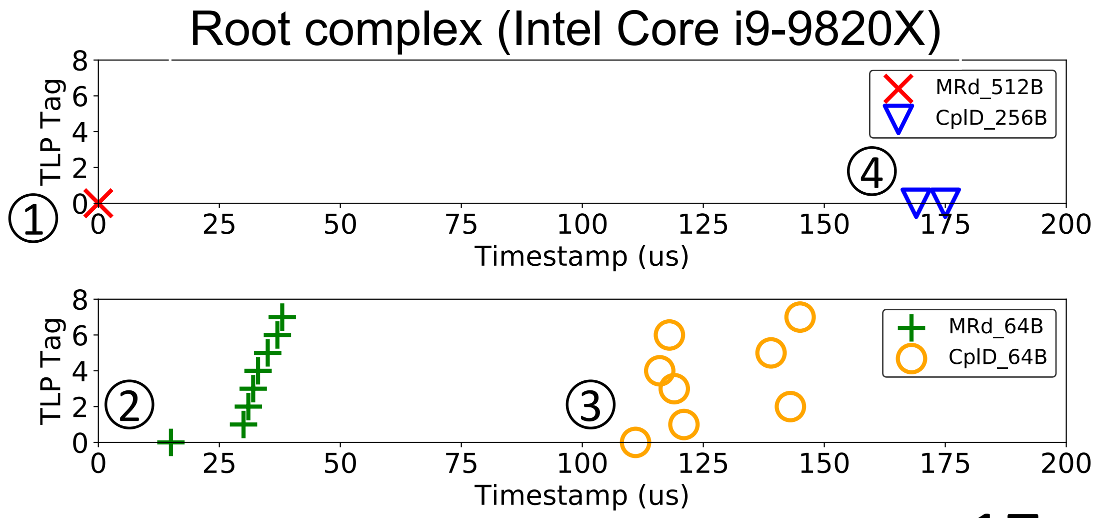


DMA read latency from LibTLP to NetTLP adapter

Use Case 1: Observing Root Complex and PCIe Switch Behavior



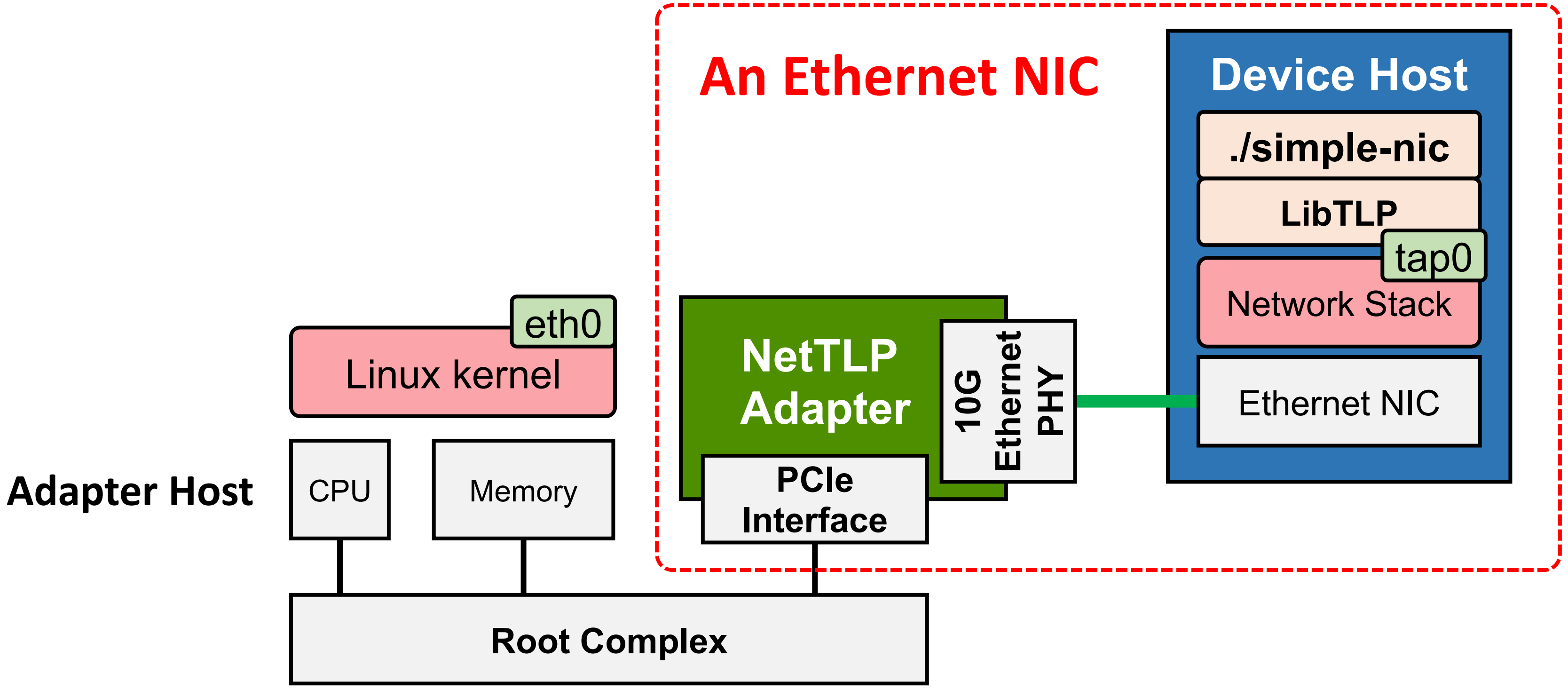




- ./dma_read sends a 512B DMA read request
- Root complex splits the 512B DMA read into eight 64B request TLPs and rebuilds two 256B completion TLPs (MaxPayloadSize = 256B)

Use Case 2: A Nonexistent NIC

- To confirm the productivity of NetTLP, we implemented an Ethernet NIC
  - Target NIC: **simple-nic** introduced by [pcie-bench SIGCOMM’18]
    - A theoretical model of a simple Ethernet NIC
  - ./simple-nic uses a tap interface as its Ethernet port



The simple-nic model certainly works with a root complex

- ./simple-nic on the NetTLP platform can TX/RX packets
- All the PCIe interactions with the root complex can be observed by tcpdump
- The device code is 400 LoC in C

tcpdump outputs (packet info only) for sending an ICMP echo packet from the host

1. NIC driver updates TX queue tail pointer	`MWr, 3DW, WD, tc 0, flags [none], attrs [none], len 1, requester 00:00, tag 0x01, last 0x0, first 0xf, Addr 0xb0000010`

2-3. NIC reads the TX queue descriptor from the main memory	`MRd, 3DW, tc 0, flags [none], attrs [none], len 4, requester 1b:00, tag 0x01, last 0xf, first 0xf, Addr 0x2f004000`
	`CplD, 3DW, WD, tc 0, flags [none], attrs [none], len 4, completer 00:00, success, byte count 16, requester 1b:00, tag 0x01, lowaddr 0x00`

4-5. NIC reads the packet data to be sent from the main memory (Addr: 0x3bdc1000 is skb->data address)	`MRd, 3DW, tc 0, flags [none], attrs [none], len 25, requester 1b:00, tag 0x01, last 0x3, first 0xf, Addr 0x3bdc1000`
	`CplD, 3DW, WD, tc 0, flags [none], attrs [none], len 25, completer 00:00, success, byte count 98, requester 1b:00, tag 0x01, lowaddr 0x00`

6. NIC generates an interrupt to NIC driver (Addr: 0xfee1a000 is MSI-X address)	`MWr, 3DW, WD, tc 0, flags [none], attrs [none], len 1, requester 1b:00, tag 0x01, last 0x0, first 0xf, Addr 0xfee1a000`

Summary

- NetTLP enables developing PCIe devices in software with IP networking style
  - **NetTLP adapter** is the bridge between PCIe and Ethernet links
  - **LibTLP** enables software PCIe devices on top of IP network stacks
- In the results
  - Observing actual TLPs with tcpdump and Wireshark
  - Implemented the simple Ethernet NIC model in 400 lines of C code
- Benchmarks, other use cases (capturing TLPs from 4 product devices and memory introspection), and their details are available in our paper

Source code and raw pcap data are available at https://haeena.dev/nettlp