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

Yohei Kuga and Ryo Nakamura, *The University of Tokyo;*
Takeshi Matsuya, *Keio University;* Yuji Sekiya, *The University of Tokyo*

https://www.usenix.org/conference/nsdi20/presentation/kuga

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)
February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

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

Yohei Kuga[1], Ryo Nakamura[1], Takeshi Matsuya[2], Yuji Sekiya[1]
[1]*The Univeristy of Tokyo,* [2]*Keio University*

Abstract

Observability on data communication is always essential for prototyping, developing, and optimizing communication systems. However, it is still challenging to observe transactions flowing inside PCI Express (PCIe) links despite them being a key component for emerging peripherals such as smart NICs, NVMe, and accelerators. To offer the practical observability on PCIe and for productively prototyping PCIe devices, we propose NetTLP, a development platform for software PCIe devices that can interact with hardware root complexes. On the NetTLP platform, software PCIe devices on top of IP network stacks can send and receive Transaction Layer Packets (TLPs) to and from hardware root complexes or other devices through Ethernet links, an Ethernet and PCIe bridge called a NetTLP adapter, and PCIe links. This paper describes the NetTLP platform and its implementation: the NetTLP adapter and LibTLP, which is a software implementation of the PCIe transaction layer. Moreover, this paper demonstrates the usefulness of NetTLP through three use cases: (1) observing TLPs sent from four commercial PCIe devices, (2) 400 LoC software Ethernet NIC implementation that performs an actual NIC for a hardware root complex, and (3) physical memory introspection.

1 Introduction

PCI Express (PCIe) is a widely used I/O interconnect for storage, graphic, network, and accelerator devices [16, 24, 25]. Not limited to connect the peripheral devices, some high-performance interconnects adopt the PCIe protocol [5, 22, 35]. Moreover, specifications of future interconnects are designed by extending the PCIe protocol [10, 14]. Such versatility of PCIe is derived from the packet-based data communication and the flexibility of PCIe topology. The PCIe specification defines building blocks comprising PCIe topologies: endpoint, switches, bridges, and root complexes. PCIe packets flow through point-to-point PCIe links between the blocks, and motherboard manufacturers can expand the PCIe topologies with these blocks depending on the use cases.

Table 1: Comparison of platforms for prototyping PCIe devices from the viewpoints of software and hardware.

		PCIe device	
		Software	Hardware
Root Complex	Software	QEMU	–
	Hardware	NetTLP	FPGA/ASIC

By contrast to the spread of PCIe, it is still difficult for researchers and software developers to observe PCIe and prototype PCIe devices, although they are crucial for optimizing performance and developing future PCIe devices. Observing PCIe transactions is difficult because PCIe transactions are confined in hardware. PCIe is not just a simple fat-pipe between hardware elements; it also has several features for achieving high-performance communication, i.e., hardware interrupt, virtualization, and CPU cache operations. Utilizing these features is important for exploiting PCIe efficiently; however, the concrete behaviors of the transactions in PCIe links cannot be determined unless special capture devices for observing PCIe hardware are used.

In addition to the observation, prototyping PCIe devices lacks productivity. Field Programmable Gate Array (FPGA) is a major platform for prototyping PCIe devices [26, 40, 44, 45, 50]. However, developing all parts of a PCIe device on an FPGA still requires significant effort, such as the great devotion of the NetFPGA project [52] for networking devices. Another approach is to adopt virtualization or simulation, e.g., GEM5 [11, 23] and RTL simulators [4, 31]. QEMU [9], which is a famous virtualization platform, can be used for prototyping PCIe devices from the software perspective. QEMU enables researchers and developers to prototype new hardware architecture; however, its environment is fully softwarized. QEMU devices can communicate with only the emulated root complex and cannot communicate with the physical root complex and other hardware connected to the root complex.

The goal of this paper is to bridge the gap between software and hardware for PCIe, as shown in Table 1. Our proposed platform, called NetTLP, offers softwarized PCIe endpoints that can interact with hardware root complexes. By using NetTLP, researchers and software developers can prototype their PCIe devices as software PCIe endpoints and test the software devices with actual hardware root complexes through the PCIe protocol. This hybrid platform of software and hardware simultaneously improves both the observability of PCIe transactions and the productivity of prototyping PCIe devices.

The key technique for connecting softwarized PCIe endpoints to hardware root complexes is to separate the PCIe transaction layer into software and put the software transaction layer on top of IP network stacks. Our FPGA-based add-in card, called NetTLP adapter, delivers Transaction Layer Packets (TLPs) to a remote host over Ethernet and IP networks. The substance of the NetTLP adapter is implemented in software on the IP network stack of the remote host with LibTLP, which is a software implementation of the PCIe transaction layer. The NetTLP platform consisting of the adapter and library enables software PCIe devices on IP network stacks to interact with hardware root complex through the NetTLP adapter. Moreover, TLPs delivered over Ethernet links can be easily observed by IP networking techniques such as tcpdump and Wireshark.

In this paper, we describe NetTLP, the novel platform for software PCIe devices. To achieve the platform, we investigate PCIe from the perspective of packet-based communication (§2) and then describe the approach to connect the software PCIe devices with hardware root complexes and process TLPs in software (§3) and describe its implementation (§4). In addition to micro-benchmarks (§5), we demonstrate three use cases of NetTLP (§6): observing behaviors of a root complex and commercial devices at a TLP-level, prototyping an Ethernet NIC in software interacting with a physical root complex, and physical memory introspection using NetTLP.

The contributions of this paper include the following:

- We propose a novel platform for prototyping PCIe devices in software, while the software devices can communicate with physical hardware such as root complexes, CPU, memory, and other PCIe devices. This platform offers high productivity for prototyping PCIe devices with actual interactions with hardware.

- We provide observability of PCIe transactions confined in hardware by the softwarized PCIe endpoints on the IP network stack. Our modified tcpdump can distinguish the encapsulated TLPs in Ethernet by NetTLP, enabling us to easily capture TLPs in an IP networking manner.

- We present detailed observation results of PCIe transactions with a root complex and commercial peripherals: an Intel root complex, Intel X520 10 Gbps NIC, Intel XL710 40 Gbps NIC, Intel P4500 NVMe, and Samsung PM1724 NVMe. The observation results by NetTLP reveal differences in their behaviors on PCIe transactions, for example, different usage of TLP *tag* fields.

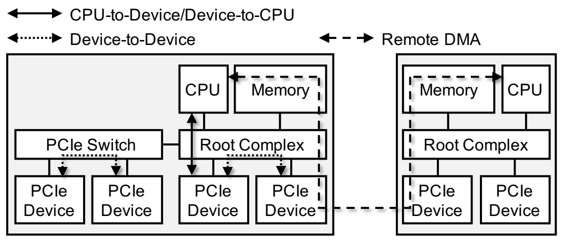


Figure 1: A PCIe topology and three communication models.

- We show a prototype of a nonexistent Ethernet NIC with NetTLP. This prototyping demonstrates the high productivity of the NetTLP platform; the NIC is certainly implemented in software, but it performs as an actual Ethernet NIC for a physical root complex.

- We demonstrate the possibility of developing memory introspection methods on NetTLP without implementing dedicated devices. As a proof-of-concept, we implemented two applications that gather process information from a remote host by DMA through NetTLP.

All source codes for hardware and software and captured data described in this paper are publicly available [1].

2 Background

PCIe is not only an interconnect, but also a packet-based data communication network. As with IP networks, PCIe has a layering model composed of a physical layer, a data link layer, and a transaction layer. The data link layer delivers PCIe packets across one hop over a PCIe link, while the transaction layer is responsible for delivering TLPs from a PCIe endpoint to a PCIe endpoint across the PCIe links. PCIe interconnect is composed of the following elements that are capable of supporting the layer functionalities: endpoints, switches, bridges, and root complexes. PCIe switches and root complexes route and forward PCIe packets in accordance with the addresses in memory-mapped I/O (MMIO) space or requester IDs. Any functionalities of PCIe stand at the packet-based communication, e.g., MSI-X for hardware interrupts is implemented by memory writes to specific memory addresses. Because of being such a packet-based network, PCIe topologies and their communication models are flexible, as depicted in Figure 1.

In IP networks, we can easily prototype and implement any part of the networks, such as end hosts, switches, and routers, and observe packets flowing in the networks; however, PCIe cannot do such things despite PCIe also being a packet-based network. PCIe was originally designed for I/O interconnects inside a computer; therefore, it is assumed that all the PCIe elements were implemented in hardware. This assumption and the current situation cause difficulty in investigating and developing PCIe and its elements.

For investigating PCIe, there are two major platforms: FPGA and QEMU. FPGA offers programmability on hardware for prototyping PCIe devices. By contrast, developing PCIe devices on FPGA still involves significant effort. Even when implementing a device, the device requires purpose-specific logic and various logic blocks such as PCIe core, DMA engine, memory controller, etc. Such functional blocks are not available, unlike software libraries. Moreover, we cannot observe the PCIe packets sent by the FPGA. We can see only part of the signals using logic analyzers, or expensive dedicated hardware. On the other hand, QEMU enables implementing PCIe devices on a full virtualized environment [15, 37]. However, QEMU does not implement the PCIe protocol, only DMA APIs. QEMU is used as a platform for researching new PCIe devices and discussing OS abstractions and implementations without real hardware and the PCIe protocol. Thus, QEMU PCIe devices cannot interact with the real host and hardware on the PCIe, although the features of root complexes have been evolving.

The two platforms have advantages and disadvantages: FPGA requires significant effort for prototyping and lacks observability, while QEMU devices cannot interact with hardware elements with the PCIe protocols. These disadvantages are because the platforms focus on only hardware or software. Root complexes and devices—the two major elements of PCIe—are hardware in FPGA or software in QEMU.

As a third platform, we advocate connecting software and hardware elements of PCIe. If PCIe devices are moved to software and connected to a hardware root complex, we achieve productive PCIe device prototyping in software and interactions with the real PCIe elements connected to the hardware root complex. Moreover, we can observe the PCIe transactions at the software PCIe device without resorting to dedicated hardware mechanisms. This relationship is similar to IP networks; IP network stacks at end hosts are software, while routers and switches are hardware.

3 NetTLP

To feasibly connect software PCIe devices and hardware root complexes, we propose to separate the transaction layer of PCIe into software, as illustrated in Figure 2. The transaction layer is responsible for the fundamental part of end-to-end PCIe communications: identifiers, i.e., memory addresses and requester IDs, routing, and issuing PCIe transactions. The softwarized transaction layer offers high productivity of PCIe device prototyping in software on top of the layer and observability of PCIe transactions by software.

To connect the softwarized transaction layer and the hardware data link layer, NetTLP has chosen a configuration that bridges a PCIe link and an Ethernet link. Because PCIe and Ethernet are packet-based networks, it is possible to deliver TLPs over Ethernet links by encapsulation. Once TLPs go to an Ethernet network, we can easily observe the TLPs like

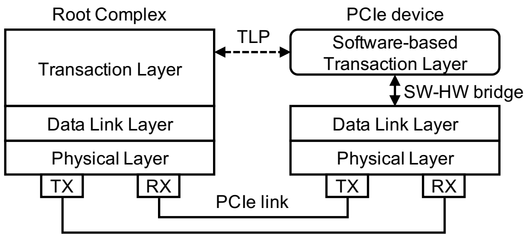


Figure 2: The layering model of PCIe and our approach that separates the transaction layer into software.

IP packets, implement the transaction layer in software, and prototype PCIe devices on top of software IP network stacks.

As with NetTLP, ExpEther [47, 48] and Thunderclap [29] also enable observing and manipulating TLPs. ExpEther extends PCIe links by delivering TLPs over Ethernet links. TLPs encapsulated by ExpEther would be observed on an Ethernet link between an ExpEther adapter and a hardware extension box in which peripheral devices are installed. In Thunderclap, Linux running on an ARM CPU on an FPGA processes TLPs with software. Similarly, some smart NICs can send and receive TLPs from CPUs on the NICs with abstracted DMA APIs [30, 33].

In contrast to the existing technologies, NetTLP focuses on prototyping new PCIe devices in software. For this purpose, manipulating TLPs with software is one of the essential functionalities. In addition, how devices and CPUs interact with each other must be designed flexibly. ExpEther does not focus on this point so that it extends PCIe links over Ethernet, and the software PCIe devices on Thunderclap pretend existing devices to reveal vulnerabilities through their drivers. In the NetTLP platform, researchers and developers can design how new software devices interact with CPUs through root complexes. More specifically, it is possible to design and implement the usage of registers of the software devices, e.g., descriptor rings, from scratch on the NetTLP platform. This functionality enables designing and implementing nonexistent devices and observing its interaction in software (Section 6.2).

3.1 Platform Overview

A key component of NetTLP delivering TLPs over Ethernet is a NetTLP adapter, which is an FPGA-based add-in card equipped with a PCIe link connected to the host and an Ethernet link. Another key component is LibTLP, which is a software library of the PCIe transaction layer on the IP network stack. The NetTLP platform is composed of two hosts, an adapter host having the NetTLP adapter and a device host where LibTLP-based applications are run, as illustrated in Figure 3.

The NetTLP adapter is responsible for the bridge between the hardware data link layer and the software transaction layer depicted in Figure 2. The NetTLP adapter delivers TLPs be-

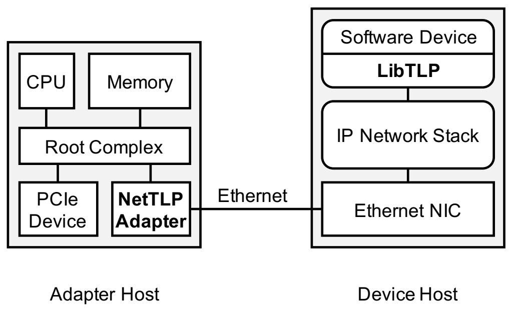


Figure 3: The overview of the NetTLP platform.

tween a host's PCIe link and the Ethernet link. When the NetTLP adapter receives TLPs from the PCIe link, the NetTLP adapter encapsulates each TLP in Ethernet, IP, UDP, and NetTLP header for sequencing and timestamping and sends the packets to the device host via the Ethernet link. When the NetTLP adapter receives a UDP packet from the Ethernet link, the NetTLP adapter checks whether the packet's payload is a TLP, decapsulates the packet, and sends the inner TLP to the PCIe link. As a result, from the perspective of the adapter host, all TLPs sent from the device host by the software are recognized as TLPs generated by the NetTLP adapter.

LibTLP implements the PCIe transaction layer in software and provides abstracted DMA APIs for applications. The applications on the device host can send and receive TLPs to the NetTLP adapter on the adapter host through UDP sockets. By using LibTLP, researchers and software developers can implement their own PCIe devices in software that perform actual behaviors of the NetTLP adapter for the root complex on the adapter host. In addition, splitting software PCIe devices and physical adapters on the distant hosts enables us to observe actual PCIe transactions flowing through the Ethernet link. We can capture the encapsulated TLPs by tcpdump at the device host or capture the TLPs on the Ethernet link by optical taps or port mirroring on Ethernet switches.

Although a NetTLP adapter is a single peripheral device, the NetTLP adapter can be applied to some PCIe communication models in PCIe topologies organized in Figure 1. Naturally, the NetTLP adapter and the software PCIe device can become a device on CPU-to-device and device-to-CPU communications. Applying the NetTLP adapter into device-to-device communication, also known as peer-to-peer DMA, realizes interactions between commercial PCIe devices and software PCIe devices. Section 6.1 shows TLPs sent from product devices by the NetTLP platform and peer-to-peer DMA integration. In addition, the NetTLP adapter can be considered a raw remote memory access device. Applications on the device host can issue DMA to any address of the memory on the adapter host through the NetTLP adapter. Section 6.3 demonstrates memory introspection methods exploiting the remote memory access by NetTLP.

3.2 TLP Processing in Software

Processing PCIe transactions in software is challenging because PCIe was originally designed to be processed by hardware. This section describes three issues to design NetTLP for achieving PCIe interactions between hardware and software.

Receiving burst TLPs: The first issue is that LibTLP needs to receive burst TLPs sent from the hardware. The minimum TLP length is 12 bytes when the TLP is a memory-read (MRd) TLP with the 32-bit address field, for instance. NetTLP adapter encapsulates TLPs with Ethernet, IP, UDP, and NetTLP headers; thus, the minimum encapsulated packet length is 64 bytes. This length is the same length as the minimum packet size of IP networks. Meanwhile, the flow control of PCIe is based on the credit system [7], and PCIe endpoints can send TLPs continuously as long as the credit remains. In particular, PCIe devices often send small TLPs using the TLP tag field to achieve high performance [21]. Because these TLP transmission intervals are continuous clock units, the throughput of encapsulated TLPs could momentarily be wire-rate on the Ethernet link with short packets.

To receive such burst TLPs by software, NetTLP exploits the TLP tag field to distribute receiving encapsulated TLPs among multiple hardware queues of an Ethernet NIC and CPU cores. The tag field is used to distinguish individual non-posted transactions that can be processed independently. The NetTLP adapter embeds the lower 4-bit of the tag values into the lower 4-bit of UDP port numbers when encapsulating TLPs. As a result, PCIe transactions to the NetTLP adapter are delivered through different UDP flows based on the tag field, and the device host can receive the flows by different NIC queues. This technique, called tag-based UDP port distribution, enables the software side to process TLPs on multiple cores associated with multiple NIC queues.

Completion Timeout: Another issue is the completion timeout. In accordance with PCIe specifications, root complexes and PCIe endpoints support a completion timeout mechanism on the PCIe transaction layer. When a requester send memory-read requests, the requester sets timeout periods for each request, and the completer need to send the completions within the periods. Hence, software PCIe devices on the NetTLP platform need to be able to send completions within the hardware-level timeout periods. The timeout period is configured in the PCIe configuration space. For instance, the minimum and maximum completion timeout periods of X520 NIC are 50 microseconds and 50 milliseconds, respectively [6]. Therefore, software PCIe devices built on LibTLP also must be capable of replying memory requests during such periods. Fortunately, Linux network stacks on general server machines are not too slow at the millisecond scale; therefore, we expect that LibTLP would meet the requirement. Section 5 examines this issue through a latency benchmark.

Encapsulation Overhead: To take TLPs to software on top of IP network stacks, NetTLP encapsulates TLPs with IP headers although encapsulation involves throughput reduction due to the header overhead. NetTLP encapsulates TLPs into multiple headers: 14-byte Ethernet, 4-byte FCS, 20-byte IP, 8-byte UDP, and 6-byte NetTLP headers. The throughput of data transfer over DMA on the NetTLP platform can be calculated with:

$$Throughput_{dma} = BW_{eth} \times \frac{TLP\_Data}{TLP\_hdr + TLP\_Data + Pkt\_Hdr + ETH\_Gap}$$

BW_{eth} is the Ethernet link speed, Pkt_Hdr is 52-byte for the headers and FCS mentioned above, and ETH_Gap is 20-byte for preamble and inter-frame gaps. From this formula, throughput with 256-byte TLP_Data (usual max payload size) and 12-byte TLP_hdr for 3DW memory-write TLP on 10 Gbps links is approximately 7.53 Gbps, which is the theoretical limitation with 10 Gbps Ethernet. Although throughput is required depending on use cases, the overhead is not significant for just prototyping software PCIe devices.

4 Implementation

We implemented the NetTLP adapter using an FPGA card and LibTLP on Linux. This section describes the details of the NetTLP adapter, APIs provided by LibTLP, hardware interrupts, and limitations of the NetTLP platform.

4.1 NetTLP Adapter

The NetTLP adapter was implemented using the Xilinx KC705 FPGA development board [51]. This board has Xilinx Kintex 7 FPGA, an Ethernet 10 Gbps port, and a PCIe Gen 2 4-lane link. We used the board because its PCIe Endpoint IP core enables user-defined logic to handle raw TLP headers. This feature is suitable for designing the NetTLP adapter. However, the Xilinx's newer PCIe IP core, which supports PCIe Gen 3, does not allow user-defined logic to handle raw TLP headers. Therefore, the current implementation of the NetTLP adapter does not support PCIe Gen 3.

Figure 4 shows the overview of the circuit diagram of the NetTLP adapter. The current NetTLP adapter has three base address register (BAR) spaces for different roles. BAR0 is used to configure the NetTLP adapter. The configurations through BAR0 support changing source and destination MAC addresses and source and destination IP addresses for encapsulating TLPs. The BAR2 space is used for the MSI-X table to support the hardware interrupts from software PCIe devices. The detail of MSI-X in NetTLP is described in Section 4.3. Both BAR0 and BAR2 memory spaces are implemented with Block RAM on the FPGA, and the NetTLP adapter has the Peripheral I/O (PIO) engine above the BAR0 and BAR2 to reply with completion TLPs for operations to the BARs.

BAR4 is different from BAR0 and BAR2; BAR4 space is connected to the Ethernet PHY and not connected to the PIO engine. All TLPs from the root complex or other devices to

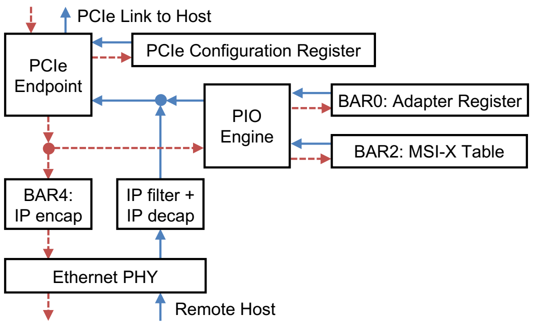


Figure 4: The circuit diagram of the NetTLP adapter.

the BAR4 space are encapsulated in Ethernet, IP, UDP, and NetTLP headers and transmitted to an external host via the Ethernet link. Namely, LibTLP on the device host communicates to the root complex on the adapter host through the memory region assigned to the BAR4.

When encapsulating TLPs to the BAR4, source and destination port numbers of the UDP headers are generated based on the tag field of their TLP headers. This is the tag-based UDP port distribution described in Section 3.2. In the current implementation, the UDP port numbers are generated with $0x3000 + (TLP\_Tag \wedge 0x0F)$. Thus, the NIC on the device host receives the TLPs by a maximum of 16 hardware queues. When the NetTLP adapter receives UDP packets from the device host, the IP filter logic checks whether the IP addresses match the configured addresses on BAR0. If the IP addresses and port numbers are correct, the packets are decapsulated, and the inner TLPs are sent to the host via the PCIe link.

The driver for the NetTLP adapter depends on types of software PCIe devices. If a software PCIe device is an Ethernet NIC, the driver is for the Ethernet NIC, and if a software PCIe device is an NVMe SSD, the driver is for the NVMe SSD. Regardless of the driver types, we implemented a simple driver that supports basic functionalities for the NetTLP adapter. This driver enables the NetTLP adapter, initializes MSI-X, and prepares a simple messaging API. The software PCIe device on the device host can obtain information about the NetTLP adapter, i.e., addresses of the BAR spaces of the adapter, PCIe bus and slot numbers, and MSI-X table. Users of the NetTLP platform can develop drivers for their software PCIe devices by extending the basic NetTLP driver.

4.2 LibTLP

LibTLP is a userspace library that implements the PCIe transaction layer. On top of the transaction layer implementation, the LibTLP provides a well-abstracted DMA API and a callback API for handling each type of TLPs.

Figure 5 shows the DMA API of LibTLP. A LibTLP instance that contains a socket descriptor, a tag value, and a destination address of a target NetTLP adapter is represented

```
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);
```

Figure 5: The DMA API of LibTLP.

by a `nettlp` structure initialized by `nettlp_init()`. The DMA APIs for DMA reads and DMA writes are invoked by specifying a `nettlp` structure. As with the `read()` and `write()` system calls, `dma_read()` attempts to read up to `count` bytes into `buf` and `dma_write()` writes up to `count` bytes from `buf`. `addr` indicates a target address of a DMA transaction. The return values of the functions are the number of bytes read or written, or -1 and `errno` is set on error. For the DMA reads, applications can notice TLP loss or completion errors through the return value and `errno`.

In addition to the DMA API that issues DMAs to the memory on the adapter host, LibTLP provides a callback API for handling TLPs from the adapter host to the device host. The callback API allows applications to register functions for major TLP types: memory read (MRd), memory write (MWr), completion (Cpl), and completion with data (CplD). When the sockets of the `nettlp` structures receive TLPs, the registered functions are invoked for the TLPs. By using this API, the applications on the device host can respond to MRd TLPs from the root complex to the BAR4 on the NetTLP adapter by sending associated CplD TLPs, for instance.

4.3 Hardware Interrupt

For hardware interrupt, the NetTLP platform supports MSI-X, which is widely used by modern PCIe devices. MSI-X interrupt is invoked by sending a MWr TLP with particular data to a specified address from a device. The address and data for the interrupt are stored in an MSI-X table on a BAR space specified by the PCIe configuration space of the device. In other words, to send a hardware interrupt by MSI-X, it is necessary to refer to the MSI-X table on the BAR.

To achieve MSI-X on the NetTLP platform, there are two approaches: (1) placing the MSI-X table on the BAR4 and a software PCIe device on a device host holds the MSI-X table, and (2) placing the MSI-X table on other BAR spaces under the PIO engine and a software PCIe device on a device host gets the MSI-X table through other communication paths. The current implementation chooses the latter approach. The former approach does not need any other communication paths to obtain the MSI-X table from the adapter host. However, the MSI-X table is initialized by the NetTLP driver, so the software PCIe device must run on the device host before the NetTLP driver is loaded on the adapter host. Moreover, software PCIe device implementations must always be capable of maintaining the MSI-X table, even if they do not use MSI-X. These characteristics might inconvenience the development of software PCIe devices. For these reasons, we placed the MSI-X table on BAR2 under the PIO engine that is controlled by only the hardware logic of the NetTLP adapter. The software PCIe devices can obtain the content of the MSI-X table through the simple messaging API provided by the basic NetTLP driver.

4.4 tcpdump and Wireshark

To observe TLPs, we slightly modified tcpdump and implemented a Wireshark plugin. In the NetTLP platform, the encapsulated TLPs flow through the Ethernet link between the NetTLP adapter and the device host; hence, the TLPs can be easily captured by the monitoring tools of IP networking. The modified tcpdump can recognize the encapsulated TLPs and display the contents of TLPs on the popular tcpdump output. The Wireshark plugin also displays the contents of TLPs on the GUI. These tools offer researchers and developers a convenient method to observe TLPs.

4.5 Limitations

The current implementation of the NetTLP platform cannot handle the PCIe configuration space. The PCIe configuration space manages properties of the PCIe device such as Device ID, Vendor ID, and address regions of BAR spaces. The PCIe configuration space is stored in the memory of the PCIe device hardware. When the host boots up or re-scans PCIe devices, the devices use the CfgRd and CfgWr TLPs to communicate with the root complex to set up their PCIe configurations. In the current implementation of NetTLP adapter, the PCIe Endpoint IP core for Kintex 7 FPGA manages the PCIe configuration space as shown in Figure 4. The IP core does not allow user-defined logic to manipulate the configuration registers by raw TLPs. Therefore, the software PCIe devices cannot see and change their PCIe configurations. As a result, the current implementation does not support functionalities that require the manipulation of PCIe configuration registers, i.e., changing structures of MSI-X tables and SR-IOV.

5 Micro-benchmarks

To estimate performances of the software PCIe devices and applications, we conducted micro-benchmarks for the throughput and latency of DMAs on the NetTLP platform.

In the NetTLP platform, there are two directions of PCIe transactions: (1) from LibTLP to the NetTLP adapter, and (2) from the NetTLP adapter to LibTLP. They represent DMA reads and writes from a software PCIe device to a root complex, and DMA reads and writes from a root complex to a software PCIe device, respectively. In the former direction, we assume that PCIe transactions issued from the software PCIe device can be processed without packet loss because all of the components on the adapter host is hardware, which has higher bandwidth (the 16 Gbps PCIe Gen 2 4-lane link of the NetTLP adapter) than the 10 Gbps Ethernet link. In

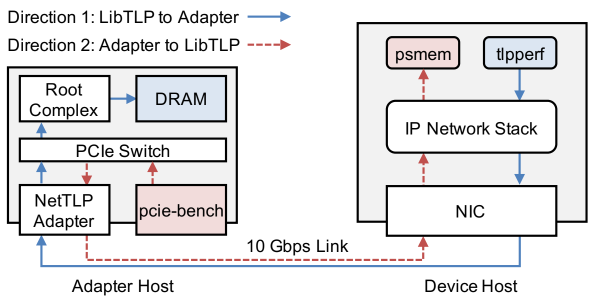


Figure 6: Benchmark setup.

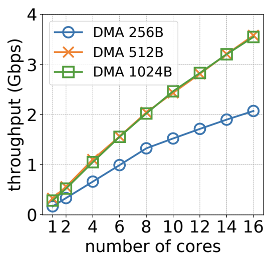


Figure 7: DMA Read throughput from LibTLP to the NetTLP adapter versus the number of CPU cores.

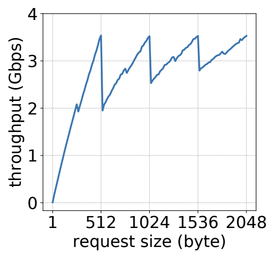


Figure 8: DMA Read throughput from LibTLP to the NetTLP adapter versus the request sizes.

the opposite direction, DMA reads from the root complex to the software PCIe devices also would not be dropped because the root complex does not send a memory read request until receiving a completion for the last read request (non-posted transaction). Based on this assumption, we measured throughput of DMA reads and writes from LibTLP (Section 5.1), and DMA reads from the NetTLP adapter (Section 5.2).

By contrast, the throughput of DMA writes from the root complex to the software PCIe device cannot be measured. The root-complex can send MWr TLPs up to the link speed of the NetTLP adapter without explicit acknowledgment (posted transactions). The current NetTLP adapter does not have mechanisms to notify congestion on the Ethernet link to the root complex; therefore, MWr TLPs are dropped if the 10 Gbps Ethernet link of the NetTLP adapter overflows. Notifying the overflow to the root complex and other devices is a future work. However, for recent peripherals such as Ethernet NICs and NVMe SSDs, usual use cases of memory writes to PCIe devices are updating registers on the devices from CPUs, and these do not require significant throughput. Therefore, we argue that the current NetTLP adapter is sufficient to prototype PCIe devices in software.

Figure 6 depicts the two directions and the components we used to generate PCIe transactions for the benchmarks. To generate PCIe transactions from LibTLP to the NetTLP adapter, we developed a LibTLP-based benchmark application called tlpperf. Users can send memory read and write requests to the memory on the adapter host through the NetTLP adapter from the device host by using tlpperf. For generating PCIe transactions in the opposite direction, we implemented a LibTLP-based pseudo memory device, called psmem, and slightly modified the pcie-bench [34]. psmem on the device host pretends a memory region associated with the BAR4 of the NetTLP adapter. As described in Section 4.1, TLPs to the BAR4 space of the NetTLP adapter are transmitted to the device host. When psmem receives a MWr TLP, psmem saves the data and the associating address. When psmem receives a MRd TLP, psmem sends CplD TLP(s) with proper data associating the requested address. In addition, to generate memory requests to the BAR4 of the NetTLP adapter, we modified pcie-bench implementation for NetFPGA-SUME. The modified pcie-bench can use the BAR4 space as the benchmark destination instead of main memory.

For the micro-benchmark, we prepared two machines for the adapter and device hosts. The adapter host was an Intel Core i9-9820X 10 core CPU and 32 GB DDR4 memory with an ASUS WS X299 SAGE motherboard. This motherboard has PCIe switches. The NetTLP adapter and the NetFPGA-SUME card for pcie-bench were installed on PCIe slots under the same PCIe switch. The device host was an Intel Core i9-7940X 12 core CPU, 32 GB DDR4 memory, and Intel X520 10 Gbps NIC with an ASUS PRIME X299-A motherboard. The device host was connected to the NetTLP adapter on the adapter host via a 10 Gbps Ethernet link. OSes were Linux kernel 4.20.2. Note that we enabled hyperthreading on the device host that had 12 physical cores to fully utilize 16 NIC queues by the tag-based UDP port distribution.

In the experiments described in this section, all the throughput results are goodput. The throughput does not include TLP and encapsulation headers. In addition, all memory requests in each iteration access the same address. We measured throughput and latency with random and sequential access patterns; however, there were no differences because of the memory access patterns in any experiment. The processing time for the software part is relatively dominant and obscures differences in performances because of the memory access patterns.

5.1 LibTLP to NetTLP Adapter

In the first benchmark, we measured the throughput of PCIe transactions from LibTLP to the memory on the adapter host through the NetTLP adapter. It is expected that the throughput would be limited by Linux kernel network stack performance where tlpperf runs because the data path on the adapter host is fully hardware and its links are the 16 Gbps PCIe Gen 2 4-lane link and the 10 Gbps Ethernet link.

Figure 7 shows the throughput of DMA reads issued by tlpperf on the device host. The key indicates request sizes of each DMA read request. As shown, the throughput linearly increases along with the number of CPU cores. This result

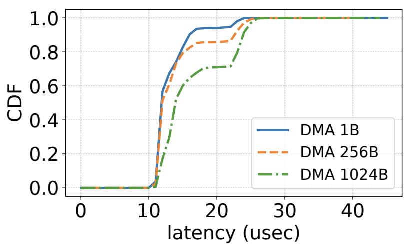


Figure 9: DMA Read latency from LibTLP to the NetTLP adapter.

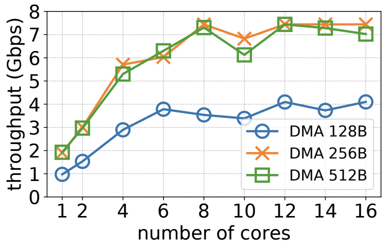


Figure 10: DMA Write throughput from LibTLP to the NetTLP adapter versus the number of CPU cores.

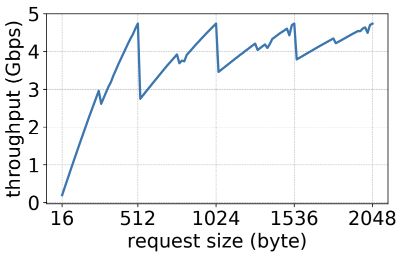


Figure 11: DMA Read throughput from the NetTLP adapter to LibTLP.

indicates that the tag-based UDP port distribution technique successfully utilizes multiple queues and multiple cores on the Linux-based device host. On the other hand, the read request size greater than 512-byte does not contribute to the throughput because the maximum read request size (MRRS) is 512-byte. The maximum throughput in this direction, which is approximately 3.6 Gbps, is less than the PCIe Gen 2 x1 link speed; however, the required throughput depends on applications and use cases. For example, Section 6 demonstrates use cases not depending on throughput. Note that the current LibTLP uses Linux Socket API; therefore, LibTLP would handle more UDP traffic with high-speed network I/O [28, 39].

Next, we measured the throughput of DMA reads from tlpperf with 16 cores while increasing the read request sizes by 16 bytes. The result shown in Figure 8 represents the *saw-tooth pattern*, which is also noted in the pcie-bench paper [34]. The saw-tooth pattern is caused by the packetized structure of the PCIe protocol. MRRS is 512-byte; therefore, when the request size is not a multiple of 512, the remaining bytes are transferred by a small size memory read. Such small-sized TLPs cause throughput reduction. Sizes of the small TLPs increase as the request sizes increase, so the throughput also increases until the request size exceeds the next multiple of 512. Slight reductions of the throughput after multiples of 256-byte are caused by the maximum payload size (MPS), which is 256-byte, in a similar manner. This result where the saw-tooth pattern appears as well as the hardware-based measurement by pcie-bench indicates that LibTLP correctly implements the packetization of the PCIe protocol.

In addition to the throughput, we measured the latency for DMA reads. The PCIe specification defines completion timeout; thus, evaluating the DMA read latency is crucial for prototyping PCIe devices in software. Figure 9 shows the result of 10000 DMA reads with 1-byte, 256-byte, and 1024-byte read requests generated by tlpperf. The latency increases with the request sizes; however, 99% latency is less than 27 microseconds regardless of the request sizes, and the maximum latency is 45 microseconds with 1024-byte DMA reads. These results correspond to the completion timeout range A (50 us to 10 ms). Therefore, we argue that prototyping PCIe devices in software with hardware root complexes is feasible from a latency perspective. According to the pcie-bench [34], DMA read latency inside a physical host is from 400 to 800 nanoseconds. Consequently, software processing for the network stack and the tlpperf application on the device host is dominant in the latency of the NetTLP platform.

We next measured the throughput of DMA writes from LibTLP. In contrast to DMA reads, DMA writes are posted transactions; therefore, we cannot measure the latency and throughput of DMA writes correctly. In this experiment, tlpperf calculates throughput when MWr TLPs are written to sockets. Figure 10 shows this measurement result. In addition to the DMA read results, DMA writes can also effectively use multiple cores and queues. In contrast to DMA reads, DMA writes reach the upper throughput with 256-byte DMA writes because MPS is 256-byte. Note that this throughput can be considered as transmitting throughput for UDP sockets of the Linux network stack.

5.2 NetTLP Adapter to LibTLP

For the second direction, we measured the throughput by generating PCIe transactions from the pcie-bench on the adapter host to the psmem running on the device host. Figure 11 shows the DMA read throughput on this direction. The result also represents the saw-tooth pattern as well as the opposite direction. Moreover, the thing that pcie-bench works with the software memory device demonstrates that the NetTLP platform can prototype one of the PCIe devices in software. Besides, the maximum throughput is approximately 4.7 Gbps. We confirmed that pcie-bench used TLP tag values from 0x00 to 0x17; thus, psmem with LibTLP could utilize the 16 cores in parallel by the tag-based UDP port distribution.

Table 2 shows DMA read latency from the pcie-bench on the adapter host to the psmem on the device host. We measured 100000 DMA reads for each request size. As shown, there are no significant differences by request sizes, unlike the original pcie-bench evaluation in hardware. This result is because the software processing on the device host—receiving and sending UDP packets—is dominant. However, the latency also meets the completion timeout range A.

Table 2: DMA Read latency from the pcie-bench to the psmem (microseconds).

Request size	Min	Median	Max	Stddev
256	14.312	17.268	87.456	1.321
512	12.2	18.764	68.552	1.550
1024	12.256	20.06	52.608	1.685
2048	11.612	18.588	68.224	2.385

6 Use Cases

This section demonstrates three use cases of NetTLP. We (1) observed specific behaviors of a commercial root complex and peripherals by capturing TLPs, (2) implemented a theoretical model of an Ethernet NIC as an actual NIC, and (3) demonstrate memory introspection for physical machines. All observations and demonstrations in this section were conducted on the same machines used in the micro-benchmarks.

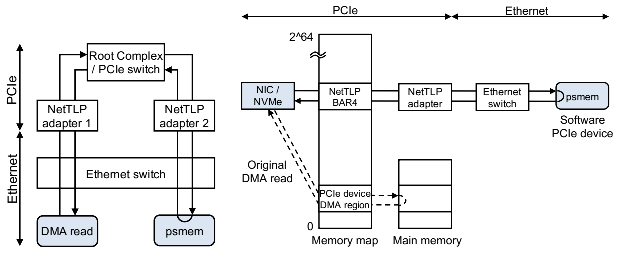
6.1 Capturing TLPs

As the first demonstration, we observe PCIe transactions of a commercial root complex, two Ethernet NICs, and two NVMe SSDs by capturing TLPs. The NetTLP adapter delivers TLPs over Ethernet links; thus, NetTLP enables us to analyze TLPs by using powerful IP network software with UNIX commands, i.e., tcpdump. Besides, the flexibility of PCIe topologies enables us to adapt NetTLP to observe various PCIe transactions issued and processed by different elements.

6.1.1 Root Complex and PCIe Switch

The first observation is to clarify the behavior of root complex. The PCIe specification does not allow PCIe switches to modify PCIe packets during switching. However, root complexes are permitted to split a PCIe packet into small PCIe packets when routing the PCIe packets between PCIe devices. The specification does not describe detailed mechanisms of TLP splitting on peer-to-peer device communication by root complexes. Although TLP splitting may negatively affect performance, its behavior depends on each root complex implementation, and observing the behavior is difficult. As a demonstration, we clarify this point by comparing actual TLPs through a root complex or a PCIe switch captured by NetTLP.

To capture the TLPs, we prepared two NetTLP adapters under the root complex or the PCIe switch on the machine used in the micro-benchmark. Figure 12a shows the topology for this observation. In the test scenario, a DMA read application on the device host sent a 512-byte MRd TLP to psmem through the two NetTLP adapters on the adapter host, and psmem returned CplD TLPs. Moreover, we switched the intermediate element from the PCIe switch to the root complex by changing PCIe slots where the NetTLP adapters were



(a) Root Complex. (b) NIC and NVMe devices.

Figure 12: Two topologies for capturing TLPs. We captured TLPs by port mirroring on the Ethernet switch.

installed. On this topology, we captured TLPs before and after passing through the PCIe switch or root complex by port mirroring on the Ethernet switch.

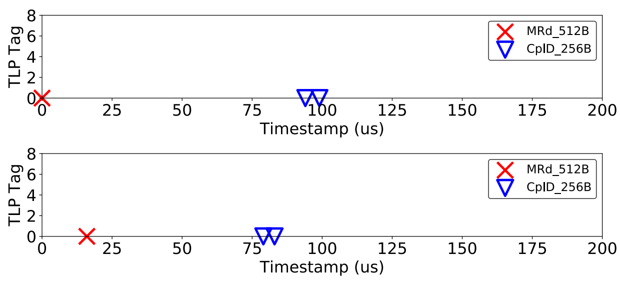
Figure 13 shows the captured TLPs. The x-axis indicates timestamps when a capture machine captured the TLPs from the mirror port. Note that the timestamps were stamped by NIC hardware so that the accuracy was on the nanosecond scale. The y-axis indicates TLP tag values of the TLPs. The TLPs were captured twice: before and behind the PCIe switch or root complex. The graphs on the upper row and lower row show the TLPs captured on the links connected to the NetTLP adapter 1 and adapter 2 depicted in Figure 12a, respectively.

Figure 13a confirms that the PCIe switch does not modify the TLPs as expected. The DMA read application sent a 512-byte MRd TLP, and psmem returned two 256-byte CplD TLPs. By contrast, Figure 13b reveals that the root complex split a 512-byte MRd TLP into eight 64-byte MRd TLPs with different TLP tag values when routing the TLPs between the NetTLP adapters. psmem returned eight 64-byte CplD TLPs, and the root complex rebuilt two 256-byte CplD TLPs from the small CplD TLPs. As a result, the DMA read application received the expected CplD TLPs that are aligned with MPS.

6.1.2 Ethernet NIC and NVMe SSD

Next, we measured and compared TLPs generated by commercial NIC and NVMe devices. Knowledge of how product devices use TLPs would be a useful guideline for developing PCIe devices with high performance. General peripheral devices communicate with the CPU by DMA to the main memory. To capture the TLPs from the devices, we used modified netmap drivers [39] for Ethernet NICs and a modified UNVMe [32] for NVMe SSDs to change the DMA address from main memory to the BAR4 of the NetTLP adapter. As a result, NetTLP enables capturing the TLPs sent from the devices on the Ethernet link connected to the NetTLP adapter.

For observing various behaviors of PCIe devices, we prepared different types and speeds of devices: Intel X520 and XL710 NICs, and Intel P4600 and Samsung PM1725a NVMe SSDs. Throughput of the devices are as follows: Intel X520 is


(a) A 512-byte memory read via the PCIe switch.

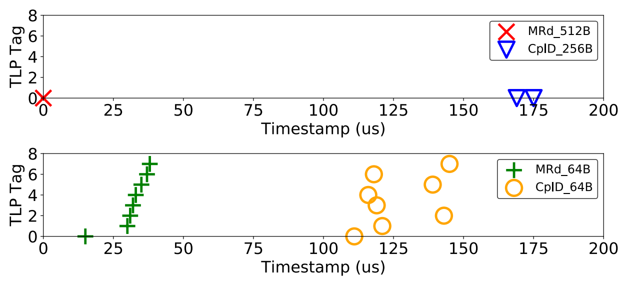
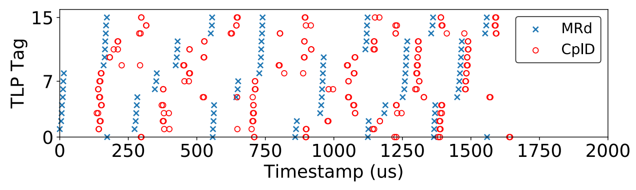
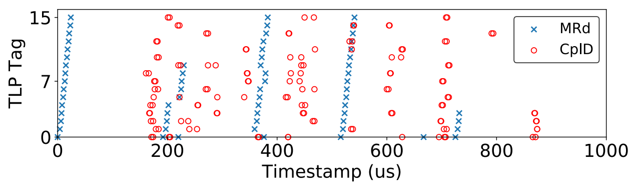
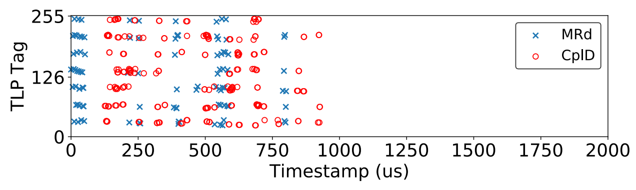

(b) A 512-byte memory read via the root complex.

Figure 13: Comparison of captured TLPs of DMA read across the PCIe switch or the root complex. The graphs on the lower row indicate the captured TLPs behind the PCIe switch or the root complex.


(a) NIC X520 (PCIe Gen2 8-lane, 10 Gbps).


(b) NVMe P4600 (PCIe Gen3 4-lane, Seq write 1575 MB/s).


(c) NIC XL710 (PCIe Gen3 8-lane, 40 Gbps).

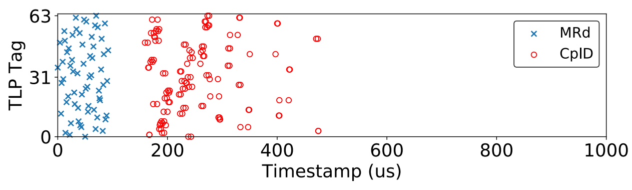

(d) NVMe PM1725a (PCIe Gen3 8-lane, Seq write 2600 MB/s).

Figure 14: Comparison of tag field usage of the NIC and NVMe devices.

a 10 Gbps Ethernet NIC, Intel XL710 is a 40 Gbps Ethernet NIC, the sequential write speed of the Intel P4600 NVMe device is 1575 MB/s, and the sequential write speed of the Samsung PM1725 is 2600 MB/s. Figure 12b shows the experimental setup of this observation. In this setup, NIC or NVMe and the NetTLP adapter were installed in PCIe slots under the same PCIe switch. The devices sent MRd TLPs for sending packets or writing data to the NVMe SSDs, and the MRd TLPs were delivered to psmem. psmem then returned CplD TLPs with Ethernet frames prepared in advance for the NIC scenario or zero-filled data for the NVMe scenario. For the NIC scenario, the NICs sent 32 1500-byte packets, and for the NVMe scenario, the NVMe SSDs wrote 32-MB data to the SSDs. Note that the block size of the Intel P4600 is 512 bytes and that of the Samsung PM1725a is 4096 bytes; therefore, we adjusted the number of NVMe write commands to write 32-MB data. To capture the TLPs, we used port mirroring on the Ethernet switch between the NetTLP adapter and the device host where psmem runs as well as the last experiment.

Figure 14 shows the result of captured TLPs of the NIC and NVMe devices. The result reveals that each PCIe device uses the TLP tag differently. X520 and P4600 use tag values from 0 to 15, PM1725a uses values from 0 to 63, and XL710 uses values from 24 to 249. The PCIe link speeds have been improved along with generations of PCIe; however, MPS has hardly improved. As a result, these PCIe devices improve data transfer throughput by sending memory requests continuously by leveraging TLP tags. The numbers of used tag values increase along with the desired throughput of the devices. According to the latency measurement in pcie-bench [34], the latency of a 512-byte DMA read is approximately 580 nanoseconds. The calculated throughput from this latency is about 7 Gbps when not using the tag field. Therefore, exploiting the tag field well is an important matter to achieve the desired throughput as this observation revealed.

6.2 Prototyping an Ethernet NIC

To confirm that NetTLP can prototype PCIe devices in software, we implemented an Ethernet NIC as a proof-of-concept on the NetTLP platform. The target NIC we implemented is simple NIC introduced by pcie-bench [34]. The original

```
23:51:48.210015 IP adapter.12291 > libtlp.12291: NetTLP: MWr, len 1, requester 00:00, tag 0x03, last 0x0, first 0xf, Addr 0xa0000010
23:51:48.210055 IP libtlp.12291 > adapter.12291: NetTLP: MRd, len 4, requester 1b:00, tag 0x03, last 0xf, first 0xf, Addr 0x2f001000
23:51:48.210069 IP adapter.12291 > libtlp.12291: NetTLP: CplD, len 4, completer 00:00, success, bc 16, requester 1b:00, tag 0x03, lowaddr 0x00
23:51:48.210083 IP libtlp.12291 > adapter.12291: NetTLP: MRd, len 25, requester 1b:00, tag 0x03, last 0x3, first 0xf, Addr 0x3bb26800
23:51:48.210095 IP adapter.12291 > libtlp.12291: NetTLP: CplD, len 25, completer 00:00, success, bc 98, requester 1b:00, tag 0x03, lowaddr 0x00
23:51:48.210122 IP libtlp.12291 > adapter.12291: NetTLP: MWr, len 1, requester 1b:00, tag 0x03, last 0x0, first 0xf, Addr 0xfee1a000
```

(a) TLPs for transmitting a 98-byte ICMP echo packet.

```
23:51:48.210134 IP libtlp.12290 > adapter.12290: NetTLP: MWr, len 25, requester 1b:00, tag 0x02, last 0x3, first 0xf, Addr 0x2f003000
23:51:48.210139 IP libtlp.12290 > adapter.12290: NetTLP: MWr, len 4, requester 1b:00, tag 0x02, last 0xf, first 0xf, Addr 0x2f002000
23:51:48.210141 IP libtlp.12290 > adapter.12290: NetTLP: MWr, len 1, requester 1b:00, tag 0x02, last 0x0, first 0xf, Addr 0xfee03000
23:51:48.212602 IP adapter.12288 > libtlp.12288: NetTLP: MWr, len 1, requester 00:00, tag 0x00, last 0x0, first 0xf, Addr 0xa0000014
23:51:48.212620 IP libtlp.12288 > adapter.12288: NetTLP: MRd, len 4, requester 1b:00, tag 0x00, last 0xf, first 0xf, Addr 0x2f002000
23:51:48.212633 IP adapter.12288 > libtlp.12288: NetTLP: CplD, len 4, completer 00:00, success, bc 16, requester 1b:00, tag 0x00, lowaddr 0x00
```

(b) TLPs for receiving a 98-byte ICMP reply packet.

Figure 15: tcpdump output of captured TLPs of the simple NIC implementation. `len` indicates length of data payload in DWORD.

simple NIC is a theoretical model of a simplistic Ethernet NIC, which does not have any performance optimizations such as DMA batching, for understanding PCIe interactions and calculating bandwidth. This nonexistent NIC model was a good target for demonstrating the productivity of NetTLP. NetTLP enables us to prototype such models of PCIe devices in software and confirm whether the models actually work with existent hardware root complexes.

The detailed interactions between a host and a simple NIC device are described in the model's implementation [2]. On the TX side, (1) the host updates a 4-byte TX queue tail pointer, (2) the device reads a 16-byte TX descriptor on the host memory, (3) the device reads the packet content and transmits the packet, and (4) the device generates 4-byte interrupt. On the RX side, (1) the host updates a 4-byte RX queue tail pointer, (2) the device reads a 16-byte RX descriptor on the host memory, (3) the device writes a received packet to the host memory, (4) the device generates 4-byte RX interrupt.

Our simple NIC implementation on the NetTLP platform performs an actual NIC with a physical root complex on the adapter host following the model's PCIe interaction. The implementation consists of two parts: a device driver for the NetTLP adapter and a software simple NIC device implementation using LibTLP. The device driver based on the basic driver treats the NetTLP adapter as an Ethernet NIC as well as usual drivers for hardware NICs. The software simple NIC creates a tap interface on the device host and uses the tap interface as its Ethernet port. The Ethernet frames transmitted to the NetTLP adapter are transferred to the device host as TLPs over the PCIe links and Ethernet links, and the Ethernet frames are transmitted to the tap interface. The software simple NIC implementation is about 400 lines of C codes, and it actually performs an Ethernet NIC.

TLPs of the simple NIC generated by the root complex and LibTLP can be observed on the Ethernet link. Figure 15a shows TLPs captured by the modified tcpdump when sending an ICMP echo packet through the simple NIC. The driver writes a TX queue pointer on the BAR4 of the NetTLP adapter (1st TLP), the simple NIC reads the TX descriptor and the packet content on 0x3bb26800 (2nd to 5th TLPs), and the simple NIC generates interrupt to 0xfee1a000 pointed by the MSI-X table after sending the packet to the tap interface (6th TLP). On the RX side shown in Figure 15b, the interaction starts from writing the received ICMP reply packet to the host memory (1st TLP) because the driver told the RX buffer to the simple NIC before receiving new packets. Afterward, the simple NIC updates the RX descriptor (2nd TLP) and generates an interrupt (3rd TLP). After the host consumes the received packet, the driver sends the buffer back to the simple NIC by updating the RX queue tail pointer (4th TLP), and the simple NIC reads the RX descriptor (5th and 6th TLPs). In this manner, the NetTLP enables implementing PCIe devices in software with hardware root complexes. Moreover, the interactions can be observed by the IP networking technique.

6.3 Physical Memory Introspection

The NetTLP provides flexible programmability for TLP interactions between hardware and software. This characteristic offers adaption of NetTLP to other use cases, for example, memory introspection. Methods for monitoring and injecting data on memory have been investigated for both physical [3,12,29,41,46] and virtual [17,49] environments. NetTLP also provides accesses to host memory via PCIe, which is similar to previous studies. However, the NetTLP adapter is a channel to manipulate the host memory remotely; therefore, researchers can implement their introspection and detection methods on top of LibTLP and IP network stack without implementing dedicated hardware or virtual machine monitors.

As the third use case, we demonstrate the possibility of adopting NetTLP into remote memory introspection through two naive applications. The first application is process-list command similar to an example of LibVMI [27]. The process-list collects process information on the Linux host equipped with a NetTLP adapter. Figure 16a shows an example usage of the process-list. When the process-list is executed, it finds

```
$ ./process-list -r 192.168.10.1 -b 1b:00 -s System.map
PhyAddr             PID STAT COMMAND
0x0000000003411740    0 R    swapper/0
0x0000087c330000      1 S    systemd
0x0000086a411dc0    647 S    systemd-journal
0x0000086ba50000    655 S    lvmetad
0x0000086c4cd940    662 S    systemd-udevd
                     ...snipped...
```

(a) process-list to collect process information.

```
$ ./codedump -r 192.168.10.1 -b 1b:00 -s System.map -p 20286 -o code.dump
code area: 0x85a48b000-0x85a48bdb0
dump complete
$ file code.dump
code.dump: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, missing section headers
```

(b) codedump to obtain the code area of a specified process.

Figure 16: Two example applications for physical memory introspection by NetTLP. Both applications are executed on the device host and read the physical memory of the adapter host.

an address of `task_struct` representing the first process from the specified System.map of the adapter host. Next, the process-list starts to walk through `task_struct` structures of the adapter host using `dma_read()`.

Next, let us focus on a single process. codedump obtains a binary of a running process from the adapter host. Figure 16b shows an example usage of this command. The codedump finds `task_struct` for the specified process ID by using the same methods of the process-list and obtains `mm_struct` representing the virtual memory of the process. The codedump then converts process-specific addresses for the code area into corresponding physical addresses by walking through the page table. Lastly, the dumped code area by DMA reads from LibTLP can be treated as a usual binary object file that can even be reassembled by objdump command. In these manners, researchers and developers can easily implement their memory introspection methods on the NetTLP platform.

7 Related Work

Future Interconnect: Some next-generation interconnect specifications are designed by extending the functionality of PCIe. CCIX [13] and CXL [14] introduce cache coherency between processors and peripherals to their interconnects. CCIX uses the PCIe data link layer and defines its transaction layer, and CLX defines CLX extensions on the PCIe data link layer and transaction layer. OpenCAPI [36] and Gen-Z [18] support IEEE 802.3 Ethernet and the PCIe physical layer. These interconnects require hardware extensions for both peripherals and host chipsets. Although such next-generation interconnects introduce new features, they are still packet-based data communications. NetTLP delivers TLPs over Ethernet by exploiting the packet-based communications. Thus, we believe that the NetTLP design can be applied to future interconnects as long as they adopt the layering model and packet-based communications.

Difference between NetTLP and RDMA: As with NetTLP, Remote DMA (RDMA) protocols also achieve DMA from distant hosts over Ethernet and IP networks for high speed interconnect. RoCEv2 encapsulates the Infiniband header and payload with Ethernet, IP, and UDP headers [8]. iWARP uses Ethernet, IP, and TCP headers [38]. In contrast to their purposes, NetTLP aims to provide the observability of PCIe transactions; therefore, it adopts directly encapsulating TLPs in IP and Ethernet. RDMA protocols need to convert the PCIe protocol into RDMA protocol in RDMA adapters. Thus, they lack observability of PCIe protocols that we demonstrated through the use cases.

Device drivers for software PCIe devices: NetTLP has made PCIe prototyping easier, but it has not contributed to the productivity of device drivers. Developing device drivers still requires certain effort. For improving the productivity of device drivers in the NetTLP platform, there are two approaches: the first approach is to use frameworks that automatically generate device drivers from templates related to protocol specifications and device characteristics [42,43]. Another approach is to write device drivers in userspace as with DPDK [20] while using some assists [19].

8 Conclusion

In this paper, we have proposed NetTLP that enables developing software PCIe devices that can interact with hardware root complexes. The key technique to achieve the platform is to separate the PCIe transaction layer into software and then connect the software transaction layer and the hardware data link layer by delivering TLPs over Ethernet and IP networks. Researchers and developers can prototype their own PCIe devices in software and observe actual TLPs by the IP networking techniques such as tcpdump. The use cases in this paper showed the observation of the TLP-level behaviors of the root complex and the product NICs and NVMe SSDs, the 400 LoC software Ethernet NIC implementation interacting with the hardware root complex, and physical memory introspection. We believe that the high productivity and observability on the NetTLP platform demonstrated through the use cases contribute to current and future PCIe development on both research and industrial communities.

Acknowledgments: We would like to thank the NSDI reviewers and our shepherd Andrew Moore for their insightful feedbacks. This work was supported by JSPS KAKENHI Grant Number JP18K18043.

References

[1] NetTLP. https://haeena.dev/nettlp/.

[2] pcie-bench/pcie-model. https://github.com/pcie-bench/pcie-model.

[3] The LeechCore Physical Memory Acquisition Library. https://github.com/ufrisk/LeechCore.

[4] Verilator. https://www.veripool.org/wiki/verilator.

[5] SD Express Cards with PCIe and NVMe Interfaces, 2018.

[6] Intel 82599 10 GbE Controller Datasheet: 3.1.3.2.1 Completion Timeout Period, March, 2016.

[7] PCI Express Base Specification: 2.6.1. Flow Control Rules, November 10, 2010.

[8] Infiniband Trade Association. Infiniband trade association. RoCEv2. https://cw.infinibandta.org/document/dl/7781.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[10] Brad Benton. CCIX, Gen-Z, OpenCAP: Overview & comparison. https://www.openfabrics.org/images/eventpresos/2017presentations/213_CCIXGen-Z_BBenton.pdf, March 2017.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[12] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digit. Investig.*, 1(1):50–60, February 2004.

[13] CCIX Consortium. Cache Coherent Interconnect for Accelerators. https://www.ccixconsortium.com.

[14] CXL Consortium. Compute Express Link. https://www.computeexpresslink.org.

[15] Scott Feldman. Rocker: switchdev prototyping vehicle. http://wiki.netfilter.org/pablo/netdev0.1/papers/Rocker-switchdev-prototyping-vehicle.pdf.

[16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[17] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[18] Gen-Z Consortium. Gen-Z. https://genzconsortium.org/specifications/.

[19] Matthew P. Grosvenor. Uvnic: Rapid prototyping network interface controller device drivers. *SIGCOMM Comput. Commun. Rev.*, 42(4):307–308, August 2012.

[20] Intel. Intel Data Plane Development Kit. http://www.intel.com/go/dpdk.

[21] Intel. PCI Express High Performance Reference Design. https://www.intel.com/content/www/us/en/programmable/documentation/nik1412473924913.html.

[22] Intel. Thunderbolt Technology Community. https://thunderbolttechnology.net.

[23] Jing Qu. GEM5: PciDevice.py. http://repo.gem5.org/gem5/file/tip/src/dev/pci/PciDevice.py.

[24] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[25] Kevin Lee, Vijay Rao, and William Christie Arnold. Accelerating facebook's infrastructure with application-specific hardware. https://code.fb.com/data-center-engineering/accelerating-infrastructure/.

[26] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, Renton, WA, July 2019. USENIX Association.

[27] libvmi. Virtual Machine Introspection. http://libvmi.com/.

[28] Linux. AF_XDP: optimized for high performance packet processing. https://www.kernel.org/doc/html/latest/networking/af_xdp.html.

[29] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[30] Mellanox. BlueField SoC. https://www.mellanox.com/products/bluefield-overview/.

[31] Mentor Graphics. ModelSim. https://www.mentorg.co.jp/products/fpga/verification-simulation/modelsim/.

[32] Micron. User space nvme driver. https://github.com/MicronSSD/unvme.

[33] Netronome. Nfp-4000 flow processor. https://www.netronome.com/m/documents/PB_NFP-4000.pdf.

[34] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 327–341, New York, NY, USA, 2018. ACM.

[35] NVM Express. scalable, efficient, and industry standard. https://nvmexpress.org.

[36] OpenCAPI Consortium. OpenCAPI 4.0 transaction layer specification. https://opencapi.org.

[37] OpenChannelSSD. The LightNVM qemu implementation, based on NVMe. https://github.com/OpenChannelSSD/qemu-nvme.

[38] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040, October 2007.

[39] Luigi Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[40] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[41] Joanna Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition. Proceedings of BlackHat DC, 2007.

[42] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 275–288, New York, NY, USA, 2009. Association for Computing Machinery.

[43] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 73–86, New York, NY, USA, 2009. Association for Computing Machinery.

[44] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[45] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openssd: A pcie-based open source ssd platform. *Proc. Flash Memory Summit*, 2014.

[46] Chad Spensky, Hongyi Hu, and Kevin Leach. Lo-phi: Low-observable physical host instrumentation for malware analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[47] J. Suzuki, Y. Hidaka, J. Higuchi, T. Yoshikawa, and A. Iwata. Expressether - ethernet-based virtualization technology for reconfigurable hardware platform. In *14th IEEE Symposium on High-Performance Interconnects (HOTI'06)*, pages 45–51, Aug 2006.

[48] Jun Suzuki, Teruyuki Baba, Yoichi Hidaka, Junichi Higuchi, Nobuharu Kami, Satoshi Uchida, Masahiko Takahashi, Tomoyoshi Sugawara, and Takashi Yoshikawa. Adaptive memory system over ethernet. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[49] Gary Wang, Zachary J. Estrada, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Hypervisor introspection: A technique for evading passive virtual machine monitoring. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, pages 12–12, Berkeley, CA, USA, 2015. USENIX Association.

[50] P. Willmann, Hyong-youb Kim, S. Rixner, and V. S. Pai. An efficient programmable 10 gigabit ethernet network interface card. In *11th International Symposium on High-Performance Computer Architecture*, pages 96–107, Feb 2005.

[51] Xilinx. Xilinx kintex-7 fpga kc705 evaluation kit. https://japan.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.

[52] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. Netfpga: Rapid prototyping of networking devices in open source. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 363–364, New York, NY, USA, 2015. ACM.