QUIC on the Highway: Evaluating Performance on High-rate Links

Benedikt Jaeger, Johannes Zirngibl, Marcel Kempf, Kevin Ploch, Georg Carle
Technical University of Munich, Munich, Germany
{jaeger, zirngibl, kempfm, plochk, carle}@net.in.tum.de

***Abstract*—QUIC is a new protocol standardized in 2021 designed to improve on the widely used TCP / TLS stack. The main goal is to speed up web traffic via HTTP, but it is also used in other areas like tunneling. Based on UDP it offers features like reliable in-order delivery, flow and congestion control, stream-based multiplexing, and always-on encryption using TLS 1.3. Other than with TCP, QUIC implements all these features in user space, only requiring kernel interaction for UDP. While running in user space provides more flexibility, it profits less from efficiency and optimization within the kernel. Multiple implementations exist, differing in programming language, architecture, and design choices.**

This paper presents an extension to the QUIC Interop Runner, a framework for testing interoperability of QUIC implementations. Our contribution enables reproducible QUIC benchmarks on dedicated hardware. We provide baseline results on 10G links, including multiple implementations, evaluate how OS features like buffer sizes and NIC offloading impact QUIC performance, and show which data rates can be achieved with QUIC compared to TCP. Our results show that QUIC performance varies widely between client and server implementations from 90 Mbit/s to 4900 Mbit/s. We show that the OS generally sets the default buffer size too small, which should be increased by at least an order of magnitude based on our findings. Furthermore, QUIC benefits less from NIC offloading and AES NI hardware acceleration while both features improve the goodput of TCP to around 8000 Mbit/s. Our framework can be applied to evaluate the effects of future improvements to the protocol or the OS.

***Index Terms*—QUIC, High-rate links, Performance evaluation, Transport protocols**

I. Introduction

QUIC is a general-purpose protocol that combines transport layer functionality, encryption through TLS 1.3, and features from the application layer. Proposed and initially deployed by Google [1, 2], it was finally standardized by the Internet Engineering Task Force (IETF) in 2021 [3] after more than five years of discussion. Like TCP, it is connection-oriented, reliable, and provides flow and congestion control in its initial design. An extension for unreliable data transmission was added in RFC9221 [4].

One goal of QUIC is to improve web communication with HTTPS, which is currently using TCP / TLS as underlying protocols. It achieves this by accelerating connection build-up with faster handshakes, allowing only ciphers considered secure, and fixing the head-of-line blocking problem with HTTP/2. The transport layer handshake is directly combined with a TLS handshake allowing 0- and 1-RTT connection establishment. To comply with currently deployed network devices and mechanisms, QUIC relies on the established and widely supported transport protocol UDP. The usage of UDP allows to implement QUIC libraries in user space. Thus, QUIC can initially be deployed without requiring new infrastructure, and libraries can be easily implemented, updated, and shipped.

On the one hand, this resulted in a variety of implementations based on different programming languages and paradigms [5]. On the other hand, previous efforts to optimize the performance of existing protocols have to be applied to new libraries, kernel optimizations cannot be used to the same degree, and the negative impact of encryption on performance has to be considered. Additionally, the heterogeneity among the QUIC implementations requires a consistent measurement environment for a reproducible evaluation. The performance differences between QUIC and TCP / TLS become more evident when the implementations are pushed to their limits on high-speed networks.

In this work, we show the impact of these effects through a fine-grained analysis of different QUIC implementations. We extend the existing QUIC Interop Runner (QIR), a framework for interoperability testing of QUIC implementations [6, 7]. Using Docker containers, it primarily focuses on functional correctness. Thus, we extend it to allow for performance-oriented measurements on bare metal.

Our contributions in this paper are:

(i) We develop and publish a **measurement framework based on the existing QIR framework**. The measurements are run on real hardware, and more metrics are collected allowing an in-depth analysis of performance bottlenecks. Besides our code, all configurations and the collected data are published along with this paper. With this, we want to foster reproducibility and allow other researchers and library maintainers to evaluate different QUIC libraries and test potential performance optimizations.

(ii) We perform a **baseline performance evaluation of different QUIC implementations** on a 10G link with the proposed framework. Tested implementations show a wide diversity in their configuration and behavior.

(iii) We **evaluate impacting factors on the performance** of QUIC-based data transmissions. We analyze the effect of, *e.g.*, different buffer sizes, cryptography, and offloading technologies. This shows how goodput can be increased beyond the default setting.

ISBN 978-3-903176-57-7

We explain relevant background regarding QUIC in Section II. In Section III, we introduce the implemented measurement framework and used configurations. We present our findings and evaluations in Section IV. Section V contains an outline of related work. Finally, we discuss our findings and conclude in Section VI.

II. Background

This section introduces relevant background for various properties impacting QUIC performance, as shown in Section IV. We identify components relevant to the overall performance and point out the main differences compared to TCP / TLS. They include the always-on encryption in QUIC, the different acknowledgment (ACK) handling, the involved buffers in the network stack, and offloading functionalities supported by the Network Interface Card (NIC).

A. Encryption

QUIC relies on TLS version 1.3, which reduces available cipher suites to only four compared to previous TLS versions. Only ciphers supporting authenticated encryption with additional data (AEAD) are allowed by the RFC [8]. AEAD encrypts the QUIC packet payload while authenticating both the payload and the unencrypted header. Supported ciphers either rely on Advanced Encryption Standard (AES), a block cipher with hardware acceleration on most modern CPUs, or ChaCha20, a stream cipher that is more efficient than AES without hardware acceleration[1].

Considering TLS in combination with TCP, data is encrypted into so-called TLS records, which are, in general, larger than individual TCP segments spanning over multiple packets. TCP handles packetization and the reliable, in-order transfer of the record before it is reassembled and decrypted at the receiver. With QUIC, each packet must be sent and encrypted individually. On receiving a packet, it is first decrypted before data streams can be reordered. Loss detection and retransmissions are done on packet- and not stream-level using the packet number similar to TCP's sequence number.

Additionally, QUIC adds another layer of protection to the header data, called header protection. Fields of the QUIC header like the packet number are encrypted again after packet protection has been applied, leading to twice as many encryption and decryption operations per packet.

B. Acknowledgments

Since QUIC provides reliability and stream orientation, it requires a similar ACK process as TCP. QUIC encapsulates QUIC packets into UDP datagrams, while the packets carry the actual payload as QUIC frames. Different frame types exist, such as *stream*, *ACK*, *padding*, or *crypto*.

An *ACK* frame contains so-called ACK ranges, acknowledging multiple packets and ranges simultaneously, potentially covering multiple missing packets. All received packets are acknowledged. However, *ACK* frames are only sent after receiving an ACK-eliciting packet. A QUIC packet is called ACK-eliciting if it contains at least one frame other than *ACK*, *padding*, or *connection close*.

[1]https://datatracker.ietf.org/doc/html/rfc8439#appendix-B

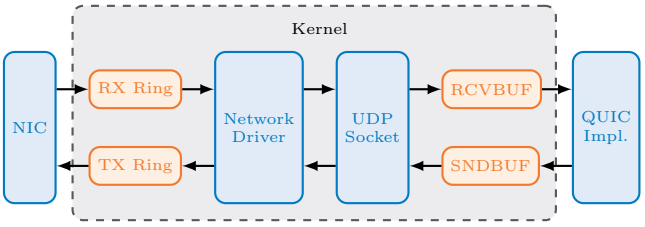

Fig. 1: Simplified overview of available system buffers relevant for a QUIC implementation.

Other than TCP, QUIC sends ACKs encrypted, inducing additional cryptographic workload both on sending and receiving side. Therefore, QUIC must determine how frequently ACKs are sent. The *maximum ACK delay* transport parameter defines the time the receiver can wait before sending an *ACK* frame [3]. Determining the acknowledgment frequency is a trade-off and may affect the protocol's performance. Fewer ACKs can lead to blocked connections due to retransmissions and flow control, while more put additional load on the endpoints.

Furthermore, sending and receiving ACKs with QUIC is more expensive than with TCP. With TCP, ACKs do not take any additional space since they are part of the TCP header and thus can be piggybacked on sent data. The kernel evaluates them before any cryptography is performed. We show the impact of acknowledgment processing and sending in Section IV.

C. Buffers

Different buffers of the system can impact the performance of QUIC. Figure 1 shows a simplified schema of buffers managed by the kernel, which are relevant for the measurement scenarios of this paper. The NIC stores received frames to the RX ring buffer using Direct Memory Access (DMA) without kernel interrupts. Conversely, it reads frames from the TX buffer and sends them out. The kernel reads and writes to those ring buffers based on interrupts and parses or adds headers of layers 2 to 4, respectively. Afterward, the UDP socket writes the payload to the Receive Buffer (RCVBUF).

However, if the ring buffer or RCVBUF is full, packets are dropped, and loss occurs from the perspective of the QUIC implementation. The default size of the receive buffer in the used Linux kernels is 208 KiB. In contrast, if the Send Buffer (SNDBUF), which resides between the implementation and the socket, is packed, the QUIC implementation will be blocked until space is available. Both scenarios reduce the goodput. We show the impact of different buffer sizes in Section IV-C.

D. Offloading

The Linux kernel offers different NIC offloading techniques, namely TCP Segmentation Offload (TSO), Generic Segmentation Offload (GSO), and Generic Receive Offload (GRO). While the first only affects TCP, the others also apply to UDP and thus QUIC. The main idea of all offloading techniques is

to combine multiple segments and thus reduce the overhead of processing packets. Following the ISO / OSI model, data sent by TCP is supposed to be split into chunks smaller or equal to the Maximum Segment Size (MSS) and then passed down to lower layers. Each layer adds its header and passes the data further. All of this is computed by the kernel. With hardware offloading, this segmentation task can be outsourced to the NIC. Offloading can only accelerate sending and receiving of packets, not the data transfer over the network [9].

So far, most offloading functions are optimized regarding TCP. QUIC profits less since packetization is done in user space. Utilizing offloading would require adjustments in the implementation to offload tasks like cryptography or segmentation [10]. In Section IV-E we compare the impact of different offloading functions on QUIC and TCP / TLS.

III. Measurement Framework

In this paper, we extend QIR initially proposed by Seemann and Iyengar [6]. It is designed to test different QUIC implementations for interoperability and compliance with the QUIC standard reporting results on a website [7]. Servers and clients from multiple implementations are tested against each other, performing various test cases, like handshake, 0-RTT, and session resumption. The main focus is on the implementations' functional correctness and less on performance. Even though all followed the same RFC drafts during specification, interoperability between different servers and clients was only present for some features. Implementations run inside Docker containers, and the network in between is simulated using *ns-3*[2]. Simple goodput measurements are available but only conducted on a 10 Mbit/s link. As of January 2023, nearly all tested libraries are close to the possible maximum goodput [7].

QIR orchestrates the measurements by configuring used client and server applications, creating required directories for certificates and files, and collecting log files and results afterward. In the sense of reproducibility, these features make QIR a powerful tool for QUIC (but not limited to) evaluations.

We extended QIR to enable high-speed network measurements on dedicated hardware servers. Figure 2 shows an overview of our framework and its features. For this work, we use different physical servers for the client and server implementation to prevent them from impacting each other. We add additional configuration parameters, such that measurements can be specified with a single configuration file, and include version fingerprints both from the implementations and QIR to the output to foster reproducibility. Additionally, we extend the logging by including several tools which collect data from different components (*e.g.*, the NIC, CPU, and sockets).

We follow these three requirements in the implementation of our measurement framework:

Flexibility: Any QUIC implementation can be used as long as it follows the required interface regarding how client / server are started and variables are passed to them. We provide example implementations for the ones given in Section III-D.

[2]https://www.nsnam.org/

Portability: The framework can be deployed to any hardware infrastructure, as depicted in Figure 2, requiring a link between the client and server nodes and `ssh` access from the management node.

Reproducibility: Experiments are specified in configuration files and results contain needed information on how they were generated, *e.g.*, the QIR and QUIC implementation version. Additionally, results contain a complete description of the used configuration, versions, and hardware.

Our code, results, and analysis scripts are available:

https://github.com/tumi8/quic-10g-paper

This includes our extension of QIR, all measurement configurations, and results shown in the paper, analysis scripts to parse the results, and Jupyter notebooks for visualization [11].

A. Workflow

We followed a similar workflow as QIR. First, our framework configures the used hardware nodes, especially the used network interfaces for the measurements. Each measurement executes four different scripts that the implementation or configuration can adjust: setup environment, pre-scripts, run-scripts, post-scripts as shown in Figure 2. The setup script can be used to install local dependencies like Python environments. Pre- and post-scripts can be utilized to configure OS-level properties like the UDP buffer size and reset them after the measurement. Also, additional monitoring scripts can be started and stopped accordingly.

Relevant configuration parameters for the client / server implementation, such as IP address, port, and X.509 certificate files, are passed via environment variables. Then the server application is started, waiting for incoming connections. Subsequently, the client application conducts a QUIC handshake and requests a file from the server via HTTP/3. Once the client terminates, the framework checks whether all files were transferred successfully and executes post-scripts to stop monitoring tools and to reset the environment to a clean state for subsequent measurements. Finally, the framework collects all logs.

B. Hardware Configuration

We decided to run the client and server applications on different hardware nodes to prevent interference and to fully include the kernel and NIC (other than with Docker). The QIR runs on another host and functions as a management node orchestrating the measurement. The management node can access the measurement nodes via `ssh`, while the others are connected via a 10 Gbit/s link, as shown in Figure 2. Different network interfaces and links are used for management and measurements. If not stated differently, all measurements were executed on *AMD EPYC 7543 32-Core* processors, 512 GB memory, and *Broadcom BCM57416 NICs*. As an operating system, we use *Debian Bullseye* on *5.10.0-8-amd64* for all measurements without additional configurations. We rely on live-boot images with RAM disks, drastically increasing I/O speed to focus on the network aspect.

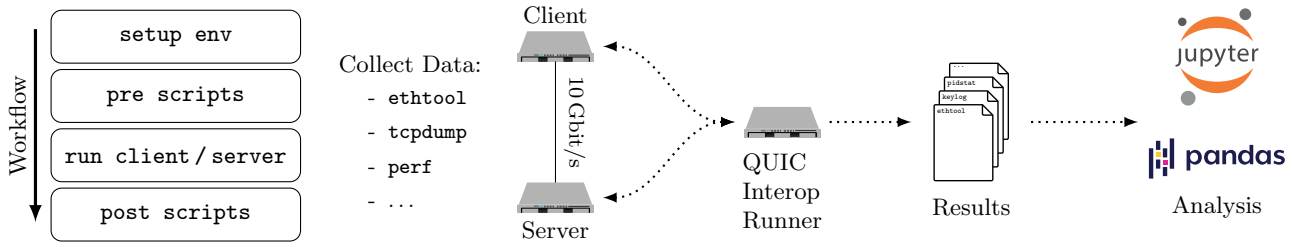

Fig. 2: Hardware architecture, measurement workflow, and analysis pipeline.

C. Collected Data and Analysis

For each measurement, the framework computes the goodput as the size of the transmitted file divided by the time it takes to transmit it. To reduce this impact of transmission rate fluctuations (*e.g.*, caused by congestion control), a large file size (8 GB) is chosen to enforce a connection duration of several seconds.

The following monitoring tools are directly integrated into our framework to collect more data from within the implementations or from the hardware hosts. **tcpdump** is used to collect packet traces which can be decrypted with the exported session keys. Additionally, implementations can enable **qlog** (a schema for logging internal QUIC events [12]) and save results to a directory set up and collected by the framework. However, both result in extensive logging that heavily impacts performance and are only considered for debugging. **ifstat**, **ethtool**, **netstat**, and **pidstat** can be started to collect additional metrics from the NIC and CPU, such as the number of dropped packets or context switches. Finally, **perf** can be used for in-detail client and server application profiling. This allows to analyze how much CPU time is consumed for tasks like encryption, sending, receiving, and parsing packets. When enabled, the output of the used tools is exported along with the measurement results.

We provide a parsing script for all generated data that handles all different output formats of the used tools. By this, it is possible to analyze the change of goodput with different configurations and get a better understanding of why this is the case. It is possible to detect client and server-side bottlenecks as root causes for performance limitations. We consider this important since we observed multiple different QUIC components limiting the performance depending on the configuration. The analysis pipeline parses available result files and outputs the final results as Python *pandas* DataFrame.

D. Implementations

Since there is not one widely used implementation for QUIC (such as Linux for TCP), we evaluate multiple implementations written in different languages. The implemented framework currently includes six QUIC libraries namely: aioquic [13], quic-go [14], mvfst by Facebook [15], picoquic by Private Octopus [16], quiche by Cloudflare [17], and LSQUIC by LiteSpeed Tech [18]. For each implementation, we either used provided examples for the client and server or made minor adjustments to be compatible with QIR.

In this paper, we mainly focus on LSQUIC and quiche since they show the highest goodput rates, as shown in Section IV-A. They are implemented in Rust and C, respectively, and are already widely deployed [19]. Furthermore, both libraries rely on BoringSSL for TLS. Thus, cryptographic operations and TLS primitives are comparable, and we can focus on QUIC specifics in the following. We compare them to the remaining libraries in Section IV-A to put their initial performance into perspective and further support our selection. We base our initial client and server on their example implementation and adapt only where necessary. Additionally, we compare QUIC with a TCP / TLS stack consisting of a server using `nginx` (version 1.18.0) and a client using `wget`.

IV. Evaluation

We apply the measurement framework to evaluate the performance of different QUIC implementations. For every measurement, the client downloads an 8 GB file via HTTP/3. Every measurement is repeated 50 times to assure repeatability. The boxplots show the median as a horizontal line, the mean as icons such as ▲, and the quartiles Q_1 and Q_3. All implementations come with different available Congestion Control (CC) algorithms. For LSQUIC, we observed unintended behavior with *BBR*, resulting in several retransmissions and only between 30 % to 70 % of the goodput of *Cubic*. We assume this is related to a known issue with *BBR* in TCP [20]. To prevent significant impact from different algorithms, we set each implementation to *Cubic* if available or *Reno* otherwise.

A. Baseline Measurements

QIR is designed to test operability between different implementations. As a first baseline, we evaluate the client and server of the implementations listed in Section III-D. Figure 3 shows the goodput for each client-server pair for all implementations. It clearly shows that the goodput widely differs (52 Mbit/s to 3004 Mbit/s) between the implementations.

The Python library aioquic shows the lowest performance, as expected since it is the only interpreted and not compiled implementation. It can be considered an implementation suitable for functional evaluation or a research implementation. When it comes to goodput, it can be neglected. The two best-performing implementations are LSQUIC and quiche. Regarding picoquic, we are not able to reach similar goodput

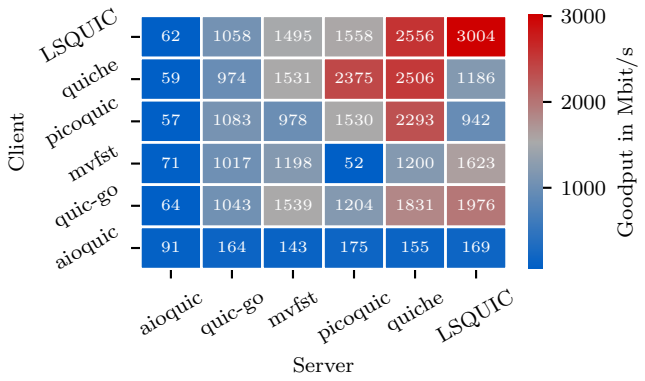

Fig. 3: Baseline goodput results for different QUIC libraries tested against each other on a 10 Gbit/s link. In comparison TCP / TLS reaches 8010 Mbit/s in the baseline scenario.

as reported by Tyunyayev et al. [21] and were not able to easily reproduce the required optimizations. Therefore, we stick with LSQUIC-LSQUIC and quiche-quiche pairs for the following evaluation since they achieved the highest goodput.

Like interoperability of QUIC features, goodput performance widely varies among client-server pairs. We conjecture that different operation modes, QUIC parameters, and efficiency of the used components result in these fluctuations. The LSQUIC server achieves the highest rate with the LSQUIC client while being less performant with clients of other implementations. Overall, the quiche server achieves decent goodput with most clients. The picoquic-mvfst measurements achieve peculiar low rates of only 52 Mbit/s.

Key take-away: *The goodput of different QUIC libraries varies drastically, not only between libraries in general but also between clients and servers. LSQUIC and quiche perform best in the baseline scenario. Only in the case of LSQUIC as server and quiche as client the goodput drops drastically.*

B. Performance Profiling

We use **perf** to analyze LSQUIC and quiche further and see the CPU time consumption for each component. During each measurement, perf collects samples for the complete system. We categorize the samples for both implementations based on their function names and a comparison to the source code. Figure 4 shows the results for the respective servers. *Packet I/O* covers sending and receiving messages, especially the interplay with the UDP socket and kernel functionality (sendmsg and recvmsg). *I/O* covers reading and writing the transmitted file by server and client. *Crypto* describes all TLS-related en- / decryption tasks. *Connection Management* covers packet and acknowledgment processing and managing connection states and streams. If no function name can be collected by **perf** or if we cannot map it to a category, we use *Uncategorized*.

We manually created this mapping by assigning each function occurring in the perf dump to a category. While we could not assign all functions, a clear trend is visible. The strict inclusion of TLS is often criticized due to the expectation of a high overhead [22]. However, *Packet I/O* takes up a majority of resources for both libraries. Passing packets from the libraries in user space to the kernel, the NIC, and vice versa currently impacts the performance most.

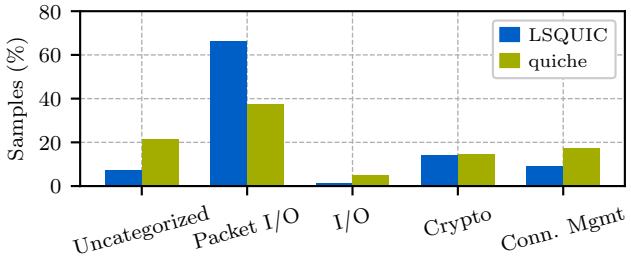

Fig. 4: Distribution of server perf samples across different categories. The results are shown in relation to the total number of samples collected by perf (including idle states).

Interestingly, we see differences in performance in the used programming language or architecture and between the two client and server combinations of different implementations. Resulting in the highest difference, the goodput with a quiche server and LSQUIC client is more than twice as high as vice versa. During these measurements, both the client and server are only at around 70 % CPU usage, no loss is visible, and no additional retransmission occurs. In this case, the bottleneck seems to be the interaction of the flow control mechanisms of both libraries in this client / server constellation. Besides general interoperability tests, our measurement framework can be used in the future to identify these scenarios and to improve QUIC libraries, their flow and congestion control mechanisms, and their interaction.

Key take-away: *Our findings show that the most expensive task for QUIC is Packet I/O. While the cost of crypto operations is visible, it is not the main bottleneck here. Overall, our results comply with related work [10], and we share our mappings so that they can be refined and extended in the future.*

C. Buffers

As explained in Section II-C, different system buffers affect a QUIC connection. The essential buffers are the ring buffers in-between the NIC and network driver, and the send and receive buffers from the socket used by the library, as shown in Figure 1.

To analyze the impact of these buffers, the measurement framework captures network driver statistics before and after each data transmission using **ethtool** and connection statistics using **netstat**. It provides the number of sent and received packets and dropped packets in each of the mentioned buffers.

The baseline measurement shows that the ring buffers drop no data. However, packets are dropped by the client receive buffer since both client implementations retrieve packets slower than they arrive. As a result, retransmissions are required by the server, and congestion control impacts the transmission. To analyze the impact on the goodput, we

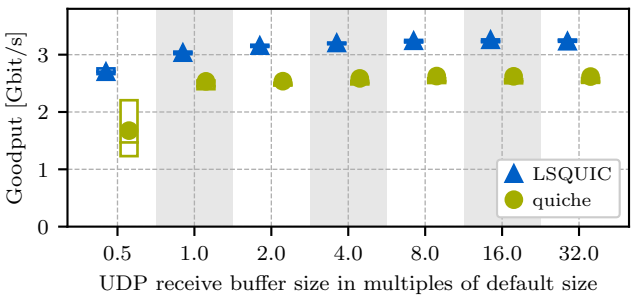

Fig. 5: QUIC goodput with different UDP receive buffer sizes. X-axis values are multiples of the default buffer size of 208 KiB.

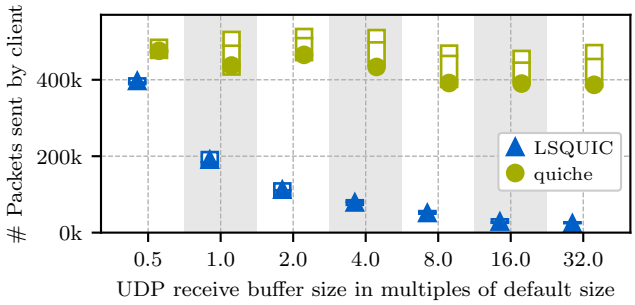

Fig. 6: Number of sent ACKs by the client. X-axis values are multiples of the default buffer size of 208 KiB.

increase the default receive buffer (208 KiB). The effect of this can be seen in Figure 5. It shows the goodput of LSQUIC and quiche pairs for different buffer sizes as multiple of the default buffer on the x-axis. The goodput for both libraries improves with increasing buffer sizes and stabilizes after a 16-fold increase.

Using LSQUIC with the default buffer size results in 11.4 k dropped packets and a loss rate of 0.2 %. With the largest tested buffer size, LSQUIC reaches a goodput of 3250 Mbit/s, an 8.7 % increase compared to the baseline shown in Figure 3. Besides the reduced loss and retransmissions, the number of ACKs sent by the LSQUIC client is drastically reduced from 180 k to 46 k (26 %). This development for all tested buffer sizes is shown in Figure 6. The reduction of sent ACK packets also results in reduced CPU usage by the client while increasing the server CPU usage shifting the bottleneck further towards the sending of QUIC packets. In all scenarios, LSQUIC sends fewer ACKs than reported by Marx et al. [23].

Regarding quiche, only 7 k packets are dropped with the default buffer size, hence a 0.1 % loss rate. Larger buffers decrease the loss by one order of magnitude and increase the goodput but only by 3 % to 2530 Mbit/s. In contrast to LSQUIC, no difference regarding sent ACKs can be seen (see Figure 6). Reducing the receive buffer further impacts both libraries, especially quiche. The goodput drops by 40 % and a more significant deviation is visible. We repeated the baseline measurements with larger UDP receive buffers for which the results can be seen in Figure 10.

Key take-away: *The default UDP receive buffer size has a visible impact on QUIC-based data transmission. We recommend a general increase of the default buffer by at least an order of magnitude to support widespread deployment of QUIC.*

D. Crypto

As explained in Section II-A, QUIC supports TLS using AES or ChaCha20. Generally, operations to encrypt and decrypt are computationally expensive but are widely optimized in hardware and software today. QUIC and TLS 1.3 only support AEAD algorithms [24], which can encrypt and sign data in one single pass. For AES, only three ciphers are available: `AEAD_AES_128_GCM`, `AEAD_AES_128_CCM`, and `AEAD_AES_256_GCM`. From them, only `GCM` ciphers are required or should be implemented [24], the `CCM` cipher is rarely used. Usually, the 128 bit `GCM` cipher is preferred [19]. For the following evaluation, we use the default cipher. ChaCha20 always uses the `Poly1305` authenticator to compute message authentication codes.

We evaluated the QUIC implementations in three different scenarios. Two of them use AES either without or with the AES New Instruction Set that offers full hardware acceleration [25] (which is the default in our test environment), and one scenario uses ChaCha20. For the following, we refer to the hardware-accelerated AES version as AES-NI. LSQUIC and quiche automatically fall back to ChaCha20 whenever hardware acceleration is unavailable. To evaluate AES without hardware acceleration, we patched the used `BoringSSL` library accordingly.

As seen in Figure 7, ChaCha20 achieves the same throughput as AES-NI for both QUIC libraries. While removing hardware acceleration decreases the goodput with AES by 11 % for LSQUIC and even by 51 % for quiche. This difference between the implementations results from the quiche client sending more than 16 times as many ACKs than LSQUIC. While the server is the bottleneck in all measurements, more packets sent by the client add additional load to the server for receiving and decrypting packets (see Section IV-C). This decryption is more expensive without hardware acceleration. Thus, crypto has a higher impact on the overall goodput. Also, we observe that the client CPU utilization of quiche is 5 % lower with ChaCha20 than with AES.

With TCP / TLS, the effect changes. While ChaCha20 reaches higher goodput rates than AES without hardware acceleration, AES-NI outperforms ChaCha20 with a three times higher goodput, almost reaching the link rate. Additionally, for TCP / TLS we observed that the client CPU bottlenecked the connection for AES-NI and ChaCha20, while the server was the bottleneck without hardware acceleration.

Key take-away: *While selecting an appropriate cipher suite drastically impacts TCP / TLS, QUIC reaches similar goodput for AES-NI and the ChaCha20-based cipher. The main bottleneck is still packet I/O and the acceleration effect is further reduced due to smaller TLS records and encrypted ACKs.*

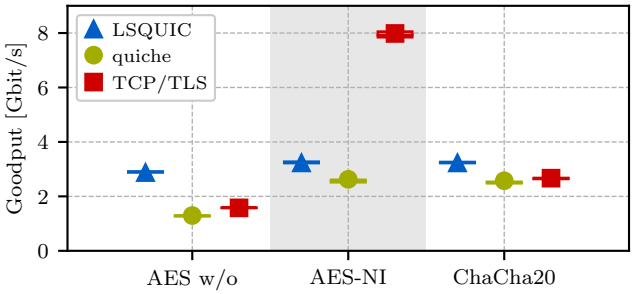

Fig. 7: Impact of different TLS ciphers on QUIC and TCP/TLS goodput. During the AES measurement without hardware acceleration, the implementations were forced to use AES by deactivating ChaCha20 in the respective TLS library to prevent the fallback.

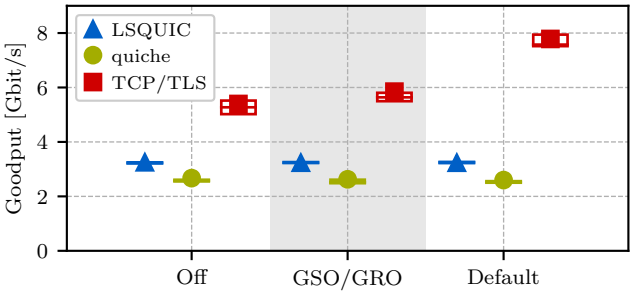

Fig. 8: Impact of hardware offloading on QUIC and TCP/TLS goodput. By default, GSO, GRO, and TSO are activated.

E. Segmentation Offloading

To reduce the impact of packet I/O, a QUIC library can combine multiple packets and rely on offloading. Similarly, an implementation can use *sendmmsg*[3] and *recvmmsg*[4] to reduce system calls and move multiple packets to/from the kernel space within a single buffer.

By default, Generic Segmentation Offload (GSO), Generic Receive Offload (GRO), and TCP Segmentation Offload (TSO) are activated in Linux. We analyze which offload impacts QUIC and TCP by incrementally activating different offloading techniques. Figure 8 indicates that all the offloading techniques hardly influence QUIC goodput, while TCP largely profits from TSO. The QUIC goodput does not change when GSO/GRO is enabled. However, the client CPU utilization increases from 82 % to 92 % for LSQUIC and from 77 % to 84 % for quiche. Turning off all offloads does not decrease goodput but decreases the client's CPU utilization and also power consumption.

While LSQUIC implements functionality to support *sendmmsg* and *recvmmsg*, we were not able to use it as of January 2023. Data transmission with the functionality activated randomly terminated with exceptions. We expect improved support for those features also by other implementations and suggest reevaluating libraries with our framework in the future.

Key take-away: *The tested QUIC implementations do not profit from any segmentation offloading techniques as of January 2023. Compared with TCP, there is much room for improvement to apply the same benefits to UDP and QUIC. Since QUIC encrypts every packet individually, including parts of the header, techniques similar to TSO cannot be applied, which would require that headers can be generated in the offloading function. However, with adjustments to the protocol and the offloading functions, speedups can be achieved for segmentation and crypto offloading [10].*

[3]https://man7.org/linux/man-pages/man2/sendmmsg.2.html
[4]https://man7.org/linux/man-pages/man2/recvmmsg.2.html

TABLE I: Different CPUs used for the measurements. The default for measurements was AMD-2 if not noted otherwise.

	CPU	Year	max GHz
AMD-1	AMD EPYC 7551P	2017	2.0
AMD-2	AMD EPYC 7543	2021	3.7
Intel-1	Intel Xeon CPU D-1518	2015	2.2
Intel-2	Intel Xeon CPU E5-1650	2012	3.8
Intel-3	Intel Xeon Gold 6312U	2021	3.6

F. Hardware

Even though an implementation in user space comes with more flexibility, tasks that are done by the kernel for TCP become more expensive with QUIC. More CPU cycles are required per packet.

We repeated the final goodput measurements with increased buffers on five host pairs with different generations of AMD and Intel CPUs listed in Table I. Client and server hosts are equipped with the same CPU for each measurement. Additionally, we optimized LSQUIC and quiche further by adding compile flags to optimize for the used architecture. The optimized implementations are referred to as LSQUIC* and quiche*.

The results in Figure 9 show that quantizing QUIC throughput highly depends on the used CPU and architecture. Both QUIC and TCP profit from newer CPUs with more modern instruction sets. Compile flags improved the LSQUIC goodput by 14 % to 20 %, while they hardly affected quiche. Also, QUIC and TCP perform differently among the different architectures. For example, when comparing the two newest CPUs AMD-2 and Intel-3, QUIC performs 27 % better on the Intel chip, almost reaching 5 Gbit/s. On the other side, TCP goodput decreases by 3 % compared to the CPU in the AMD-2 host pair. This shows that QUIC (in the user space) and TCP (in the kernel) profit differently from CPU architectures and instruction sets.

Key take-away: *The used hardware is highly relevant for evaluating QUIC performance. Newer CPUs lead to a higher goodput for both QUIC implementations even though their frequency is slightly lower, e.g., comparing Intel-2 and Intel-3. While not feasible for all research groups, we suggest attempting an evaluation of potential improvements to QUIC*

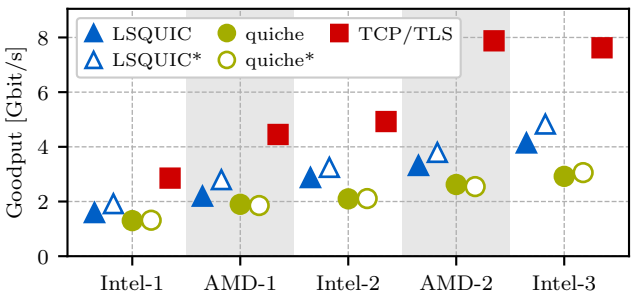

Fig. 9: Goodput as measured on different hardware architectures listed in Table I. LSQUIC* and quiche* are built with compile flags to optimize for the used architecture.

libraries on different CPU generations and frequencies in the future to better quantify their impact.

V. Related Work

The interoperability of different QUIC implementations has been tested by research throughout the protocol specification. Seemann and Iyengar [6], Piraux et al. [26], and Marx et al. [23] developed different test scenarios and analyzed a variety of QUIC implementations. Furthermore, Rüth et al. [27], Piraux et al. [26], and Zirngibl et al. [19] have already shown a wide adoption of QUIC throughout the protocol specification and shortly before the release of RFC9000 [3]. They show that multiple large corporations are involved in the deployment of QUIC, various implementations are used, and different configurations can be seen (*e.g.*, transport parameters). However, they mainly focused on functionality analyses, interoperability, and widespread deployments but did not focus on the performance of libraries.

Different related works analyzed the performance of QUIC implementations [10, 21, 28–32]. However, they either analyzed QUIC in early draft stages, *e.g.*, Megyesi et al. [30] in 2016, or mainly focused on scenarios covering small web object downloads across multiple streams, *e.g.*, by Sander et al. [29] and Wolsing et al. [32]. Similar to our work, Yang et al. [10] orchestrated QUIC implementations to download a file. With a single stream, they reached a throughput between 325 Mbit/s and even 4121 Mbit/s for Quant. However, they omit HTTP/3 and only download a file of size 50 MB. Therefore, the impact of the handshake and cryptographic setup is higher, and other effects might be missed.

Endres et al. [33] adapted the QIR to evaluate QUIC implementations similar to the starting point of our framework. However, they still relied on the virtualization using docker and network emulation using ns-3 and focused on a different scenario, namely satellite links.

Tyunyayev et al. [21] combined picoquic with the Data Plane Development Kit (DPDK) to bypass the kernel. They compared their implementation to other QUIC stacks and increased the throughput by a factor of three. They argue that the speedup is primarily due to reduced I/O but do not investigate other factors in more detail.

VI. Conclusion

In this work, we analyzed the performance of different QUIC implementations on high-rate links and shed light on various influence factors. We systematically created a measurement framework based on the idea of the QUIC Interop Runner. It allows automating QUIC goodput measurements between two dedicated servers. It can use different QUIC implementations, automate the server configuration, collect various statistics, *e.g.*, from the network device and CPU statistics, and provide means to collect, transform, and evaluate results.

We applied the presented framework to evaluate the goodput of mainly LSQUIC and quiche on 10 Gbit/s links and analyzed what limits the performance. A key finding in this work is that the UDP receive buffer is too small by default, which leads to packets getting dropped on the receiver side. This results in retransmissions and a reduced goodput. We show that increasing the buffer by at least an order of magnitude is necessary to reduce buffer limits in high link rate scenarios. We observed several differences in the behavior of LSQUIC and quiche, such as differing default parameters, *e.g.*, the UDP packet size or a diverse approach regarding the acknowledgment sending rate. When comparing different TLS 1.3 ciphers, QUIC almost reaches the same goodput with ChaCha20 as with the hardware-accelerated AES ciphers, which behaves differently with TCP. We could not measure any performance increase with the support of segmentation offloading features of the operating system. Finally, we show that evaluating QUIC highly depends on the used CPU. By applying various optimizations, we increased the goodput of LSQUIC by more than 25 % and achieved up to 5 Gbit/s on Intel CPUs.

Even though QUIC has many similarities to TCP and the specification took several years, our work shows that many details already analyzed and optimized for TCP are still limiting QUIC. Furthermore, the variety of implementations complicates a universal evaluation and yields further challenges to improve performance in the interplay of libraries, *e.g.*, the drastically reduced goodput using an LSQUIC server and quiche client, as shown in Section IV-A.

To allow for an informed and detailed evaluation of QUIC implementations in the future, we publish the framework code, the analysis scripts, and the results presented in the paper [11]. The measurement framework can be applied to evaluate future improvements in QUIC implementations or operating systems.

Acknowledgment

The European Union's Horizon 2020 research and innovation programme funded this work under grant agreements No 101008468 and 101079774. Additionally, we received funding by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria. This work is partially funded by Germany Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK001K) and 6G-ANNA (16KISK107).

References

[1] Jim Roskind. (June 27, 2013) Experimenting with QUIC. Accessed: 2023-02-10. [Online]. Available: https://blog.chromium.org/2013/06/experimenting-with-quic.html

[2] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 183–196.

[3] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9000.txt

[4] T. Pauly, E. Kinnear, and D. Schinazi, "An Unreliable Datagram Extension to QUIC," RFC 9221, Mar. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9221

[5] I. Q. W. Group. (2023) Implementations. Accessed: 2023-02-10. [Online]. Available: https://github.com/quicwg/base-drafts/wiki/Implementations

[6] M. Seemann and J. Iyengar, "Automating QUIC Interoperability Testing," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20. Association for Computing Machinery, 2020, pp. 8–13.

[7] ——. (2020) QUIC Interop Runner. Accessed: 2023-02-10. [Online]. Available: https://interop.seemann.io/

[8] M. Thomson and S. Turner, "Using TLS to Secure QUIC," RFC 9001, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9001.txt

[9] The kernel development community. (2023) Segmentation Offloads. Accessed: 2023-02-10. [Online]. Available: https://docs.kernel.org/networking/segmentation-offloads.html

[10] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making QUIC Quicker With NIC Offload," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20, 2020, p. 21–27. [Online]. Available: https://doi.org/10.1145/3405796.3405827

[11] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle. (2023) Code and Data Publication. [Online]. Available: https://github.com/tumi8/quic-10g-paper

[12] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, "Main logging schema for qlog," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-qlog-main-schema-04, Oct. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/04/

[13] aiortc. (2023) aioquic. Accessed: 2023-02-10. [Online]. Available: https://github.com/aiortc/aioquic

[14] L. Clemente and M. Seemann. (2023) quic-go. Accessed: 2023-02-10. [Online]. Available: https://github.com/quic-go/quic-go

[15] Facebook. (2023) mvfst. Accessed: 2023-02-10. [Online]. Available: https://github.com/facebookincubator/mvfst

[16] Private Octopus. (2023) picoquic. Accessed: 2023-02-10. [Online]. Available: https://github.com/private-octopus/picoquic

[17] Cloudflare. (2023) quiche. Accessed: 2023-02-10. [Online]. Available: https://github.com/cloudflare/quiche

[18] LiteSpeed Tech. (2023) lsquic. Accessed: 2023-02-10. [Online]. Available: https://github.com/litespeedtech/lsquic

[19] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle, "It's over 9000: Analyzing early QUIC Deployments with the Standardization on the Horizon," in *Proc. ACM Int. Measurement Conference (IMC)*, 2021.

[20] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, "Towards a Deeper Understanding of TCP BBR Congestion Control," in *2018 IFIP networking conference (IFIP networking) and workshops*. IEEE, 2018, pp. 1–9.

[21] N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette, "A High-Speed QUIC Implementation," in *Proceedings of the 3rd International CoNEXT Student Workshop*, ser. CoNEXT-SW '22, 2022, p. 20–22.

[22] (2020) QUIC IETF Mailinglist. Accessed: 2023-02-10. [Online]. Available: https://mailarchive.ietf.org/arch/msg/quic/SBetxLwCq5I7un2tkzFb7tXhJMU/

[23] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 14–20.

[24] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[25] S. Gueron. (2010) Intel® Advanced Encryption Standard (AES) New Instructions Set. Accessed: 2023-02-10. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[26] M. Piraux, Q. De Coninck, and O. Bonaventure, "Observing the Evolution of QUIC Implementations," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 8–14.

[27] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, "A First Look at QUIC in the Wild," in *Proc. Passive and Active Measurement (PAM)*. Springer International Publishing, 2018.

[28] A. Yu and T. A. Benson, "Dissecting performance of production quic," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1157–1168. [Online]. Available: https://doi.org/10.1145/3442381.3450103

[29] C. Sander, I. Kunze, and K. Wehlre, "Analyzing the Influence of Resource Prioritization on HTTP/3 HOL Blocking and Performance," in *Proc. Network Traffic Measurement and Analysis Conference (TMA)*, 2022.

[30] P. Megyesi, Z. Krämer, and S. Molnár, "How quick is QUIC?" in *Proc. IEEE ICC*, 2016, pp. 1–6.

[31] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai, "Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1366–1381, 2022.

[32] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–7.

[33] S. Endres, J. Deutschmann, K.-S. Hielscher, and R. German, "Performance of QUIC Implementations Over Geostationary Satellite Links," 2022. [Online]. Available: https://arxiv.org/abs/2202.08228

Appendix

We repeated the baseline measurements presented in Section IV-A but with increased UDP buffer size and the optimized implementations for LSQUIC and quiche. As seen in Figure 10, LSQUIC profits from the optimizations and the larger buffer while other implementations such as aioquic and quic-go are limited by other other bottlenecks.

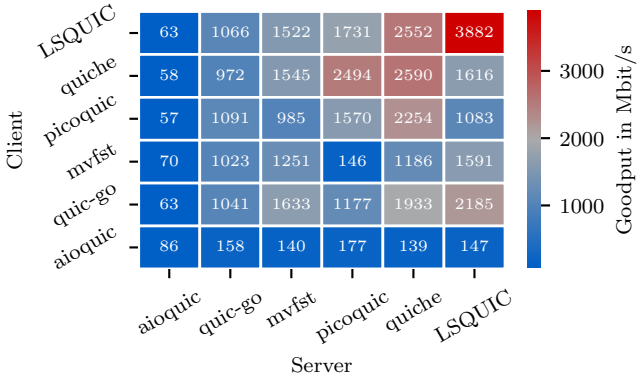

Fig. 10: Goodput results for all implementations with increased buffer sizes and optimize compile flags for each implementation.