embeddedworld2023
Exhibition&Conference
... it's a smarter world

Composable Edge Cloud Systems With NVMe-over-5G URLLC

Frederik Pfautsch
Missing Link Electronics
Berlin, Germany
frederik.pfautsch@missinglinkelectronics.com

Endric Schubert
Missing Link Electronics
San Jose, CA
Inst. of Microelectronics, Univ. Ulm
endric.schubert@missinglinkelectronics.com

***Abstract*—5G addresses machine-type communication with low energy demands. 5G Edge Clouds reduce processing latency by bringing extra compute close to the "edge". To be cost-efficient, Edge Cloud systems are implemented following modern data center technology, such as NVMe that uses PCIe. PCIe typically is implemented for short-range up to 30 cm. For composability, there is so-called NVMe-over-Fabric, including Long-Range PCIe, which "tunnels" PCIe over network via a distributed switch. PCIe switches are defined in the PCIe specification.**

5G URLLC starts with 3GPP Release 15 meets the requirements for implementing such Long-Range PCIe, enabling the design of composable Edge Cloud Storage over 5G. This paper demonstrates a proof-of-concept "NVMe-over-5G" implementation along with a methodology for detailed latency analysis. We start with key aspects of 5G URLLC, PCIe, and Long-Range PCIe, followed by introducing NVMe and composable Data Center architectures. We then present our setup NVMe-over-5G running on the experimental 5G system at Fraunhofer and TU Berlin. We use COTS hardware comprising AMD/Xilinx Ultrascale+ MPSoC and 5G modems to connect a standard NVMe SSD with a standard CPU running Linux and the NVMe protocol. We break down the overall latency and highlight how much each intermediate component contributes to the total PCIe NVMe end-to-end latency. Our findings show that in general 5G URLLC can meet the PCIe/NVMe requirements, as long as certain optimizations are implemented to reduce tail latency.

***Keywords*—5G, URLLC, PCIe, NVMe, FPGA, edge cloud, TCP/IP, latency**

I. Introduction

5G goes further than its successors and addresses multiple different use cases [1]:

- End-users require bandwidth for video streams
- Sensor networks require low-energy connectivity
- Industrial automation demands reliable and low-latency transmissions

5G has the challenge of fulfilling these heterogeneous requirements and aligning them in one common standard and even simultaneously in the same cell without negatively influencing each other. One of the key ideas of 5G is disaggregation, i.e. removing the need for specialized hardware [2]. With standardized interfaces, the network can be made of different components by different vendors with interchangeable hardware and software. Each component can be containerized and run on commodity servers. Thus, a network can be tailored to specific needs and requirements, even live on demand. Using commodity hardware also simplifies building "campus networks" or a network connecting robots and sensors on factory floors by using open standards and supporting interoperability. By splitting components, 5G User Equipment (UE) can be reduced in size and weight, shifting the intelligence "into the cloud". Instead of, e.g. storing all necessary data on the device itself, data can be stored centrally in the edge cloud. Consider a swarm of drones, e.g. taking measurements or recording data. Instead of having to carry data storage equipment, the data can be streamed to the closely connected edge cloud.

Peripheral Component Interconnect Express (PCIe) is a commonly used, high-speed, point-to-point serial protocol connecting CPUs with expansion cards for high-performance access to storage, e.g. SSDs or accelerators, such as GPUs or Field-Programmable Gate Arrays (FPGAs) [3]. Even though it is mainly known for consumer PCs and professional servers, it is becoming increasingly popular in the embedded world because of its low latency, high throughput, and wide range of support [4]. Each new PCIe generation approximately doubles the available bandwidth, with PCIe Gen 4 providing almost 32 GB/s on 16 lanes. Non-Volatile Memory Express (NVMe) is an example of a modern, fast, PCIe-based communication protocol. Contrary to a software-based, CPU-controlled implementation, NVMe utilizes the PCIe protocol efficiently by issuing multiple read and write requests in parallel. The device reads and writes the data that is stored in several ring buffers per CPU core.

Previous work concentrated on tunneling PCIe over TCP/IP using an FPGA-based setup with multi-gigabit wired connections. A pair of FPGA nodes implement a PCIe switch according to the PCIe specification, thus fully transparent to the PCIe

Root Complex (RC) and PCIe device. TCP provides a reliable transport mechanism with retransmissions in case of errors and congestion control. The Transaction Layer Packets (TLPs) get encapsulated in TCP/IP/Ethernet packets for transportation, independent of the underlying physical medium. This so-called "Long-Range PCIe" with its distributed PCIe switch enables users to connect PCIe devices to CPUs over distances longer than the usual $30\,\mathrm{cm}$.

Ultra-Reliable Low-Latency Communication (URLLC) addresses one of the use cases defined by the 3GPP specification: The demand of reliable communication with low-latency requirements [5]. 5G allows additional, larger subcarrier spacings, shortening the transmission time of a 5G slot containing 14 Orthogonal Frequency-division Multiplexing (OFDM) symbols. URLLC shortens the transmission even further by introducing "mini-slots" which consist of only $2-4$ symbols. Furthermore, URLLC can briefly interrupt ongoing traffic to keep the low-latency guarantees.

We propose an experimental setup tunneling these encapsulated TLPs over a 5G Release 15 network. According to the specification, 5G should provide enough bandwidth to tunnel simple NVMe requests, such as detecting an off-the-shelf SSD and allowing basic read/write operations. In addition, we analyze the end-to-end latency of such requests and evaluate the performance of the whole setup. Due to Release 15 only implementing the basic requirements of URLLC, we could not utilize the low-latency properties of 5G [6]. Our measured end-to-end PCIe latency is $16.1\,\mathrm{ms}$ on average. Due to the tail latency of TCP, we observed PCIe Completion Timeouts (CTOs), especially during enumeration at boot. $99.12\,\%$ of the latency can be attributed to the 5G network. Thus, this specific network cannot meet PCIe demands reliably. However, this proof-of-concept setup can be used to test other networks practically and, e.g. measure the improvement of upcoming 5G features, such as Release 16 and full URLLC support [7].

II. Experimental Setup

Our experimental setup consists of the following parts: On the network side, a standalone 5G Release 15 network, hosted by Fraunhofer HHI as part of the 5G Berlin testbed using a *Nokia AirScale* Remote Radio Head (RRH) and a *Nokia AirScale* 5G Next-Generation Node B (gNB) Baseband Unit (BBU). The BBU is connected to an ng4T 5G core. The network operates at $3.6\,\mathrm{GHz}$ and $100\,\mathrm{MHz}$ bandwidth, with a subcarrier spacing of $30\,\mathrm{kHz}$.

On the user equipment side, we used two *Quectel RM500Q-GL* modems. Each modem connects to a *proFPGA uno ZU19EG* ASIC prototyping platform using USB3. The modems are controlled by the Processing System (PS) running Linux 5.10 with standard drivers. In the Processing Logic (PL), the ZU19EG's PCIe hard macro is either configured as PCIe RC or Endpoint (EP). A commercial off-the-shelf (COTS) mini ATX mainboard with a *Ryzen 3200G* CPU is connected to the FPGA configured as PCIe EP (upstream). A

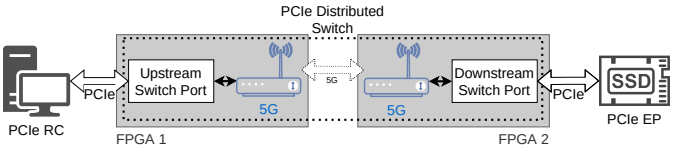

Fig. 1: Distributed PCIe switch with two FPGAs and a PCIe RC and EP

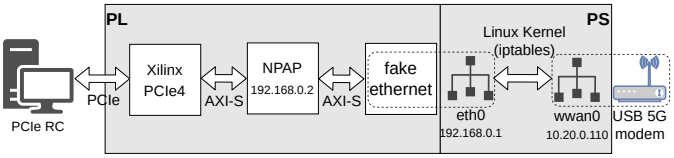

Fig. 2: Block diagram of the Upstream FPGA with 5G interface

COTS NVMe SSD is connected to the FPGA configured as PCIe RC (downstream). Both FPGAs together make up the distributed PCIe switch (Fig. 1).

For a request from the CPU/RC, the upstream PL encapsulates the PCIe TLPs in TCP/IP/Ethernet packets and forwards these to the PS. The PS forwards them to the USB modem to be sent to the corresponding downstream modem over 5G, where the TLPs are decapsulated again.

Our setup is based on previous work [8] by Braun et al. . They encapsulated TLPs in TCP/IP/Ethernet packets and used $10\,\mathrm{Gbit/s}$ QSFP+ cable for transmission. Additionally, they added mechanisms to combine multiple TLPs in one Ethernet packet to coalesce multiple requests. This improved performance by reducing the number of TCP packets that must be acknowledged.

The 5G modem only supports USB3 as an interface, working out-of-the-box with Linux 5.10. Thus, we added additional hardware to the PL, consisting of a Direct Memory Access (DMA) engine, FIFOs for buffering, and Block RAM (BRAM) to simulate Advanced eXtensible Interface (AXI)-mapped registers as a PL/PS crossover interface to forward Ethernet packets between the FPGA and Linux running on the CPU side. On the software side, the *Xilinx axienet* driver for the Ethernet subsystem creates a Linux network interface. Together, the software side with the network interface and the hardware side with its AXI-Stream interface can send Ethernet frames between the PL and PS. We call this "fake ethernet".

The *Quectel RM500Q-GL* is supported by the standard *qmi_wwan* driver shipped with Linux 5.10. The driver creates a wwan network interface. We used the iptables framework to forward data between the wwan-interface and the eth-interface created by the *axienet* driver. Simpler, more resource-efficient bridging was impossible because the wwan network interface is a layer3 point-to-point interface. Thus we used Source Network Address Translation (SNAT) and Destination Network Address Translation (DNAT) to forward specific ports

to the PL and rewrite outgoing packets with the appropriate IP address. Fig. 2 shows a block diagram for the PCIe upstream device.

As reference setup and for comparison to related work, we used wired USB-GBit-Ethernet adapters for replacing the modems. Thus, we create a setup that is also wirebound but matches our 5G setup apart from the modems and the network itself.

III. Methodology

Various tools and techniques were used to measure the latency of all components in our setup. We defined the following criteria for all measurements and describe each measurement in detail afterward.

- An experiment runs over at least $5\,\mathrm{min}$ (mostly 10 – $20\,\mathrm{min}$) while taking at least $10\,000\,\mathrm{individual}$ values
- We also tested using different Ethernet packet sizes to investigate the effect of packet size on latency:
  - Hardware-based: $68\,\mathrm{B}$, $515\,\mathrm{B}$, $1027\,\mathrm{B}$, and $1518\,\mathrm{B}$
  - Software-based: $64\,\mathrm{B}$, $128\,\mathrm{B}$, $256\,\mathrm{B}$, $512\,\mathrm{B}$, $1024\,\mathrm{B}$, and $1518\,\mathrm{B}$
- Line rate limitation of $1\,\mathrm{Mbit/s}$ in hardware and at least $10\,\mathrm{ms}$ between each packet in software to avoid overwhelming the network and thus introducing additional latency
- Outlier filtering by removing values outside the 3σ interval

A. PCIe, end-to-end, CPU to NVMe SSD

Our setup is transparent for the host CPU and the attached NVMe SSD. The FPGAs implement a PCIe switch according to the PCIe specification [3]. Thus the SSD can be used by a standard OS running on the CPU without additional modifications or drivers.

By repeatedly reading the $32\,\mathrm{bit}$ Base Address Register (BAR) address 0, we could measure the SSD's access latency reliably. Our setup consisted of a driver for exposing the access statistics to userspace and a ruby script for initiating the measurement and exporting the access latency.

B. PL to PL, FPGA to FPGA

We replaced the PCIe traffic with randomly generated data for measuring the FPGA-to-FPGA latency. We could measure the latency accurately by using timestamps as "random data" which were returned by the other, receiving, downstream FPGA.

C. PS to PS, embedded OS

For the third end-to-end measurement, we used a C program running in userspace to measure the latency of packets echoed by the other FPGA utilizing `gettimeofday()`. Using ICMP packets with ping yielded similar results. However, due to the potentially different handling of ICMP packets vs. TCP packets and to allow for accurate comparison with the PCIe traffic, we chose to measure with generated TCP traffic.

D. Hardware latencies in PL

Our hardware setup consisted of three main components: The PCIe hard macro, the TCP/IP/Ethernet stack with TLP handling, such as encapsulation and decapsulation, and "fake ethernet" for passing the generated Ethernet frames to the PS side. By using the Xilinx Integrated Logic Analyzer (ILA), we measured the latency of all components. Due to the same magnitude of packet size influence and processing latency in general, we calculated the latency as a range for the smallest and largest packets possible.

E. PL/PS interface

While the PL-side of the "fake ethernet" interface mainly contains only AXI-Stream FIFOs for the DMA, the PS-side, i.e. the *Xilinx axienet* driver, controls what and when data is sent or fetched. Due to the unpredictable nature of the Linux scheduler, this latency varies slightly. We measured this connection in twoways: Once with the userspace C program configured as loopback, thus echoing all received packets back to the sender, i.e. the PL. The second time, we configured the PL as loopback with the software side measuring the latency.

F. Linux iptables

The Linux iptables framework was used for routing the packets between the "fake ethernet" interface (eth0) to the modem interface (wwan0) of the *qmi_wwan* driver and vice versa. Due to the modem only offering a level 3 interface, we could not use a more resource-friendly bridge and had to resort to iptables. This required additional overhead due to IP address rewriting and re-calculating the checksums. We measured the latency using *libpcap* and *tcpdump*'s kernel timestamps. We recorded traffic being forwarded over both interfaces and calculated the timestamps' difference.

G. Linux USB network stack

Linux' *usbmon* facility allows capturing USB packets similarly to network packets using *tcpdump*. However, this method cannot measure the time the modem needs to process and send the packets. While USB technically has a *SUBMIT* and corresponding *COMPLETE* packet, measuring the time between these did not provide any meaningful results. Additionally, it is unclear on the modem's side at which stage of processing the *COMPLETE* packet is sent back to the host and if it is dependent on the 5G network, including the delay of waiting for the appropriate slot for up- or download due to Time-Division Duplex (TDD) [9]. Thus, we could only measure the kernel's processing time of the *qmi_wwan* driver. The remainder of the latency was attributed to the general 5G latency.

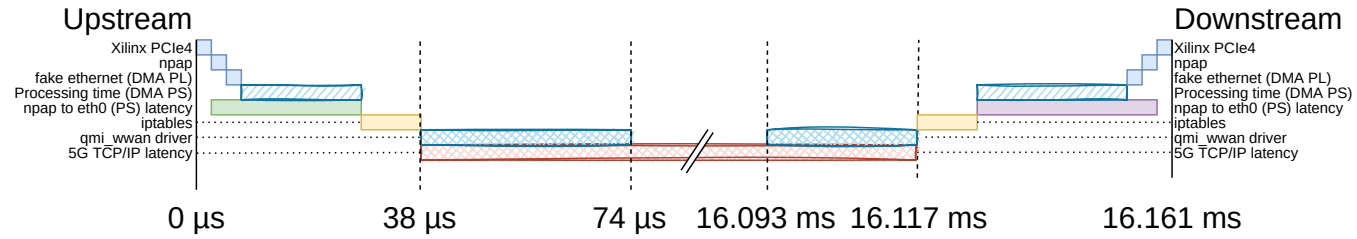

Fig. 3: Latency chain (5G latency not shown fully)

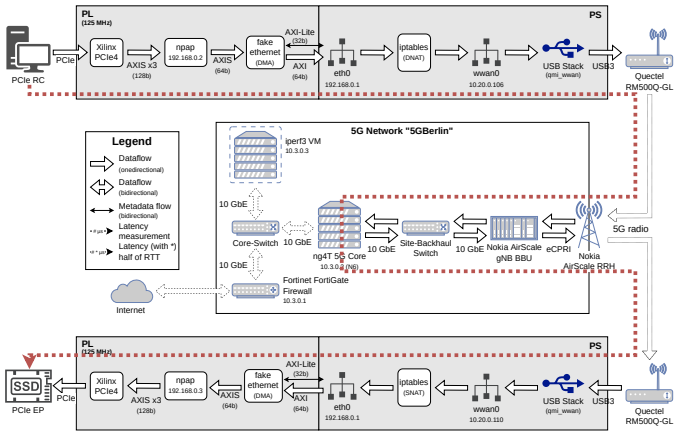

Fig. 4: "The Map" for the PCIe end-to-end latency path using the 5G network

H. Other latencies, e.g. PS to 5G Core

Using a simple ICMP ping and *hping's* TCP "ping", we measured the latency to the 5G core, benchmark machines within the 5G core network and the other board of the distributed PCIe switch to establish a baseline of the expected latency and to verify our measurements.

IV. Results

First, we present the overall latency, measured from PCIe RC to PCIe EP, then describe each intermediate component in detail.

In Fig. 3 you can see the total end-to-end latency of a request sent by the PCIe RC to the PCIe EP. The path is broken up into sections how much each individual component contributes to the total end-to-end latency. First, we can see that the majority of latency ($\approx$99.12 %) is introduced by the 5G network, with the total latency being 16.161 ms ms and the 5G network taking 16.019 ms alone. Some cases, e.g., "*qmi_wwan* driver" and "5G TCP/IP latency", overlap because some measurements were only measurable or analyzable at specific points. While we could measure how long the driver needs to send a received Ethernet packet over USB, we could not accurately measure the 5G latency "behind" the USB interface.

	Upstream (µs)	**Downstream** (µs)
NPAP	0.256 – 3.616	0.256 – 3.616
"fake ethernet"	0.816 – 2.320	0.816 – 2.320
PL → PS → PL *	22.5 – 28.1	22.5 – 28.1
PS → PL → PS *	27.1 – 33.0	27.1 – 33.0
Linux iptables	10.2	11.0
Linux USB Network Stack*	36.2	23.9
ICMP ping to 5G core*	11 700	
ICMP ping to VM*	11 600	
hping to VM*	14 100	

(a) Individual component latencies

	5G (µs)	**GbE** (µs)
PCIe*	16 161	186.3
PL to PL*	20 254 – 32 987	165.7 – 196.4
PS to PS*	14 998 – 22 597	156.0 – 212.6

(b) End-to-end latencies

TABLE I: Measured individual component latencies and end-to-end latencies

Fig. 4 shows the whole setup, including the 5G network architecture with its components. Shown is the end-to-end path of the PCIe request. The upstream FPGA with the connected PCIe RC is depicted at the top, the downstream FPGA with the connected PCIe EP at the bottom. The results of each latency measurement between all of the components as described in Section III are depicted in Table I.

Fig. 5 shows a detailed analysis of the latency distribution of PCIe over 5G, which makes up the mean value of 16.161 ms ms, shown as a dashed vertical line. We can also see that the distribution roughly represents a Gaussian distribution. 3σ-filtering discarded 40 out of the 100 000 measured values over 54 min. The standard deviation equals 3230.04 µs which is also visible as large dispersion of the data. Accessing the *ADATA XPG SX8000 128GB* SSD using basic operations such as creating test files that were written to or read from was possible, albeit considerably slower. Due to the high latency and the small TLP sizes, the theoretical 5G bandwidth of about 50 – 70 Mbit/s could not be fully utilized and practically maxed out at about 13 Mbit/s when writing a large, randomly generated 1 GiB file.

Fig. 6 shows the latency distribution for dummy-traffic instead of PCIe from PL to PL. The network path is shown

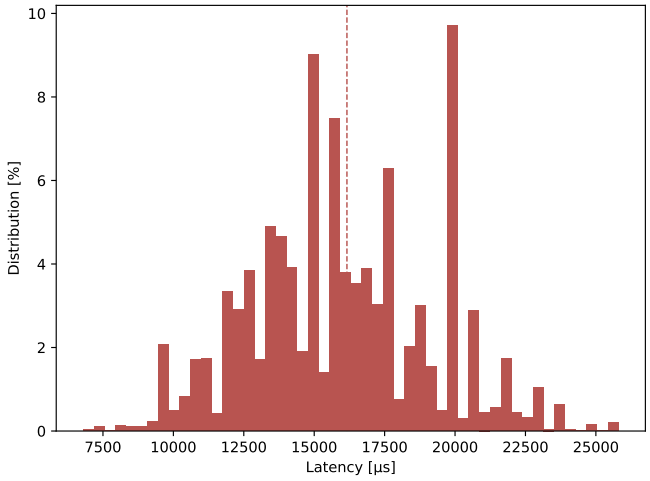

Fig. 5: 5G PCIe latency distribution reading the 32 bit BAR address 0 with the dashed vertical line showing the mean value

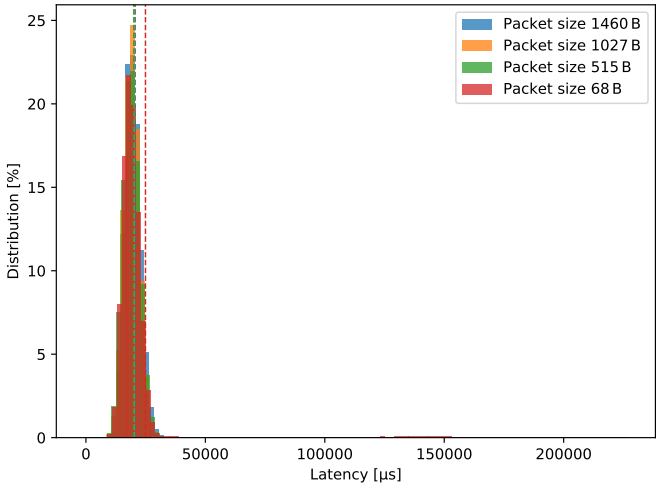

Fig. 6: 5G PL-to-PL end-to-end latency distribution, including tail latency

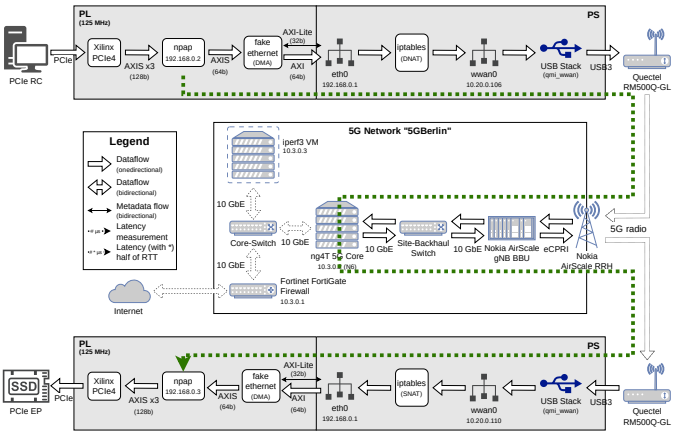

Fig. 7: "The Map" for the PL-to-PL end-to-end latency path using the 5G network

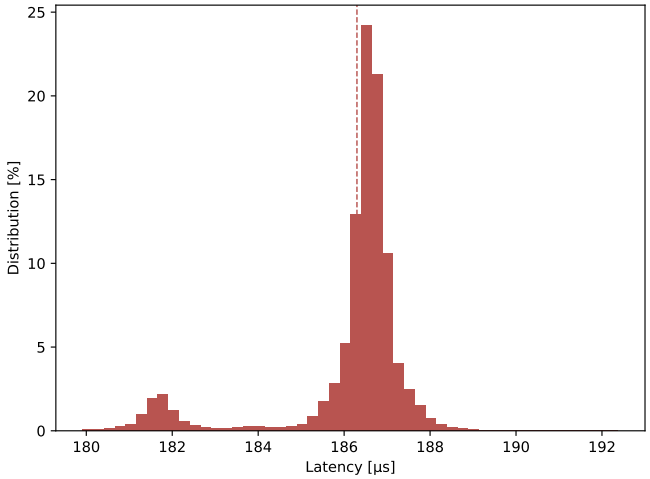

Fig. 8: 1 GbE PCIe latency distribution reading the 32 bit BAR address 0 with the dashed vertical line showing the mean value

in Fig. 7. While the majority of packets have a latency of on average 20.2 ms one-way across all packet sizes, some individual packets (less than 0.01 %) have a tail latency of more than 200 ms one-way. This affects mostly packets of smaller sizes, such as 68 B.

Fig. 8 shows the same experiment, but with the *EDIMAX EU-4306* USB Ethernet adapters replacing the *Quectel RM500Q-GL*. It shows two almost distinct distributions, with minimal, individual standard deviations. The distribution on the right with a mode of 186.5 µs represents most values. The mean latency is 186.30 µs, shown as a dashed lane. On the left, the smaller distribution shows approximately 10 % of the values with latencies ≈5 µs smaller, probably due to caching or frequency and scheduling changes. While the CPU accessing the SSD and running the benchmarking tool was fixed to the full 3.6 GHz with the performance scheduler and disabled c-states, the CPUs of the PS running *PetaLinux* can also influence the measurements because of interrupt scheduling and caching.

Finally, we can summarize three main results:

1) Due to the high latency and the significant variance, the setup does not boot using the 5G network and requires a wired connection. While PCIe is technically able to handle PCIe CTOs with Round-Trip Times (RTTs) as high as 4 – 64 s in the highest range, the default range of devices being 50 – 50 000 µs. With an average latency of 16.161 ms one-way and a standard deviation of 3.596 ms, we presume that it leads to timeouts with the system being stuck in an unrecoverable PCIe state. Our test with dummy-PCIe traffic has also shown a very high tail latency.

2) The 5G Release 15 network introduces the majority of latency ($>99\,\%$) in our setup compared to our reference setup. The use of COTS hardware required additional workarounds, i.e. "fake ethernet" and Linux iptables, which are responsible for an additional $78\,\mu s$ of latency.
3) Measuring the PCIe latency using the reference setup results in a mean end-to-end latency of $186.30\,\mu s$. This value, especially keeping the additional overhead of $138.1\,\mu s$ for "fake ethernet", Linux iptables, and the USB stack in mind, is similar to Schubert's measured latency of $52.5\,\mu s$ for a comparable NVMe SSD. This shows that our reference setup is comparable to Schubert's setup.

V. Conclusion

We extended an existing distributed PCIe switch using a TCP/IP connection over $10\,\mathrm{Gbit/s}$-Ethernet with a 5G-based "tunnel". The usage of COTS components, such as a USB 5G modem, required us to implement internal routing mechanisms to transfer the Ethernet frames generated by the PL of the FPGA to the USB 5G modem attached to the PS/CPU of the System-on-Chip (SoC), using the standard Linux driver. We analyzed the resulting PCIe end-to-end latency but also the individual latencies of all components in between, making up the end-to-end latency. Due to the high latency of the 5G network with $16.161\,\mathrm{ms}$ and the high variance, the BIOS cannot enumerate the remotely attached PCIe device and presumably encounters PCIe CTOs. Booting over a wired connection is possible, and Linux can successfully re-enumerate the PCIe tree when switched back to the 5G connection.

Our measurements have shown that the 5G network causes $>99\,\%$ of the end-to-end latency. An additional $\approx 0.23\,\%$ are artificially introduced by being limited to COTS components and having to implement workarounds to route generated TCP/IP packets to the USB 5G modem.

In summary, our proof-of-concept setup is valid for latency measurements. Our reference setup measures similar latencies for a $1\,\mathrm{Gbit/s}$-Ethernet wired connection, comparable to previous work by [8], which differs only in the use of USB Ethernet adapters instead of USB 5G modems. We have shown that it is possible to "tunnel" PCIe over 5G, although not entirely reliable due to the TCP tail latency. While the latency and bandwidth pose a significant bottleneck and potential deal breaker due to PCIe CTOs, improvements such as URLLC with the upcoming next releases of 5G should alleviate these issues.

Acknowledgment

We would like to thank Ronald Freund, Fraunhofer HHI for providing us access to the 5G campus network. Additionally, we are also grateful to Kai Habel for his support and knowledge of whom to talk to regarding the 5G network. We would also like to extend our thanks to Konstantin Koslowski for patiently answering all our questions regarding the 5G network and providing us with valuable insights.

References

[1] Nokia Corporation, "5G New Radio Network. Use Cases, Spectrum, Technologies and Architecture," White Paper, 2021. [Online]. Available: https://onestore.nokia.com/asset/205407

[2] S. Sun, M. Kadoch, L. Gong, and B. Rong, "Integrating network function virtualization with SDR and SDN for 4g/5g networks," *IEEE Network*, vol. 29, no. 3, pp. 54–59, 05 2015.

[3] PCI-SIG, *PCI Express Base Specification Revision 6.0, Version 1.0*, 01 2022.

[4] M. Jones. (2009, Feb.) PCIe catches up in embedded system design. [Online]. Available: https://www.embedded.com/pcie-catches-up-in-embedded-system-design/

[5] 3GPP, "Ts 23.501; technical specification group services and system aspects; system architecture for the 5g system (5gs)," Tech. Rep. Release 15.13.0, 03 2022. [Online]. Available: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3144

[6] 3GPP, "Release 15 Description; Summary of Rel-15 Work Items," Tech. Rep. Release 15, 10 2019. [Online]. Available: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3389

[7] I. Rahman, S. M. Razavi, O. Liberg, C. Hoymann, H. Wiemann, C. Tidestav, P. Schliwa-Bertling, P. Persson, and D. Gerstenberger, "5g evolution toward 5g advanced: An overview of 3gpp releases 17 and 18," Ericsson, techreport, Oct. 2021.

[8] E. Schubert, A. Braun, and U. Langenbach, "PCI Express over IP - Accelerated," MLE. Embedded World Conference, 2016.

[9] A. A. Esswie and K. I. Pedersen, "On the ultra-reliable and low-latency communications in flexible TDD/FDD 5g networks," in *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 01 2020.