Making QUIC Quicker with NIC Offload

Xiangrui Yang, Lars Eggerts, Jörg Ott, Steve Uhlig, Zhigang Sun, Gianni Antichi

NUDT, NetApp, TUM, QMUL

SmartNIC to accelerate transport protocols

A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS

Yatin Hoskote, *Member, IEEE*, Bradley A. Bloech
Vasantha Erraguntla, *Member, IEEE*, David Finan, Jason I
Greg Ruhl, James W. Tschanz, *Member, IEEE*, Sriram Vang
Howard Wilson, Jianping Xu, *M*

***Abstract*—This programmable engine is designed to offload TCP inbound processing at wire speed for 10-Gb/s Ethernet, supporting 64-byte minimum packet size. This prototype chip employs a high-speed core and a specialized instruction set. It includes hardware support for dynamically reordering out-of-order packets. In a 90-nm CMOS process, the 8-mm^2 experimental chip has 460 K transistors. First silicon has been validated to be fully functional and achieves 9.64-Gb/s packet processing performance at 1.72 V and consumes 6.39 W.**

***Index Terms*—Gigabit Ethernet, offload, packet processing, spe-**

10Gbps Implementation of TLS/SSL Accelerator on FPGA

Takashi Isobe, Satoshi Tsutsumi
Central Research Laboratory
Hitachi, Ltd., Japan
{takashi.isobe.fm, satoshi.tsutsumi.ba}
@hitachi.com

Koichiro Seto, Kenji Aoshima, Kazutoshi Kariya
Tsukuba Network Technical Center
Hitachi Cable, Ltd., Japan
{seto.koichiro, aoshima.kenji, kariya.kazutoshi}
@hitachi-cable.co.jp

***Abstract*—This paper proposes the one-chip architecture to mount all processes for TLS/SSL ciphered communication into one FPGA or ASIC, and shows the 10 Gbps implementation of low-power (23 W) TLS/SSL accelerator on 65 nm FPGA. The usage of FPGA/ASIC enables high efficient processing and low-power consumption by using parallel, optimized and pipelined processing. One-chip architecture achieves high throughput by using a switch to avoid the congestion in exchanging data between multiple processing-blocks. In this research, to reduce the circuit area in the one-chip architecture, high-efficient processing design (a parallel processing circuit shared with**

TLS/SSL accelerator, aiming at twice or more times larger throughput (10 Gbps) than the add-on server and one-tenth power consumption (27 W) of that.

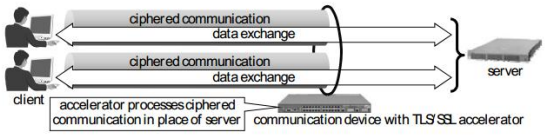


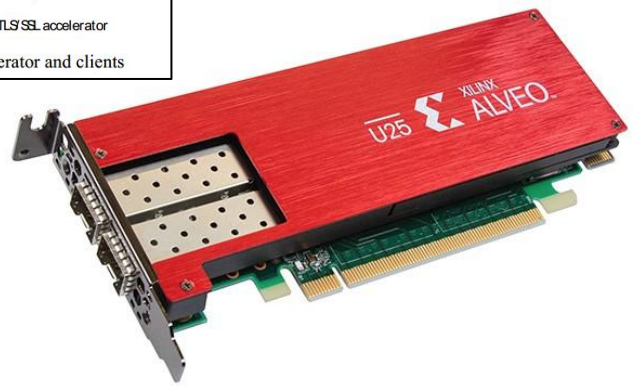
Fig. 1: Ciphered communication between a accelerator and clients

Accelerating QUIC via Hardware Offloads through a Socket

Wochtman, Joanna Muniak, Manasi Deval

dware Capabilities

ware used to enable these offloads consists of
® Ethernet Controller XL710 40Gbe backplane
connected to an Intel® Arria® 10 FPGA, which
e link speed to 25 Gbps. The Arria 10 acts as
n-the-wire QUIC processing agent. Consequently,
o separate control plane interface to configure the



Enabling Programmable Transport Protocols in

Mina Tahmasbi Arashloo
Princeton University

Alexey Lavrov
Princeton University

Manya Ghoba
MIT

David Walker
Princeton University

David Wentzlaff
Princeton University

Muhammad Asim Jamshed
Intel Labs

KyoungSoo Park
KAIST

Abstract

Data-center network stacks are moving into hardware to achieve 100 Gbps data rates and beyond at low latency and low CPU utilization. However, hardwiring the network stack in the NIC would stifle innovation in transport protocols. In

to the NIC to either be used directly through the socket API (TCP Offload Engine [10]) or to enable RDMA (iWARP [7]).

These protocols, however, only use a small fixed set out of the myriad of possible algorithms for reliable delivery [16, 21, 24, 27, 33, 34] and congestion control [12, 17, 19,

ous optimizations such as kernel-bypassing and zero-

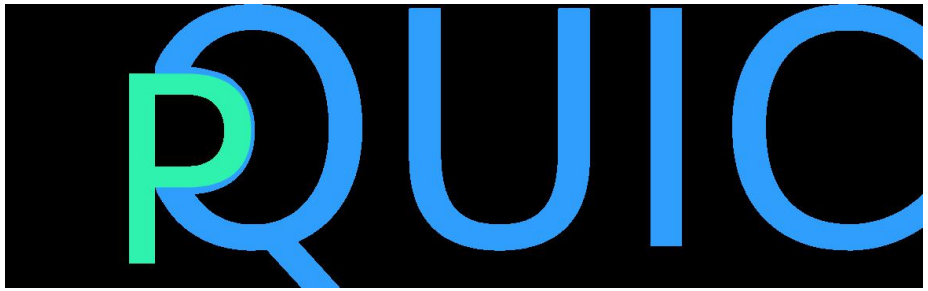
Ensuring the desirable properties of TCP, however, often entails a severe performance penalty. This is especially pronounced with the recent trend that the gap between CPU capacity and network bandwidth widens. Two notable scenarios where modern TCP servers suffer

And the trend in QUIC…

- Understandardization at IETF(v29 so far);
- Used by 4.6% of all websites (9.1% of overall traffic 2019) and growing;
- Google has pushed 42.1% of its traffic via QUIC.

Yet its also a complex thus resource burning protocol.

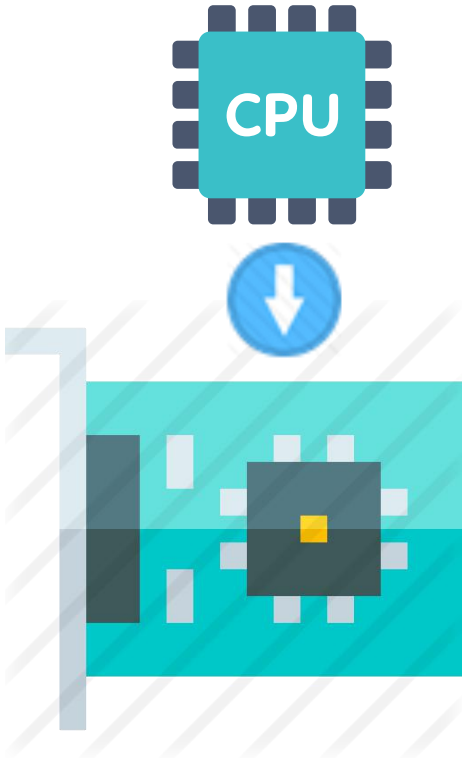
According to Google[1], QUIC burns 3.5 times more CPU cycles than TCP&TLS.



Rüth, Jan, et al. "A First Look at QUIC in the Wild." International Conference on Passive and Active Network Measurement. Springer, 2018.
Langley, Adam, et al. "The quic transport protocol: Design and internet-scale deployment." Proceedings of SIGCOMM. 2017.

The question in the context of QUIC is:

Goal: What are the **primitives in QUIC** that should be offloaded onto **SmartNICs**?

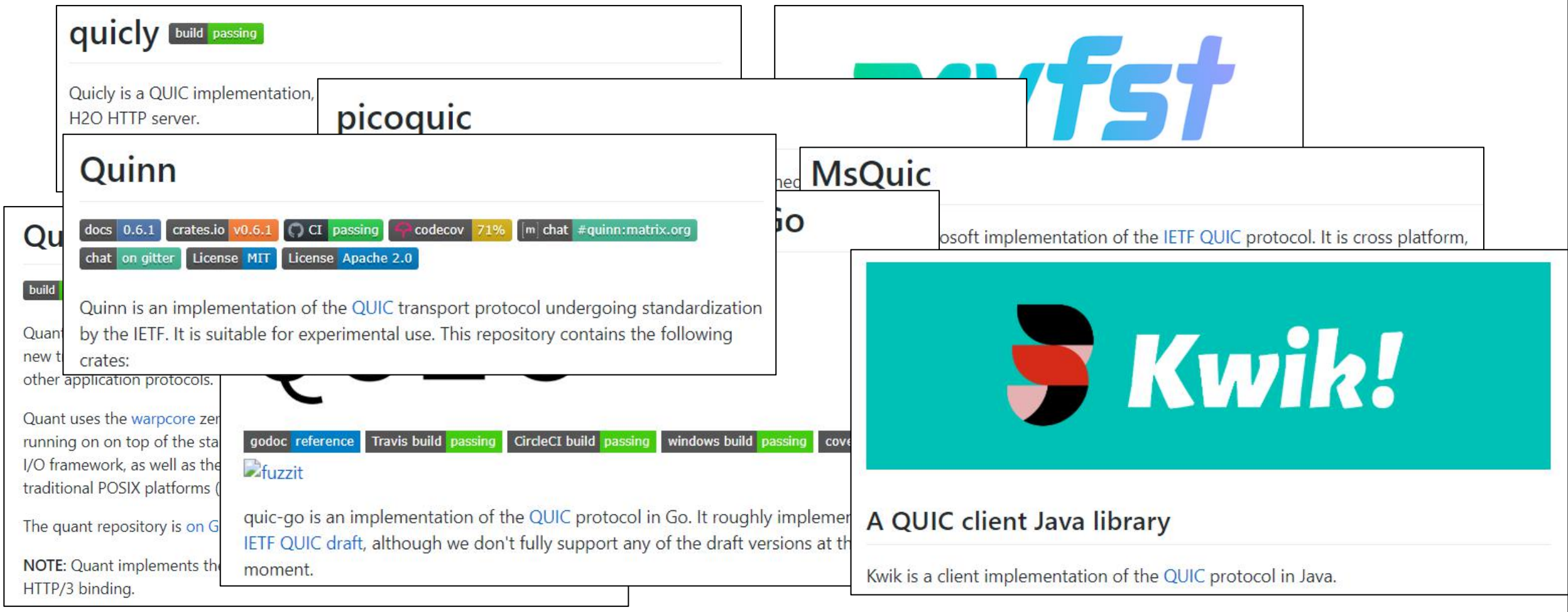


Test!

Measurement!

There are so many different QUIC impls!

- QUIC is envolving really **FAST**, **29** versions within **3** years, over **20** impls!



How do we choose among them?

Principle:

- **Comply with the lestast draft version**? **Yes**!
- **Opensource**? **Yes**, we might need to add instrumentations.
- **Same programming language while efficient**? **Yes**!

And its also good to compare different I/O engines! (socket, kernel-bypsss...)

proj	version	language	I/O engine	Repo address	Server & Client
mvfst	27	**C++**	posix socket	**https://github.com/facebookincubator/mvfst**	S & C
quant	27	**C**	netmap	**https://github.com/NTAP/quant**	S & C
quicly	27	**C**	posix socket	**https://github.com/h2o/quicly**	S & C
picoquic	27	**C**	posix socket	**https://github.com/private-octopus/picoquic**	S & C

Next is the testbed…

- Server and client are pinned to 2 sperate cores and isolated using different network namespace;
- TLEM is used to simulate different traffic scenarios (loss, delay, re-order); **better performance!**
- NIC-offload features are disabled to avoid potential interferences.

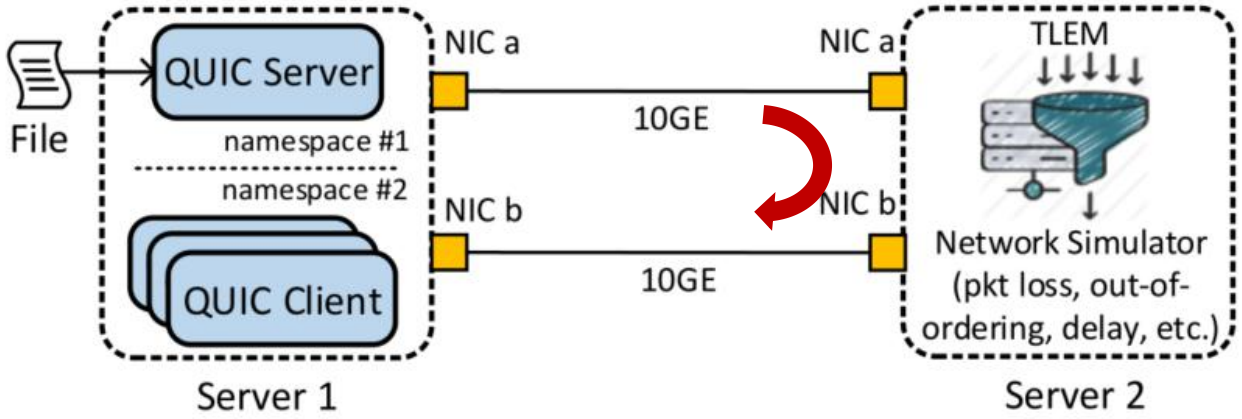


Figure 1: Testbed.

CPU	Intel Xeon Silver 4114 CPU, 2.2GHz
RAM	64GB
NIC driver	ixgbe (offload features are disabled)
OS	Ubuntu 18.10, Linux 4.18.0-25-generic
Emulator	TLEM

Rizzo, Luigi, Giuseppe Lettieri, and Vincenzo Maffione. "Very high speed link emulation with TLEM." 2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN). IEEE, 2016.

Lesson 1: I/O Engines matter A LOT

- Start with one connection:

Table 2: Maximum throughput vs CPU usage.

	Quant	Quicly	Picoquic	Mvfst
throughput	4121Mbps	463Mbps	489Mbps	325Mbps
Server	90.1%	54.8%	60.4%	47.2%
Client	88.2%	52.3%	49.9%	46.4%

with **netmap**, the overall throuhput grows **10x** higher compared to other QUIC impls

around **50%** for QUIC impls using posix socket

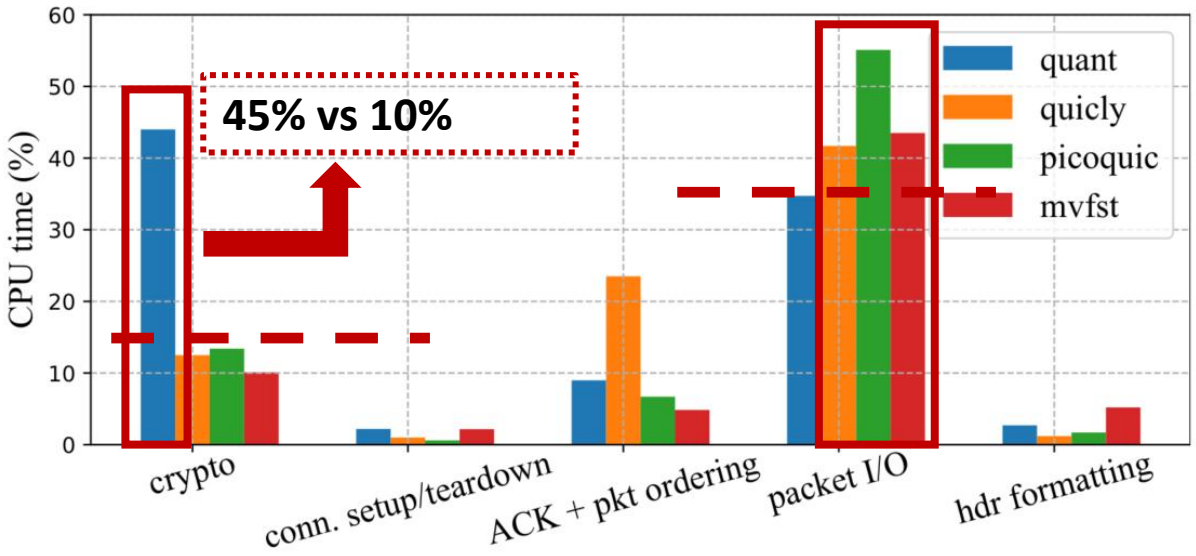
with **netmap**, the core utilization of both server and client gets around **90%**

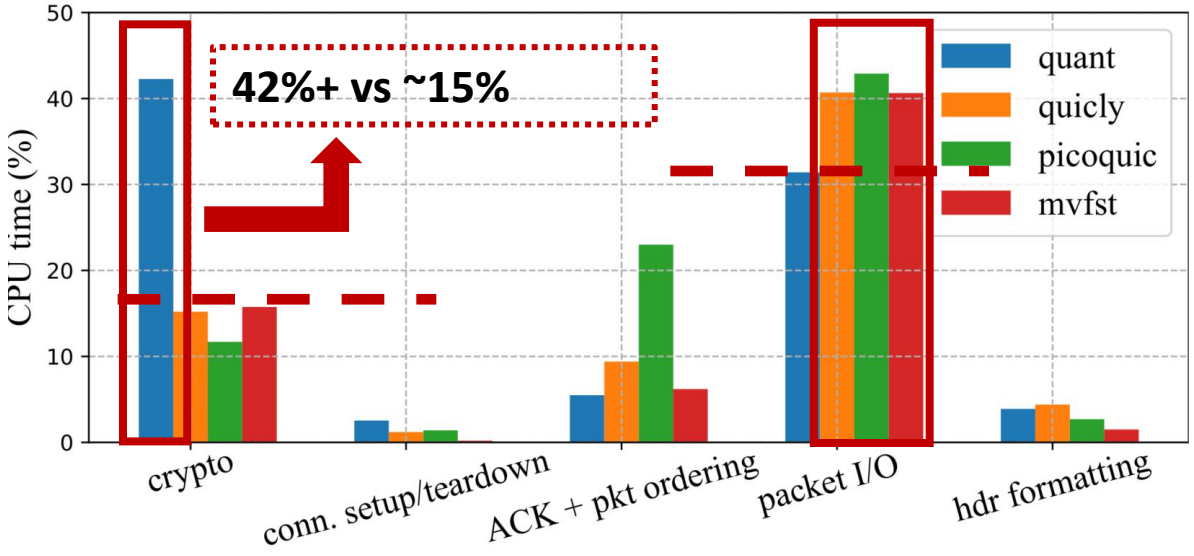
Then the question is:

what are the bottlenecks in different QUIC impl?

Lesson 2: Crypto engines cost 40%+ CPU cycles

Then we breakdown the CPU utilization of both server & client

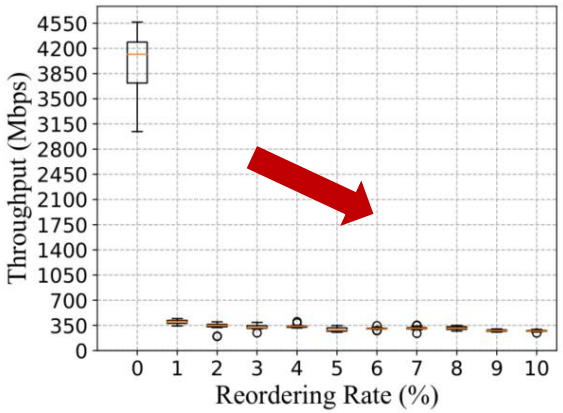




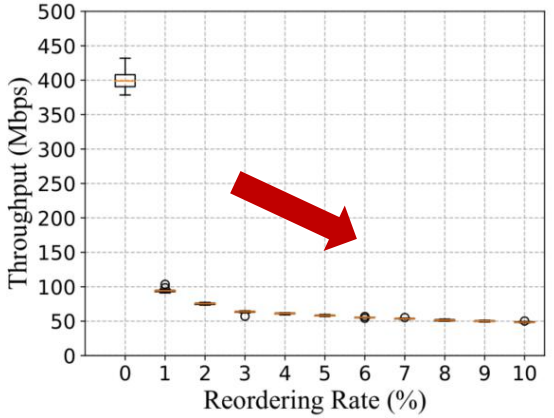
In quant, the performance bottleneck (45%+) is the crypto func used for AEAD operations.
While in the other 3 impls, the bottleneck (~45%) is the data copies between user/kernel.

Lesson 3: Packet reordering harms performance

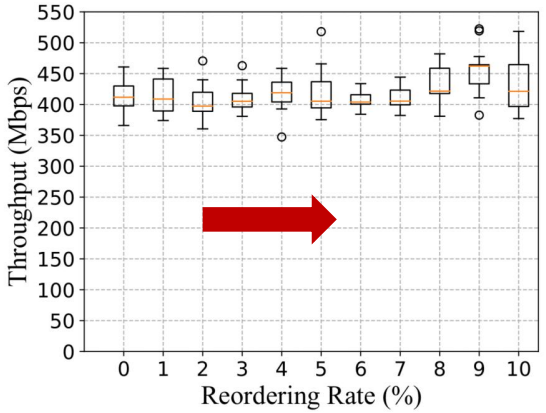
Then, we introduce different levels or traffic interference on the link with TLEM



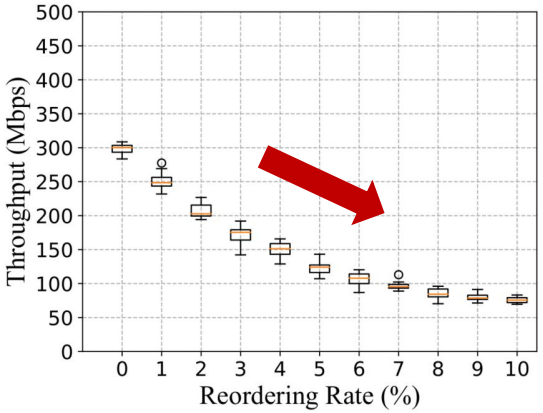
quant



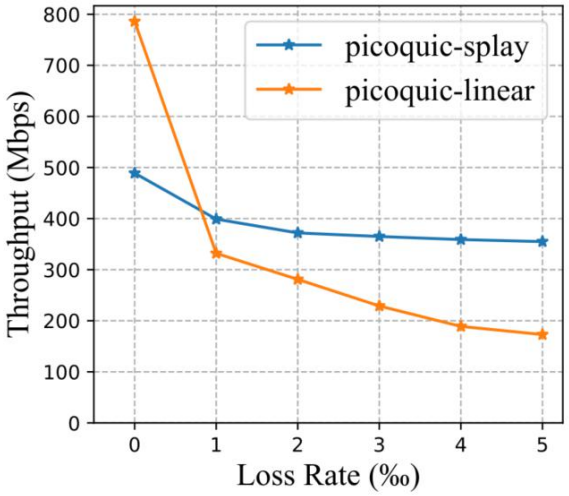
quicly

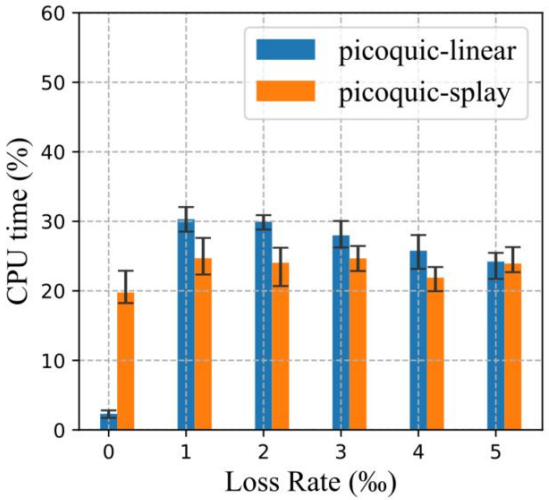


picoquic



mvfst



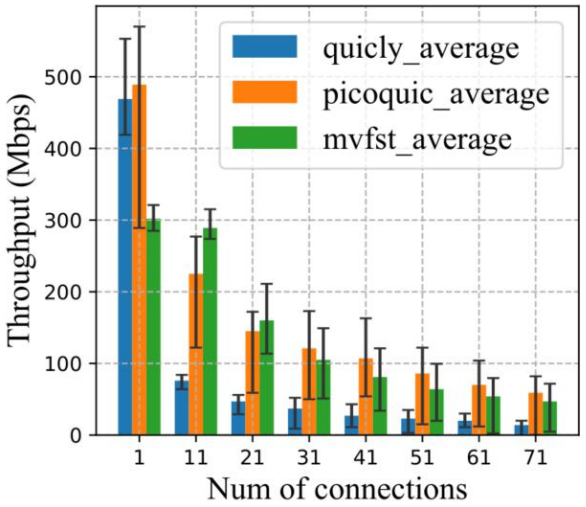


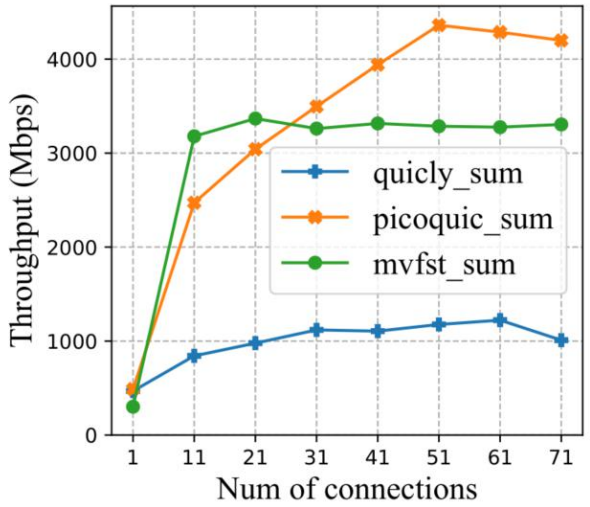
an example:
picoquic(linear) vs picoquic(splay tree)

An unefficient reorder algorithm could be a potential performance bottleneck

Inspired by: https://github.com/private-octopus/picoquic/issues/741#issuecomment-665062732

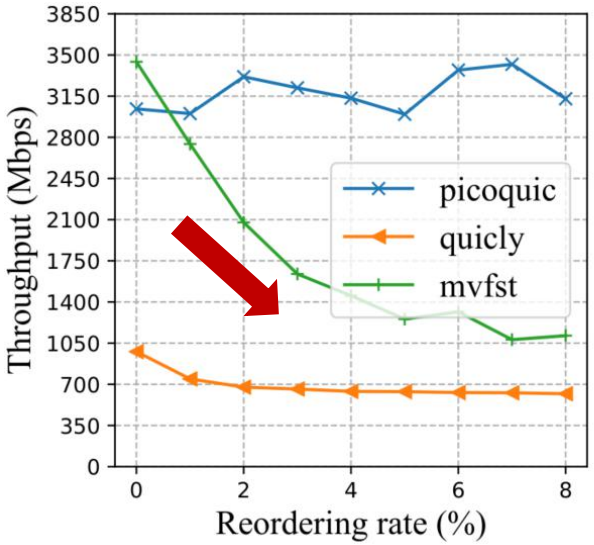
Other findings (multi-connection)...

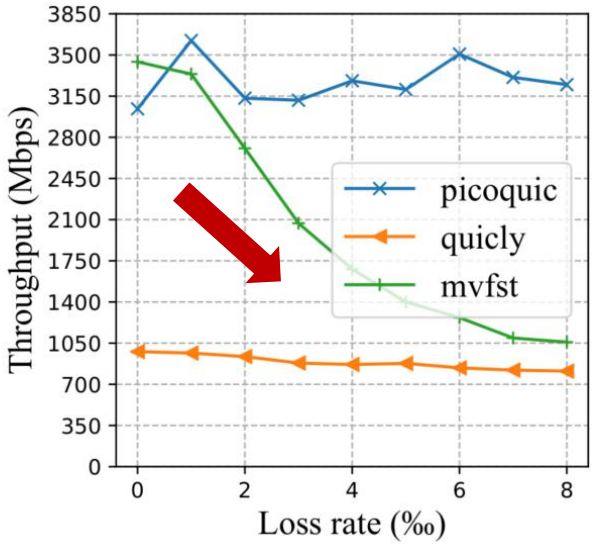




- Picoquic and Mvfst outperform Quicly of about 4x when the connections exceeds 40.
- High throughput without kernel-bypass but instead relying on multiple connections.
- CPU cost of each connection doesn't change much.

- 21 connections simultaneously;
- similar to single conn scenario, packet ooo has negetive effect on throughput (quicly & mvfst)
- Throughput of mvfst is heavily influenced by both packet out-of-order and packet loss, could be a potential bug.





A recap to the measurement we did

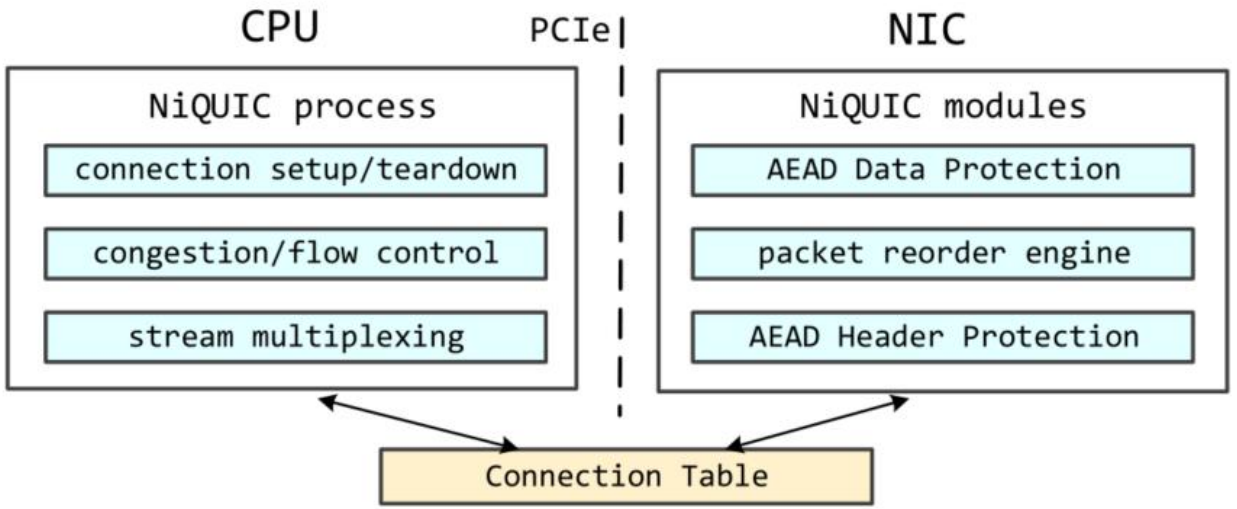
- **Lesson #1**: **Data copy** between user/kernel space costs around **50%** CPU usage, can be avoided efficiently by kernel bypass techniques.

- **Lesson #2**: With kernel-bypass, **crypto operations** become the main performance bottleneck, costing **40%+** overall cycles.

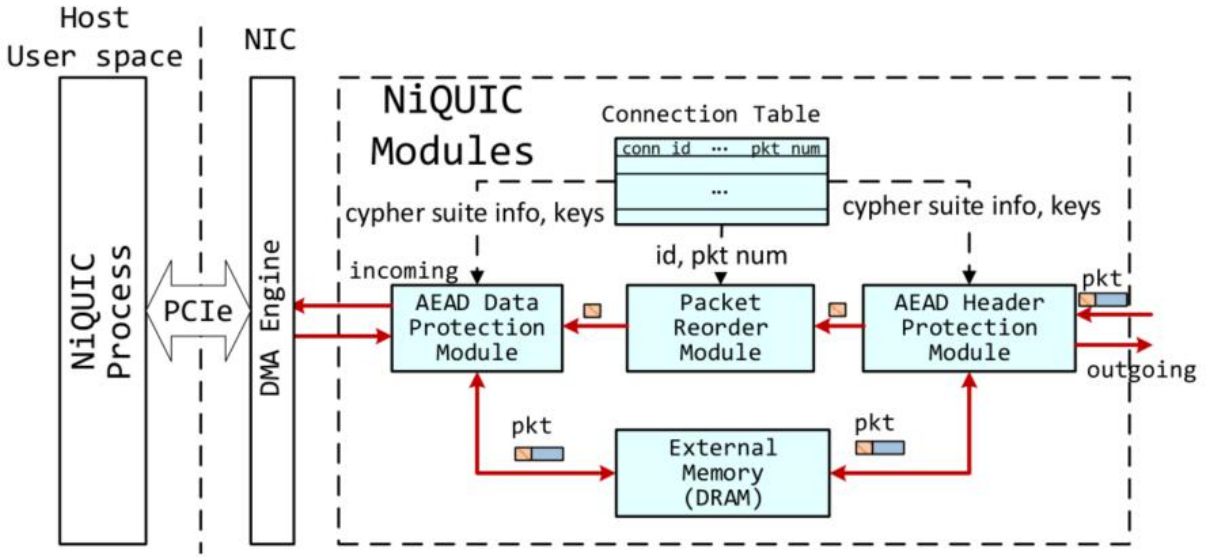
- **Lesson #3**: The way dealing with **packet out-of-order** matters a lot to the performance when the network is in such scenarios.

So, how do we offload QUIC efficiently?

- **Guidelines**:
  1. Provide NIC-support for AEAD operations;
  2. Move packet reordering to the NIC;
  3. Keep control operations in the host CPU.



- **High-level Design**:
  - HW: AEAD engine, reorder engine
  - SW: control plane operations



CPU <-----> conn table <-----> NIC

Potential challenges?

- **Hardware/Software Synchronization**
  - a general connection table could be of great help
  - overhead of table entry updating? (AEAD keys, etc.)
  - Algorithms of determine which conn shall be offloaded?
- **Low frequency for most AEAD IP core**
  - the possibility of parallelize multi modules?
  - timing issue & resource usage on FPGA?
- **Packet reordering on FPGA**
  - HBM on Xilinx board (AU280) could be useful
  - TCAM is a perfect tool for reordering on the hardware
  - How to distinguish packet ooo from packet loss (timer shall be needed)

Limitations and ongoing work

- Didn't consider the influence of the offloading features that current NIC provides (GSO, packet pacing, etc);
- Didn't investigate some commercial QUIC implementations like msquic from Microsoft, quiche from Netflix and so on.

But we've started to patch that!

opensource @ `https://github.com/Winters123/QUIC-measurement-kit`

Questions?