If you cite this paper, please use the NOMS reference: J. Späth, S. Lachnit, M. Kempf, K. Holzinger, G. Carle, J. Zirngibl, "Kernel Bypass Surgery: A Viable Procedure for Maximizing QUIC Bandwidth?," NOMS 2026-2026 IEEE Network Operations and Management Symposium, Rome, IT, 2026.

Kernel Bypass Surgery: A Viable Procedure for Maximizing QUIC Bandwidth?

Johannes Späth*, Stefan Lachnit*, Marcel Kempf*, Kilian Holzinger*, Georg Carle*, Johannes Zirngibl†
* Technical University of Munich, Germany,
{spaethj, lachnit, kempfm, holzinger, carle}@net.in.tum.de
† Max Planck Institute for Informatics, Germany,
jzirngib@mpi-inf.mpg.de

***Abstract*—QUIC is a protocol that aims to solve many problems of the TCP/TLS stack. Previous work has shown that the performance of those implementations differs greatly in high-bandwidth scenarios. One major reason for this is that QUIC is usually implemented in user space, building on the existing UDP functionality offered by the kernel network stack. While this design decision enables fast development cycles, it comes at the cost of a high number of context switches between kernel and user space. In this paper, we investigate the impact of kernel bypass on performance. We integrated the Data Plane Development Kit (DPDK) into three QUIC implementations, bypassing their use of the kernel networking stack, and updated a fourth stack that previous work combined with DPDK. We compare two deployment scenarios for our DPDK QUIC implementations: (1) running exclusively on the NIC port, and (2) sharing a port with the kernel using Single Root I/O Virtualization (SR-IOV). Our analysis shows that kernel bypass can greatly increase the goodput achieved with QUIC, with a speedup reaching up to a factor of 3× and goodput over 10 GBit/s. However, the reachable performance increase highly depends on the implementation, and in one case performance did also not further increase when using DPDK. Furthermore, we show that offloading techniques can reach similar performance if offered by the system and used correctly by the implementations.**

I. Introduction

While QUIC is often seen only as the foundation of HTTP/3 and web scenarios, it gains increasing interest and adoption in high-throughput scenarios. Use cases, *e.g.*, video streaming and large-scale file distribution, integrate QUIC to leverage its extensibility and security features. There is ongoing work on Media over QUIC by the Internet Engineering Task Force, and implementors of file sharing protocols like SMB started to integrate it [1], [2].

A primary goal during the standardization of QUIC was to allow fast development cycles, library flexibility, and expandability. Furthermore, TLS is included by default, requiring asymmetric cryptography. Therefore, QUIC is designed to be implemented in user space and only relies on the UDP data path of the system. A majority of available QUIC libraries follows this paradigm. Therefore, each QUIC packet has to be sent and received by the library in user space via the kernel and NIC. Related work has shown that this can result in a drastic packet I/O overhead, especially if each packet is exchanged between user and kernel space individually [3], [4], [5]. While this is not necessarily a problem in normal web scenarios, new use cases will require optimized throughput.

On one side, offloading, *e.g.*, Generic Receive Offload (GRO) and Generic Segmentation Offload (GSO), can reduce this impact and drastically improve performance. However, these optimizations are rather new for UDP, and they require support by the application, which is not yet the case for many QUIC stacks. On the other hand, kernel bypass technologies, *e.g.*, the Data Plane Development Kit (DPDK), could be used to circumvent the packet I/O bottleneck. This has repeatedly been suggested [4], [6], [7] and initially been tested by Tyunyayev et al. [8] with a single library, *picoquic*. However, libraries follow different paradigms in how they implement QUIC. Further research is required to assess the effort to add DPDK to different libraries, the impact of these changes, and the interoperability. This does not only impact general functionality but also secondary attributes, *e.g.*, the effectiveness of flow and congestion control or pacing.

In detail, our contributions are:

(i) We integrate DPDK into the QUIC libraries *MsQuic*, *LSQUIC*, and *quiche*, and update *picoquic-DPDK*. We publish these artifacts to allow further extensions, tests, or improvements. We highlight that each QUIC library requires a custom integration of DPDK.

(ii) We compare the impact of DPDK on the performance of these QUIC libraries. Especially the implementations with rather low goodput without DPDK achieve a significant speedup of up to 3×, with packet I/O overhead being reduced by up to 90 %. The fastest DPDK library achieves a goodput of 10.8 Gbit/s in our measurements.

(iii) We compare the performance of QUIC with DPDK to the original libraries relying on the kernel networking stack. In that regard, we show that only one of the four implementations utilizes both GSO and GRO. With those optimizations, it also reduces packet I/O overhead and achieves a goodput of 8.3 Gbit/s, comparable to the corresponding DPDK version.

(iv) We show that DPDK can be practically deployed using Single Root I/O Virtualization (SR-IOV) virtual functions, such that it does not require exclusive access to a NIC and with no negative impact on performance. In our experiments, this configuration even achieved slightly higher and more stable goodput, which we attribute to the *iavf* DPDK driver.

(v) We publish our artifacts (see Appendix Section B).

II. Background

This section provides relevant background for performance bottlenecks in QUIC's packet I/O and optimizations that can be applied to mitigate these bottlenecks.

QUIC Performance Bottlenecks The fact that QUIC builds upon UDP introduces performance challenges due to UDP's inherent lack of transport layer optimizations compared to TCP [9]. A significant bottleneck arises from frequent context switches caused by the interaction with the kernel's socket interface for data transmission. This is intensified in QUIC, where functionalities formerly handled by the kernel's transport layer, such as acknowledgments, are implemented in user space, leading to increased interactions with the UDP socket. Consequently, packet I/O has been identified as a primary performance bottleneck for QUIC implementations [3], [4], [10]. To mitigate this, various techniques can be used. Message batching effectively reduces the number of system calls and context switches by allowing multiple messages to be processed with a single kernel interaction, *e.g.*, utilizing `sendmmsg` and `recvmmsg` system calls.

Segmentation Offload The idea of segmentation offloading is rather old and support for TCP Segmentation Offload (TSO) has been included into the Linux kernel already in 2002 [11]. Support for UDP GSO is relatively new and has been added in 2018 [12]. Applications that want to use this feature need to actively support it, as they need to be able to handle larger UDP datagrams than the usual size, which has to fit into the Maximum Transmission Unit (MTU) of the underlying communication layer. With GSO, messages of up to 64 KiB can be passed to the kernel network stack at once. They will be segmented either there or, in case the hardware supports it, the process is offloaded to the NIC. GRO works accordingly for the receive direction, *i.e.*, the NIC or kernel network stack reassembles multiple datagrams to large messages before passing them to the application. By that, the number of system calls can be reduced, and per-packet processing overheads like checksum calculation and verification can be offloaded.

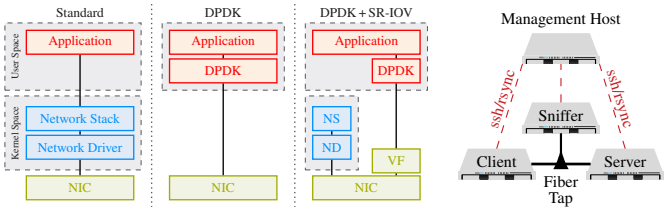
In Linux, the `ethtool` utility allows to configure network driver settings, including UDP segmentation and receive offload. The following settings are relevant for that:

(i) `tx-udp-segmentation`: This option determines whether hardware support for UDP segmentation offload is enabled [13]. If so, the segmentation task is offloaded to the NIC. However, not all NICs and drivers support this feature.

(ii) `generic-segmentation-offload`: GSO serves as a software-based fallback for segmentation offload when the NIC lacks support for this feature [14], but it must be available and enabled even for hardware offloading.

(iii) `rx-gro-hw`: This option enables hardware support for receive offloading [15]. Similar to transmit offloading, the NIC aggregates individual packets before delivering them to the kernel only when this is enabled. Hardware receive offloading requires `rx-checksumming` to be active.

(iv) `generic-receive-offload`: Similar to GSO, GRO acts as the software fallback for receive-side hardware offloading [14]. It must be enabled for coalesced messages to be received from sockets that request this functionality.



(a) Comparison of kernel network stack and bypass architectures.

(b) Topology of the measurement setup.

Fig. 1: QUIC data path options and measurement setup.

Kernel Bypass with DPDK Traditional kernel-based packet processing is designed to be applicable for general-purpose applications. However, achievable performance for specialized applications can be limited by system calls between user and kernel space, the overhead of copying packet buffers, and features that are not always needed. An alternative approach to overcome these limitations is to skip the kernel network stack and move all packet processing to user space. One popular framework that implements this paradigm is the Data Plane Development Kit (DPDK) [16]. DPDK comprises libraries for implementing high-performance user space packet processing applications. It provides user space drivers for many modern NICs that poll actively for packets without introducing additional latency through the use of interrupts. It allows the application to process packets in bursts. Additionally, it provides functions for efficient memory allocation, handling of packet buffers, and parsing of packets. However, usually the complete NIC has to be bound to DPDK (see Figure 1a).

NIC Resource Sharing with SR-IOV SR-IOV is a technology that splits the available resources of a single physical PCIe function into multiple virtual functions. For NICs, multiple virtual ports are created from a single physical port. Received packets are then forwarded to a queue assigned to a virtual port, based on the destination MAC address. This allows to use the NIC with DPDK and the kernel in parallel, as depicted in the right part of Figure 1a.

III. Related Work

The performance of QUIC has been a major area of research since the standardization phase of the protocol [5], [17], [18] and afterwards [3], [4], [8], [10], [19], [20].

Zhang et al. [10] evaluated the performance of QUIC in high-speed networks using browsers without optimizations. Jaeger et al. [3], Kempf et al. [4], and König et al. [7] evaluated multiple QUIC implementations, configurations, and offloading features, respectively. Similarly, König et al. [21] evaluated the effects of offloading, different HTTP versions, and parallelization on QUIC. They further showed the packet I/O overhead by increasing the packet size (*e.g.*, to 9000 B), thus reducing the number of context switches. All these works identified that a major performance bottleneck is the communication between user and kernel space and the resulting CPU and I/O overhead. They further showed the impact of different libraries and their paradigms on the performance. They argued

that offloading can reduce the overall performance impact and suggested considering kernel bypass. Tyunyayev et al. [8] evaluated the impact of kernel bypass on a QUIC library, *i.e.*, they added DPDK to *picoquic*. They reached a threefold increase in throughput, but focused on a single implementation and did not evaluate segmentation offload.

In comparison, we evaluate the impact of DPDK on multiple libraries and their differences. We show that DPDK is not a universal solution that can simply be applied but requires individual optimization based on libraries.

IV. Implementations

We integrate DPDK into three libraries and update *picoquic-DPDK* to DPDK v24.11. Integrating DPDK into QUIC libraries involves two main aspects. First, general networking functionality must be implemented, since DPDK bypasses the kernel networking stack. This includes functionality such as the Address Resolution Protocol (ARP), packet header creation (with checksums), DPDK packet transmission and reception, as well as device and environment initialization, which is independent of the QUIC library. Second, effective DPDK use requires direct integration into each QUIC library, replacing the kernel's UDP socket interface. This integration has to be done for each QUIC library to adapt to its specifics. We highlight some challenges and peculiarities for our implementations in the following.

Microsoft's *MsQuic* implementation ships in the kernel of the Microsoft Windows operating system [22]. However, it supports multiple architectures, including Linux and macOS. *MsQuic* is implemented in C and has a strong focus on performance. The architecture consists of two separate layers, *QUIC* and *platform*, where the former is for protocol-specific logic, while the latter provides an abstraction for operating system-specific parts like TLS, UDP, threads, and locks [22]. *MsQuic* provides configurable execution profiles that optimize thread scheduling for different performance goals, such as minimizing latency or maximizing throughput [22]. In our setup, we select the maximum throughput profile, which dedicates separate threads to the UDP data path and QUIC protocol logic. We use *MsQuic* v2.4.8 for our investigations and as the basis for the DPDK version.

UDP socket communication is abstracted as a *datapath* in the *platform* layer. Therefore, we implement DPDK support as a separate *datapath* in that layer. Some work regarding kernel bypass is already integrated into *MsQuic*, especially with respect to Express Data Path (XDP). There also exists a rudimentary and non-functional DPDK *datapath* in the codebase [23], which we were using as the basis for our DPDK implementation. Following this basis, *MsQuic-DPDK* uses a ring buffer for storing packets to be sent. Packets are enqueued into the buffer one at a time. In the main DPDK loop, the packets are dequeued from the ring and sent out. A separate thread is started for the DPDK busy polling loop, and received packets are delivered using callbacks (*i.e.*, processing happens in that thread). While being abstracted from QUIC logic by the layered architecture of *MsQuic*, changes to the internals of the library were still necessary to include DPDK. This also means that future updates in the *MsQuic datapath* API will affect the DPDK *datapath*.

Litespeedtech's *LSQUIC* is written in C. It includes an HTTP/3 library and uses BoringSSL. In our experiments, we use a modified version, based on release v4.2.0.

In our custom DPDK datapath implementation, we chose the run-to-completion model. This means that all packet processing is performed in the same thread sequentially, fitting well to *LSQUIC* being single threaded. We use the burst version of receive and send functions, as handling multiple packets at once usually improves performance. *LSQUIC* uses `malloc` for memory allocation. DPDK, however, uses hugetables to store packet buffers (`mbufs`). Without further needed copying, those are directly accessible to NICs. To improve throughput, we chose to change packet-related memory allocation of *LSQUIC* to directly write packet data to `mbufs` and, thus, avoiding memory copies. For that purpose, a custom memory allocator was created that keeps track of additionally needed context, such as the related QUIC connection, header type, packet length, and offsets needed to write Ethernet, IP, and UDP headers. DPDK deallocates sent `mbufs` automatically, so a custom implementation of `free` is not required.

Cloudflare's *quiche* is a QUIC implementation in the programming language Rust. It was designed to integrate into existing implementations. Thus, it does not take over the control flow of the application [24]. All interactions with the operating system, including transmission and reception of UDP packets and file I/O, are handled by the application. Packets are received by the application and passed to *quiche*. Instead of using callback functions, received stream data and events related to the connection state have to be actively requested by the application within the active polling loop.

For our DPDK implementation, we use *quiche* version 0.23.2. We use the C wrapper provided by *quiche* to reimplement parts of the example HTTP client and server implementations. Our implementation requires no changes to the library code of *quiche*. By relying only on the relatively stable external API, we minimize the changes required to update to a new version of *quiche*.

In the original example code, the application control flow is managed by an event loop library [25] invoking callback functions on external events. In the DPDK implementation, we replace the event loop library with a busy polling loop. During runtime, the implementation constantly loops through the following operations: (1) check for received packets using the DPDK API and pass them to *quiche*, (2) request and transmit new datagrams from *quiche*, (3) handle file I/O and timeouts, as well as other connection events. Consequently, all processing happens in a single thread. We do not use an additional thread that polls for received packets and buffers them. We rely on the RX ring of the NIC to store packets until they are processed by the application. The number of buffered packets is, therefore, limited by the configured number of available receive descriptors in DPDK. When receiving packets through the DPDK API, we process up to 32 packets

in a burst. For transmission, we request all currently available packets from *quiche* up to a maximum configured burst size of 32. These packets are then transmitted as a burst. Overall, rewriting the interaction between *quiche* and socket-based packet I/O to DPDK is straightforward, primarily involving changes to packet handling and device initialization.

picoquic is a minimalist QUIC implementation written in C. Tyunyayev et al. [8] integrated kernel bypass into this library and evaluated the performance gain. For that, they replaced the existing packet loop with an active polling loop that uses DPDK. For packet transmission, they integrated a DPDK buffer to store packets, which are then sent out in batches. Their implementation attempts to accumulate a maximum number of packets in this buffer prior to transmission. Once no new packets are provided, *e.g.*, when the congestion controller prevents it, the buffer is flushed.

For our measurements, we fixed compatibility issues with DPDK v24.11. Further, we modified the DPDK parameters to match the ones of the other three implementations (see Appendix Table II) for comparability.

V. Measurement Framework

To evaluate the implementations and compare DPDK-based versions to their default library, we rely on measurements in a hardware testbed. We use the framework published by Jaeger et al. [3], which allows automating performance measurements of different libraries. While focusing on the goodput of a QUIC connection, it collects metadata from additional tools, *e.g.*, `perf`, `ethtool`, and `netstat`.

Using DPDK, many of these tools cannot extract metadata anymore. Therefore, we extend our implementations to output similar information and add an optical splitter and a dedicated capture node to our setup, comparable to Kempf et al. [26]. The resulting topology is shown in Figure 1b.

For detailed performance analysis, we use Linux `perf`, which periodically samples the call graph during execution to estimate the relative time spent in each function. We use a similar approach to Jaeger et al. [3] to classify sampled function names into five categories: *I/O*, *Packet I/O*, *Crypto*, *Connection Management*, and *General*.

The *I/O* category includes functions for reading or writing files; *Packet I/O* covers packet sending and receiving, including driver-level handling and kernel-level header processing; *Crypto* contains encryption and decryption of headers and payloads; *Connection Management* includes library functions managing connection and stream states; all remaining functions are categorized as *General*.

All four tested QUIC DPDK implementations use polling, continuously checking for new packets or whether actions like timeout handling are required. Consequently, we measure many `perf` samples in the corresponding functions, which, however, do not reflect their actual resource requirements. To compare the CPU utilization and distribution between different functional categories, we exclude polling from the obtained profiling data using the following approach.

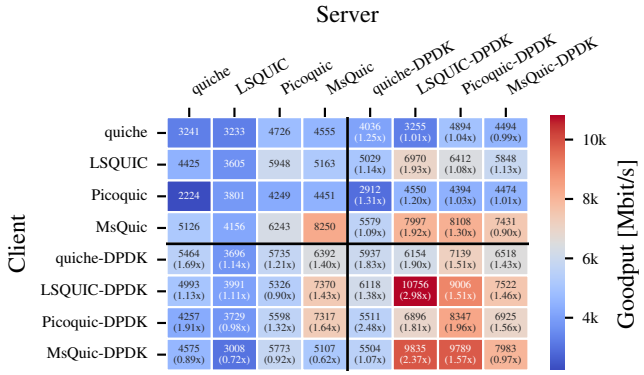


Fig. 2: Goodput matrix for all combinations of client and server implementations, with and without DPDK. For the combinations involving DPDK, the speedup relative to the same combination without DPDK is given.

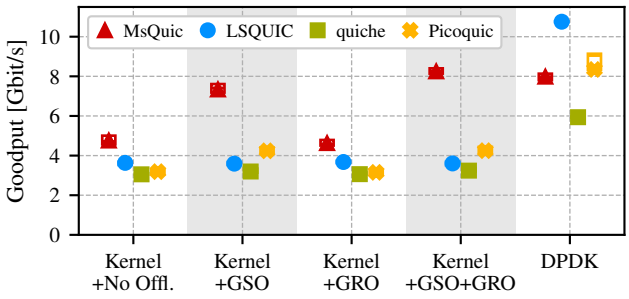
Some functions, like the main packet loop, do no useful work themselves and are excluded from analysis. Others, such as the DPDK receive function, perform work only in some iterations and cannot be fully excluded, as this would omit packet-processing costs. To quantify polling overhead, we instrument our implementations by measuring the time for each invocation using the CPU timestamping counter; afterwards, we determine whether meaningful work occurred. Following Rydén and Näslund [27], we accumulate time spent on useful work and polling to compute a *polling ratio*, which scales the corresponding samples. Instrumentation overhead and interface setup times are also excluded for our analysis.

We increased the UDP receive buffer 32-fold (from 208 KiB) to prevent client-side packet loss [4] and, due to sporadic send buffer errors observed in our measurements, also increased the send buffer by the same factor. Server and client hardware/software specifications and the DPDK parameters for all implementations are listed in the Appendix Tables I and II.

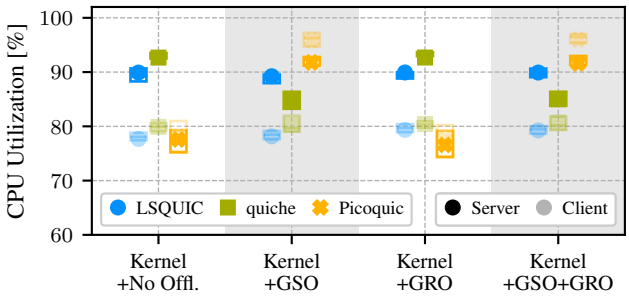
Integrating DPDK with SR-IOV Typically, a DPDK application takes full control of the physical NIC port, which can be undesirable when deploying a QUIC application alongside other server functions. Thus, we explore sharing a single physical port with the kernel using SR-IOV virtual functions by assigning only a virtual function to DPDK, leaving the physical port bound to the kernel driver. No changes were needed for our DPDK QUIC implementations, as the DPDK drivers abstract most hardware differences.

VI. Evaluation

Using the measurement framework, we evaluate the performance of four different QUIC stacks, *MsQuic*, *LSQUIC*, *quiche*, and *picoquic*, as well as their DPDK versions (see Section IV). In the following, we often refer to those stacks simply as *the four implementations*. We performed measurements in different settings regarding kernel bypass and segmentation offload. All goodput- and CPU utilization-related data was obtained from 8 GiB file downloads over 50 repetitions for repeatability. The data for the `perf` samples in Section VI-B



(a) Goodput.



(b) CPU utilization. The y axis starts at 60 % for readability.

Fig. 3: Goodput and CPU utilization of the QUIC stacks across optimization variants. CPU data is excluded for DPDK (active polling) and *MsQuic* (multithreading), as in both cases, the `pidstat` logs are not meaningful.

was collected with 1 GiB file size and 10 repetitions; the variance for these measurements was within reasonable limits. Any boxplots in this section depict the median as a horizontal line, the mean as an icon like ▲, and the quartiles Q_1 & Q_3 as the limits of the bounding boxes. The goodput matrix and all barplots show the mean values of the respective data points.

A. Implementation Matrix

In a first step, we investigate the goodput for all combinations of client and server implementations, both with and without DPDK. The results are shown in Figure 2. It is apparent that even though all four implementations are rather performant in their baseline without DPDK, there are significant differences amongst the combinations, spanning a factor of up to 3.7. The *MsQuic* client is the fastest of the four, with a mean goodput of up to 8.3 Gbit/s in combination with the *MsQuic* server. *quiche* is the slowest server in this comparison, reaching only 2.2 Gbit/s together with the *picoquic* client. When using the DPDK versions for both client and server, all combinations increase significantly in goodput, with speedups between 1.8× and 3×. The only exception to this is *MsQuic*, which is slightly slower with DPDK, reaching a goodput of 8.0 Gbit/s. *picoquic-DPDK* performs especially well as a server, achieving between 7.1 Gbit/s and 9.8 Gbit/s, while *LSQUIC-DPDK* and *MsQuic-DPDK* are the best-performing client implementations. The combination with the highest performance overall is *LSQUIC-DPDK* on both sides, with a mean goodput of 10.8 Gbit/s. Using DPDK only for the client or server also improves performance in most cases. This especially holds for *MsQuic*, where the standard client achieves between 5.6 Gbit/s and 8.1 Gbit/s with the DPDK servers, and the standard server reaches between 5.1 Gbit/s and 7.4 Gbit/s with the DPDK clients. Only the *MsQuic-DPDK* client performs rather bad with the non-DPDK servers, always falling behind its non-DPDK counterpart.

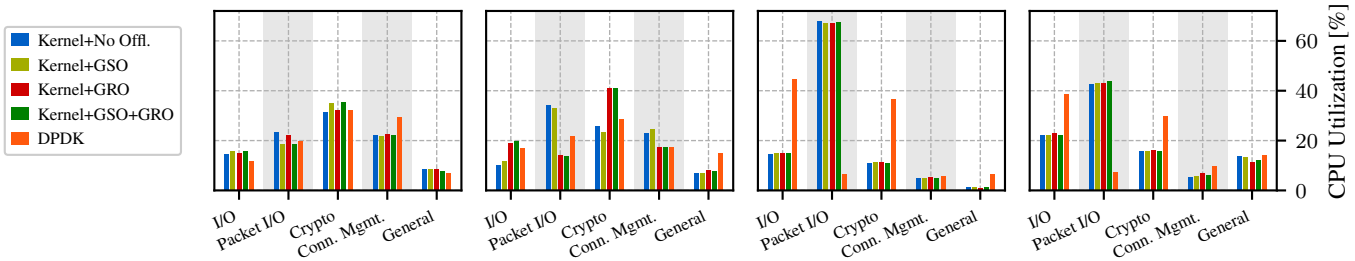
B. Comparison of Segmentation Offload and Kernel Bypass

As described in Section II, segmentation offload can be used to reduce packet I/O overhead, but it has to be supported by the implementation. By default, GSO and GRO are activated in the Linux kernel. To evaluate the effect of those optimizations on the four QUIC stacks, we disabled them using the `ethtool` utility. Figure 3a depicts the goodput of the four implementations for different combinations of GSO and GRO, as well as for the DPDK versions. Figure 3b shows the CPU utilization of the server and client machines, as reported by the `pidstat` utility. When comparing the server and client utilization, we get an indication of whether the sending or receiving side is limiting the performance of the connection. However, this only works for single-threaded applications and when there is no active polling. Otherwise, the CPU utilization reported by `pidstat` does in general not allow to determine the limiting side.

The results show that *MsQuic* benefits significantly from offloading, with a speedup of 1.7× between deactivating and activating both GSO and GRO. *MsQuic* is also the only implementation amongst the four that supports GRO. Activating GRO alone does not yield a noticeable impact on performance, likely due to being server-limited at this point. However, when transitioning from GSO alone—where *MsQuic* appears to be client-limited—to the combination of GSO and GRO, the goodput increases by 12 %. *LSQUIC* does not support GSO or GRO and is thus not affected by deactivating them regarding goodput and CPU utilization. Since *quiche* and *picoquic* both only support GSO, activating GRO does not have any impact. For *quiche*, we mainly observe reduced server CPU utilization with GSO but similar goodput, while *picoquic* has a visible goodput improvement by 33 %. *picoquic*'s CPU utilization increases with GSO for both server and client, with a higher increase at the client. This indicates that the receiving side is now limiting, and that including support for GRO into *picoquic* could further improve goodput.

When comparing the speedup of segmentation offload and kernel bypass with DPDK for the four implementations, it is apparent that in most cases, DPDK yields a substantially higher increase in goodput. The most prominent example here is *LSQUIC*, with a speedup of 3×. However, it is worth noting that *LSQUIC* does not utilize GSO or GRO, and that for *MsQuic*, segmentation offload slightly outperforms its DPDK version. This indicates that, if applied correctly, both segmentation offload and kernel bypass help to tackle the packet I/O bottleneck of QUIC.

To study the performance impact of the different optimizations, we profile their execution using `perf`, as described



(a) *MsQuic* server. (b) *MsQuic* client. (c) *LSQUIC* server. (d) *LSQUIC* client.

Fig. 4: Distribution of CPU utilization between different functional categories.

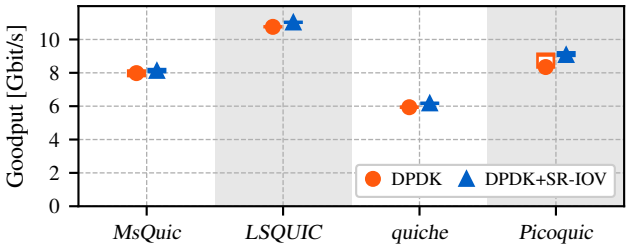


Fig. 5: Goodput of four DPDK implementations with and without SR-IOV.

in Section V. The results of this analysis for *MsQuic* and *LSQUIC* are shown in Figure 4. For *MsQuic* without offloading, we observe a high share of cryptographic operations, constituting 32 % of the samples at the server. Packet I/O, on the other hand, poses 24 % at the server and 34 % at the client. Packet I/O at the server decreases by 21 % with GSO and even more significantly with GRO at the client by 58 %. With DPDK, server packet I/O is very similar to GSO+GRO, while at the client, packet I/O is slightly higher compared to offloading. As mentioned in Section IV, *MsQuic-DPDK* uses a ring buffer for storing packets to be sent, and only single packets are enqueued at once. If the dequeuing happens fast enough, which we expect to happen in the active polling loop, packets are not sent in bursts, but only one at a time, which negatively impacts performance and hinders *MsQuic-DPDK* from achieving a higher goodput.

As explained earlier, *LSQUIC* does not support GSO or GRO. Therefore, no differences in the CPU utilization can be noted between the different configurations. For DPDK, we see a drastic decrease in packet I/O for both the server ($-90\,\%$) and the client ($-84\,\%$). Apparently, the buffering model in *LSQUIC-DPDK* is achieving very efficient packet I/O, which explains the high goodput of 10.8 Gbit/s. In turn, the shares of cryptography and file I/O increase significantly, indicating that those pose the new bottlenecks after tackling packet I/O.

Note that the data for *quiche* and *picoquic* is provided in Figure 6 in the Appendix, but not discussed in the paper. However, the behavior regarding segmentation offload (if supported) and kernel bypass is similar.

C. Performance of DPDK with SR-IOV

Finally, we investigate the effect on performance when using DPDK together with SR-IOV. As mentioned in Section II, this setup allows to use DPDK without requiring exclusive access to a NIC. In Figure 5, we compare the goodput achieved by the four DPDK QUIC implementations in both settings, *i.e.*, using plain DPDK and DPDK with SR-IOV. While there are no substantial differences between the two, we do notice a slight increase in goodput for the SR-IOV variant across all four implementations. This effect is largest with *picoquic-DPDK*, where goodput increases by 9 %. In addition, the performance is more stable across measurement runs. The only notable difference between the two settings lies in the NIC drivers used by DPDK: plain DPDK uses the *ice* driver for the physical function, while SR-IOV uses the *iavf* driver for virtual functions. We attribute the improvement to more efficient send/receive routines in *iavf*. Note that *iavf* operates only with virtual NICs and requires SR-IOV to be enabled.

VII. Conclusions

In this paper, we investigated the effect of kernel bypass with DPDK on the performance of QUIC. For that, we integrated DPDK into the three QUIC stacks *MsQuic*, *LSQUIC*, and *quiche*, and we updated the existing *picoquic-DPDK* introduced by Tyunyayev et al. [8]. We showed that while including DPDK has great potential to tackle the packet I/O bottleneck and increase goodput, it is not a universal solution for QUIC performance. This has several reasons: (1) By default, DPDK requires full NIC access, making it impractical, especially for the client side. (2) Integrating DPDK into QUIC libraries requires custom modifications, with speedups highly depending on both the original stack design decisions and the DPDK data path. (3) GSO and GRO can offer similar performance, while being less intrusive and easier to integrate.

Using SR-IOV virtual functions, we showed that QUIC with DPDK can also be deployed without exclusive NIC access but with similar or even better performance. We found that both kernel bypass and segmentation offload tend to increase burstiness of the produced traffic (see Appendix Section D), which has to be considered for multi-hop scenarios. Integrating DPDK into QUIC stacks for performance improvement might be especially promising for CDNs, as they operate in controlled environments. Popular CDNs already use user-space networking for other high-performance services [28], [29]. In summary, our evaluation provides key insights into kernel bypass and segmentation offload in QUIC, supporting ongoing development and effective use across diverse scenarios.

Acknowledgments

This work was funded by EU Horizon Europe, project GreenDIGIT (101131207), the German Research Foundation (DFG), projects HyperNIC (503359370) and SLICES-Sustainability (566292327), the Bavarian Ministry of Economic Affairs, Regional Development and Energy, project 6G Future Lab Bavaria, and the German Federal Ministry of Research, Technology and Space, project 6G-life (16KIS2414). We acknowledge Luca Otting's work on *LSQUIC-DPDK*.

References

[1] S. Metzmacher, *net: define IPPROTO_QUIC and SOL_QUIC constants*. Accessed: Feb. 8, 2026. [Online]. Available: https://lwn.net/ml/all/5d5ac074-1790-410e-acf9-0e559cb7eacb@samba.org/

[2] N. Pyle. "SMB over QUIC now available in Windows Server Insider Datacenter and Standard editions. "[Online]. Available: https://techcommunity.microsoft.com/blog/filecab/smb-over-quic-now-available-in-windows-server-insider-datacenter-and-standard-ed/3975242

[3] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle, "QUIC on the Highway: Evaluating Performance on High-rate Links," in *IFIP Networking Conference (Networking)*, 2023.

[4] M. Kempf, B. Jaeger, J. Zirngibl, K. Ploch, and G. Carle, "QUIC on the Fast Lane: Extending Performance Evaluations on High-rate Links," *Computer Communications*, 2024. DOI: 10.1016/j.comcom.2024.04.038

[5] 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*, 2020. DOI: 10.1145/3405796.3405827

[6] K. Ananyev. "Minutes of DPDK Technical Board Meeting. "[Online]. Available: https://mails.dpdk.org/archives/dev/2024-February/286910.html

[7] M. König, O. P. Waldhorst, and M. Zitterbart, "QUIC(k) Enough in the Long Run? Sustained Throughput Performance of QUIC Implementations," in *IEEE 48th Conference on Local Computer Networks (LCN)*, 2023.

[8] N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette, "A High-Speed QUIC Implementation," in *Proceedings of the 3rd International CoNEXT Student Workshop*, 2022, pp. 20–22.

[9] A. Ghedini. "Accelerating UDP packet transmission for QUIC. "[Online]. Available: https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/

[10] X. Zhang et al., "QUIC is not Quick Enough over Fast Internet," in *Proceedings of the ACM Web Conference*, 2024. DOI: 10.1145/3589334.3645323

[11] L. Torvalds, *Development kernel 2.5.33 released*. Accessed: Feb. 8, 2026. [Online]. Available: https://lwn.net/Articles/8930/

[12] W. De Bruijn and E. Dumazet, "Optimizing UDP for content delivery: GSO, pacing and zerocopy," in *Linux Plumbers Conference*, 2018.

[13] W. De Bruijn, *Re: Question About Using UDP GSO in Linux Kernel 4.19*. Accessed: Feb. 8, 2026. [Online]. Available: https://www.spinics.net/lists/netdev/msg672086.html

[14] Linux Kernel Development Community, *Linux Kernel Documentation – Segmentation Offloads*. Accessed: Feb. 8, 2026. [Online]. Available: https://docs.kernel.org/networking/segmentation-offloads.html

[15] M. Mirosław, *Netdev Features Mess and How to Get Out From It Alive*. Accessed: Feb. 8, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/netdev-features.html

[16] LF Projects, LLC, *DPDK*. Accessed: Feb. 8, 2026. [Online]. Available: https://www.dpdk.org

[17] P. Megyesi, Z. Krämer, and S. Molnár, "How quick is QUIC?" In *Proc. IEEE ICC*, 2016. DOI: 10.1109/ICC.2016.7510788

[18] 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*, 2019.

[19] A. Yu and T. A. Benson, "Dissecting Performance of Production QUIC," in *Proceedings of the Web Conference 2021*, 2021. DOI: 10.1145/3442381.3450103

[20] 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*, 2022.

[21] M. König, S. Rust, M. Zitterbart, and B. Scheuermann, "Examining the Heterogeneous Throughput Performance Landscape of QUIC Implementations," in *IFIP Networking Conference (Networking)*, 2025.

[22] Microsoft Corporation, *MsQuic Documentation*. Accessed: Feb. 8, 2026. [Online]. Available: https://github.com/microsoft/msquic/tree/v2.4.8/docs

[23] Microsoft Corporation and N. Banks, *datapath_raw_dpdk.c*. Accessed: Feb. 8, 2026. [Online]. Available: https://github.com/microsoft/msquic/blob/v2.4.8/src/platform/datapath_raw_dpdk.c

[24] Cloudflare, Inc., *Enjoy a slice of QUIC, and Rust!* Accessed: Feb. 8, 2026. [Online]. Available: https://blog.cloudflare.com/enjoy-a-slice-of-quic-and-rust/

[25] tokio-rs. "Mio – Metal I/O. "[Online]. Available: https://github.com/tokio-rs/mio

[26] M. Kempf, S. Tietz, B. Jaeger, J. Späth, G. Carle, and J. Zirngibl, "QUIC Steps: Evaluating Pacing Strategies in QUIC Implementations," *Proc. ACM Netw.*, 2025. DOI: 10.1145/3730985

[27] A. Rydén and E. Näslund, *Estimate CPU Utilization for Data Processing with DPDK*.

[28] L. Zhu et al., "Deploying user-space TCP at cloud scale with LUNA," in *USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, ISBN: 978-1-939133-35-9.

[29] Tencent Cloud. "F-Stack. "[Online]. Available: https://www.f-stack.org/

APPENDIX

A. Hardware and DPDK Parameters

Table I provides an overview of the hardware used for our measurements. To improve comparability between the effects of DPDK on the considered QUIC implementations, we configured identical DPDK parameters for all four tested implementations. The configured values are listed in Table II.

TABLE I: Hardware and software specifications of the server and client.

OS	Debian 12 Linux 6.1.0-17
CPU	Intel Xeon Gold 6421N (32 cores)
RAM	512 GiB (DDR5)
NIC	Intel E810-CQDA2 (100 Gbit/s)
DPDK	v24.11

TABLE II: DPDK parameters used for all implementations.

Parameter	Value
Bursts (TX, RX)	32
Mbuf count	16383
RX/TX descriptors	4096
Mbuf cache	256
Tx ring size	4096

B. Artifacts

The source code of the four QUIC DPDK implementations as well as our measurement framework introduced in Section V is published on GitHub:

https://github.com/tumi8/quic-bypass-paper

Furthermore, we include the configurations for all measurements. These artifacts can be used for reproducing, extending, or improving the measurements presented in this work. In addition, we provide all measurement results presented in this work, including detailed logs and packet captures.

C. Perf Analysis for quiche and picoquic

Figure 6 shows the distribution of CPU utilization between different functional categories for the *quiche* and *picoquic* libraries. These plots are obtained using the same setup used for Figure 4 in Section VI-B.

D. Effects of Offloading and Kernel Bypass on Burstiness

As explained in Section IV, *LSQUIC*, *quiche*, and *picoquic* process packets in bursts in their DPDK version. To evaluate the effect of this on the generated traffic, we capture the packets on the wire using an optical splitter, as described in Section V, and investigate the spacing between them. Figure 7 shows the CDF of the inter-arrival times at the capture node for the different implementations, both with and without DPDK. The data for this evaluation was collected from 1 GiB file transfers across 10 repetitions; the variance for these measurements was within reasonable limits. As expected, *LSQUIC*, *quiche*, and *picoquic* produce traffic with more bursts when using DPDK. This effect is especially significant for *LSQUIC*, where without DPDK, 50 % of the packets are spaced by at least 2.9 µs, while with DPDK, 99 % of the packets are sent back-to-back with only the serialization time in between them. The only exception is *MsQuic*, which is less bursty in its DPDK version compared to kernel networking. Explanations for that are the efficient usage of segmentation offload in this implementation, as well as the inefficient use of DPDK with single packets being processed at a time. Implementations that use GSO, *i.e.*, *MsQuic*, *quiche*, and *picoquic*, cause burstier traffic than *LSQUIC*, which doesn't use GSO. This evaluation shows that both segmentation offload and kernel bypass with DPDK can lead to bursty traffic, which has to be taken into account for multi-hop scenarios.

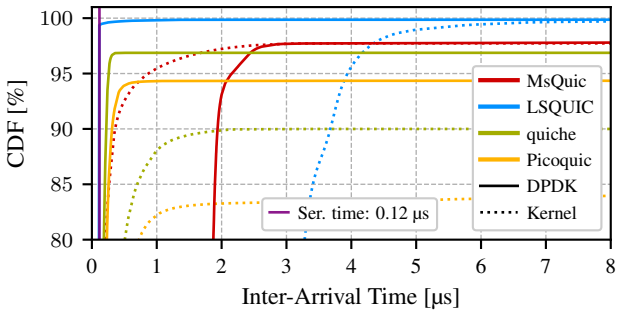
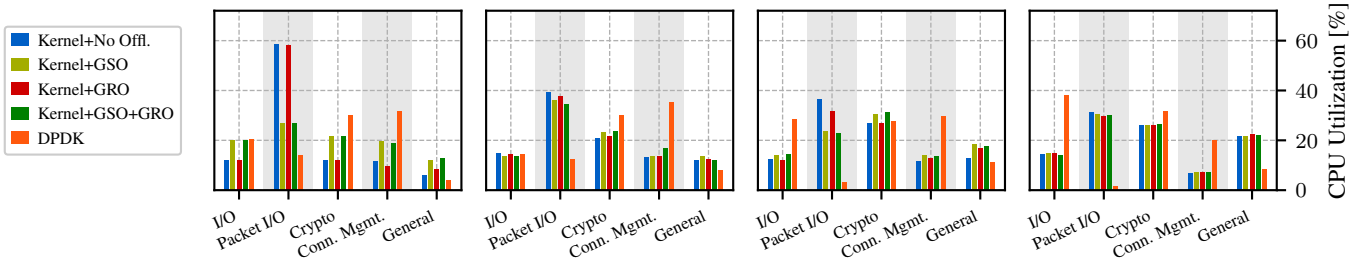


Fig. 7: CDF of the inter-arrival times ($\leq$ 8 µs) of packets observed at the capture node.



(a) *quiche* server. (b) *quiche* client. (c) *Picoquic* server. (d) *Picoquic* client.

Fig. 6: Distribution of CPU utilization between different functional categories.