Understanding QUIC's Throughput Speedbumps

Saubhik Mukherjee, Demi Lei, Mostafa Ammar and Ahmed Saeed
Georgia Institute of Technology

Abstract

QUIC is experiencing phenomenal success as a low-latency, secure, and flexible protocol, adopted and deployed by multiple hyperscale content providers. However, the throughput of all its implementations remains remarkably low compared to that of TCP + TLS. Approaches to accelerating QUIC's throughput remain scattered, applied to a single implementation at a time, and one approach at a time. In this paper, we systematically identify the different bottlenecks that arise in different implementations. We use our findings to propose an application-agnostic architecture to accelerate single-connection throughput in QUIC. We demonstrate the value of our proposal by accelerating two implementations: `quicly` and `mvfst`, improving their throughput by 1.36× and 1.88× respectively, compared to their baseline implementations.

CCS Concepts

• **Networks** → **Network protocol design**.

Keywords

QUIC, Performance Optimization

ACM Reference Format:
Saubhik Mukherjee, Demi Lei, Mostafa Ammar and Ahmed Saeed. 2025. Understanding QUIC's Throughput Speedbumps. In *Applied Networking Research Workshop (ANRW '25), July 22, 2025, Madrid, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3744200.3744780

1 Introduction

QUIC has become one of the main standards for web applications [32], with a wide range of implementations that vary in complexity and performance [10]. This flexibility in designing the implementation of the protocol allows different stakeholders to innovate independently and customize their implementations for their use cases. The diversity in QUIC implementations is more conspicuous compared to other transport protocols because QUIC is fully implemented in userspace, making it easier to develop new independent implementations compared to TCP which requires kernel changes. This flexibility comes at the expense of requiring each implementation to attempt to reinvent the wheel of performance optimization and tweaking. In addition, it complicates efforts to methodically examine QUIC-specific performance bottlenecks. In fact, existing work on improving the performance of QUIC focuses on a single implementation and one or two optimizations [11, 15, 27, 29]. For example, `MsQuic` and `picoQUIC-DPDK` demonstrate the benefits of bypassing the kernel's UDP implementation [11, 27]. Further, MsQuic pipelines protocol processing and socket-based IO using two separate threads [11]. In contrast, TCP upgrades have been slow but cumulative with a broad immediate impact on application performance (e.g., [14]).

To reduce wasted effort, we believe that design guidelines and best practices should be clearly defined for QUIC implementations. For example, guidelines could be defined to minimize CPU requirements, while others focus on specific per-connection metrics like latency or throughput. Such an approach would also allow users to pick the implementation that most suits their use cases based on the guidelines it follows. To that end, this paper demonstrates the feasibility of implementation-agnostic improvements in QUIC performance, focusing on *single-connection throughput* defined as the throughput achieved for the packets communicated under a specific QUIC connection ID. This focus stems from the surprisingly high variability in single-connection throughput between different QUIC implementations, making it a good use case for our objective. Moreover, we anticipate that as the popularity of QUIC grows, some of its applications will require achieving high single-connection throughput (e.g., file transfer between datacenters and AR/VR applications).

We examine five QUIC implementations and identify classes of fundamental throughput bottlenecks, herein referred to as speedbumps. First, we show that some implementations face poor single-connection performance due to their own inefficient implementation. Then, we show that Linux's UDP stack is seldom a throughput bottleneck. Finally, the choice of the crypto library can have a significant impact on performance. Not only that, crypto represents a fundamental limitation on the performance of all QUIC implementations.

We leverage our analysis to propose implementation-agnostic guidelines for designing high-throughput QUIC implementations. Our proposal improves the throughput of a QUIC implementation through pipeline parallelism. In particular, packets from the same connection are processed in parallel by breaking stack processing into the following five stages:

ANRW '25, Madrid, Spain

ACM ISBN 979-8-4007-2009-3/2025/07
https://doi.org/10.1145/3744200.3744780

(i) IO, *(ii)* UDP/IP processing, *(iii)* QUIC protocol handling (without crypto), and *(iv)* crypto. Each stage runs in its own lightweight user threads that communicate asynchronously through shared memory. To demonstrate the value of our proposal, we modify the implementation of two popular QUIC implementations, `quicly` [8] and `mvfst` [6], improving their performance by 1.36× and 1.88×, respectively.

A software-only approach. We focus on software-only implementations of QUIC. We realize that offloading encryption and decryption operations to hardware can dramatically improve throughput. However, it is important to note that NICs that support such offloads are costly and not widely deployed. Considering QUIC's growing popularity as a standard Internet transport, we find it critical to develop performant and portable software-only architectures, maintaining the portability of current stacks and enabling more users to receive the benefits of high-throughput QUIC implementations.

2 QUIC's Throughput Speedbumps

2.1 Overall Performance

We measure the throughput of five different QUIC implementations: `mvfst`, `picoquic`, `quicly`, `picoquic-dpdk`, and `MsQuic`. To measure the throughput of each implementation, we use an `iperf`-like tool that continuously attempts to send data from a server to a client. All evaluation is done with Generic Segmentation Offload (GSO) enabled on Linux machines with 25 Gbps NICs. All implementations use `AES128_GCM` encryption using OpenSSL.

Figure 1 compares the overall throughput achieved by the five implementations. We compare them to TCPLS, an optimized implementation of TCP+TLS to provide similar functionality to QUIC but using TCP while achieving high throughput [25]. `picoquic-dpdk` and `MsQuic` provide comparably good performance out of the box, having been optimized specifically for high throughput. `picoquic-dpdk` implements `picoquic` on top of a bare-bone implementation of a UDP stack using DPDK [27]. `MsQuic` pipelines QUIC protocol processing and socket-based IO using two separate threads [11]. Both implementations rely on batching to amortize the cost of different stack operations. However, both implementations achieve less than 60% of the TCPLS throughput: `picoquic-dpdk` achieves 58% and `MsQuic` achieves 53%.

To better understand how CPU cycles are spent in QUIC implementations, we profile three QUIC stacks: `quicly`, `mvfst`, and `picoquic`. We chose `quicly` and `mvfst` because they represent two poles of QUIC implementations. `quicly` is lightweight, while `mvfst` has many complex features. `picoquic` and `MsQuic` optimize throughput. We focus on `picoquic` because it offers a convenient API for changing the underlying network stack and encryption library in a Linux environment. `MsQuic` supports the same features. However, `MsQuic` optimizations, including Schannel, are exclusive to Windows, complicating our comparisons.

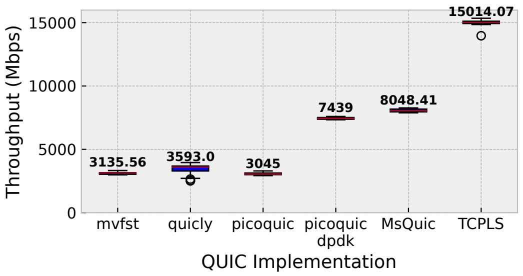

Figure 1: Throughput of different QUIC implementations. Numbers reported here are consistent with those publicly reported by the `MsQuic` team [5]. The discrepancy between numbers reported here for `picoquic-dpdk` and those reported in [27] are due to using different crypto engines.

We use `perf` to record the CPU cycles spent by each function in each of the implementations. Table 1 shows a breakdown of the CPU cost for each component of a QUIC implementation. Our component breakdown is based on distinct objects in an implementation's binary as identified by `perf`. In particular, we focus on three main components: *(i)* QUIC protocol handling (without crypto), *(ii)* IO, representing the network stack from calls at the driver level to the socket API, and *(iii)* crypto. Due to our focus on QUIC, we ignore the cost of the application deployed on top of QUIC (e.g., web server or file server). Further, we ignore other libraries used in a QUIC implementation whose combined CPU cost is less than 5% of the total CPU consumed by the stack, most of which is consumed by libc.

QUIC implementation is a speedbump. One throughput speedbump is self-evident: the implementation of the QUIC protocol handling itself, excluding crypto. Indeed, in most of our discussions with practitioners developing some of these stacks, high throughput is mentioned as a non-goal because current QUIC use cases require a few hundred Mbps in the worst case. However, as discussed earlier, the phenomenal success achieved by QUIC is likely to encourage its implementation in even more use cases (e.g., large file transfers between datacenters or AR/VR gaming), motivating some practitioners to look for ways to improve the throughput of QUIC [11, 27]. In the data reported in Table 1, `mvfst` stands out as an implementation where the QUIC protocol handling components consume the majority of the CPU cycles. Next, we examine the other two speedbumps.

2.2 IO and UDP/IP IO

A common notion is that the performance of QUIC is hampered by inefficient implementations of UDP in modern operating systems [25, 27, 29, 31]. The idea is that, while the TCP stack has several implementation optimizations and common NIC offloads, UDP lacks either. The implication is that a high-throughput stack is required to overcome that problem. For

	Linux					**Kernel-bypass**	
	quicly (no opt.)	**quicly**	**mvfst**	**picoquic**	**picoquic (Fusion)**	**picoquic**	**picoquic (Fusion)**
IO	60.7%	21.3%	12.3%	19.9%	26.3%	56.3%	53.1%
Crypto	25.3%	55.9%	18.9%	43.2%	34.7%	20.5%	19.7%
Protocol	8.5%	14.2%	52.7%	26.5%	35.1%	16.7%	23.96%
libc	4.7%	8.5%	14.1%	9.3%	3.4%	3.8%	1.8%
Throughput (Gbps)	3.5	7.2	3.1	3.0	3.2	7.4	10.4

Table 1: Breakdown of the CPU cycle distribution of three QUIC sender implementations. The breakdown is based on the shared object identified by `perf`. OpenSSL is the crypto engine used unless otherwise specified. When Fusion is used for crypto, its CPU cost is identified by the cost of `ptls_fusion*` functions. Throughput values are similar to those in figures 1, 2, and 3.

Implementation Variant	UDP (baseline)	UDP (+ GSO)	UDP (+ GSO & GRO)	TCP
Throughput (Gbps)	4.33	14.5	21.4	23.5

Table 2: Throughput of UDP using iPerf3 for different configurations of UDP with TCP throughput as a reference. Results in red represent tuned implementations.

example, `MsQuic`'s implementation relies on Windows XDP and `picoquic-dpdk` uses DPDK to bypass that overhead. As mentioned earlier, we focus on Linux-based optimizations. However, kernel bypass introduces significant CPU overhead due to busy-polling (cf. `picoquic-dpdk` in Table 1), requiring careful handling to ensure scalability. The UDP Linux stack has seen significant optimization [16] which has been widely demonstrated [18, 23]. We conduct some experiments to find the maximum achievable UDP throughput.

UDP throughput is highly dependent on the optimizations implemented by an application both at the sender and the receiver, requiring more complicated code to achieve higher throughput, unlike TCP, where high throughput can be achieved using the normal socket API. For example, the upstream implementation of `iperf3` has a maximum UDP throughput of 4.13 Gbps. We modified the implementation of `iperf3`, incrementally adding Generic Segmentation Offload (GSO) support at the sender and Generic Receiver Offload (GRO) support at the receiver, comparing their combined performance to the throughput of TCP. GSO and GRO enable batch processing of transmitted and received packets, respectively. The operation is still performed in software, but requires fewer system calls to transmit the same amount of data.[1] Figure 2 shows a comparison between the different implementations of `iperf`.

There are two main observations. First, achieving high throughput requires optimizations at both the sender and the receiver. This is a persistent theme in QUIC stack optimization as we will discuss later when dealing with crypto. Second, the throughput of the UDP stack is much higher than that achieved by many QUIC stacks, comparing Table 2 and Figure 1. To better highlight the impact, consider the comparison between the two `quicly` implementations in Table 1: `quicly` (no opt.) and `quicly`. Both use GSO at the server. The main difference between the two implementations is the use of GRO at the client in the latter, allowing it to achieve 2× the throughput reported in Figure 1 (i.e., 7.2 Gbps). More importantly, the performance of `quicly` with GSO and GRO is only 2% lower than the performance achieved by `picoquic-dpdk` which requires kernel bypass. Moreover, using GSO and GRO reduces the IO cost at the server from 60.7% to 21.3% of the total CPU consumed by the stack, shifting the bottleneck from IO to crypto.

Unit	Overhead of Encryption		Total Overhead of Decryption
	Total	Initialization	
1500B × 6	5100ns	1668ns	4128ns
3000B × 3	3741ns	839ns	3229ns
4500B × 2	3147ns	570ns	2792ns
9000B × 1	2578ns	281ns	2466ns

Table 3: Latency of encryption and decryption in nanoseconds for different units of encryption, representing different MTUs, keeping the total bytes encrypted at 9000 bytes.

2.3 Crypto

The other source of overhead extraneous to a QUIC implementation is the overhead of the crypto library (e.g., OpenSSL [7], Fusion [4], or Schannel [9]). Optimizing IO shifts the majority of the CPU overhead to crypto for most implementations. Changing the behavior of the crypto library or its overhead can have a dramatic effect on the performance of a QUIC implementation. For example, consider the impact of replacing OpenSSL with Fusion in `picoquic`. Although it does not improve the throughput of `picoquic` on Linux by much, it improves `picoquic-dpdk`'s throughput by 40%, allowing it to achieve 10.4 Gbps in the median. Earlier work has also identified crypto as a QUIC bottleneck [25, 26]. [25] mentions that "QUIC packets are smaller units than TLS records for encryption and decryption." Our goal is to better understand this overhead, including the impact of increasing the MTU.

We use Google's benchmark library to evaluate the latency of the encryption function call [1]. In particular, we evaluate the latency of encrypting 9000 bytes of data, passed to the encryption library in different unit sizes, starting from 1500-byte MTU to 9000-byte jumbo frames. The time for encrypting a batched set of six 1500-byte packets is 5100 ns, leading to a maximum encryption throughput of 14.1 Gbps. Increasing the encryption unit size improves throughput, reaching 27.9 Gbps for 9000-byte jumbo frames, representing a fixed upper limit on QUIC's throughput, lower than that available in TCP + TLS stacks. The difference stems from the fact that QUIC applies crypto on a per-MTU-packet basis. Thus, the

[1] TCP GSO and GRO are typically offloaded to the NIC.

cost of encryption cannot be further amortized by increasing the encryption unit size beyond jumbo frames. TLS operates on top of TCP and can have larger record sizes (i.e., units of encryption), enabling higher throughput.

Ideally, the cost of encrypting 9000 bytes should be the same regardless of the encryption unit size, aside from minor overheads caused by the additional function calls. We investigated the difference between operating on 1500-byte MTUs and 9000-byte MTUs. We found that the major source of added overhead is the overhead of cipher initialization, which is done per packet. For example, when using OpenSSL and AEAD ciphers, we benchmark the overhead of `aead_do_encrypt_init` and `ptls_cipher_init`, shown in Table 3 as initialization latency. The difference between the initialization latency for the 9000-byte case and the 1500-byte case accounts for 54% of the difference in overall encryption latency. This overhead is hard to overcome as the initialization depends on packet-specific attributes. For example, `aead_do_encrypt_init` uses a packet's sequence number. Decryption has an overhead comparable to encryption, limiting receiver throughput. Thus, we find that simply optimizing the server by improving encryption performance yields modest throughput improvements but pushes the bottleneck to the client (i.e., limited by decryption throughput). Improving overall throughput requires symmetrical optimization of the encryption and decryption paths for a connection (i.e., optimizing the implementation of both client and server).

Crypto as a fundamental QUIC speedbump. The throughput of the crypto engine is limited by its fixed per-packet overhead. While different crypto engines yield different performances, the overhead per packet (e.g., initialization, memory operations, function calls, etc) forms a fundamental roadblock. Increasing the unit size of crypto is impossible in the current QUIC standard as crypto is done per packet.

3 Design Guidelines

We use our observations to develop implementation-agnostic design guidelines to maximize single-connection throughput of QUIC. Our design guidelines are centered on the basic idea of fine-grain pipeline parallelism of packet processing. In particular, we separate the busy-polling IO from UDP/IP processing into their own stages. In addition, we separate the handling of the QUIC protocol from the crypto processing. This fine-grain definition of pipeline stages implies increased communication between threads handling each stage. To minimize the overhead created by this design, we leverage ideas developed for low-latency data center network stacks. In particular, we use lightweight user threads that are cheap to preempt and that can voluntarily yield at low cost, allowing the threads to communicate over pinned shared memory.

Pipelining. We define the pipeline stages as follows:

- *IO:* This stage is responsible for interrupt handling or busy polling to fetch packets from the NIC or transmit packets through the NIC. The most basic form of pipelining separates this stage on a dedicated core, while having all other stages processed serially on a single stage. Such separation is feasible in both Linux-based stacks and kernel-bypass stacks.
- *UDP/IP:* This stage is responsible for network stack operations including the link layer, network layer, transport layer, and socket operations. The overhead of this stage can be reduced when using kernel-bypass stacks, by avoiding the overhead of systems calls and crossing into kernel space. Moreover, kernel-bypass stacks have even higher throughput than the Linux UDP stack.
- *Crypto:* This stage keeps track of all crypto metadata needed for the encryption and decryption of packets belonging to connections handled by this stage. The stage simply looks up crypto metadata for an incoming packet and invokes the crypto engine. To simplify the implementation of the crypto stage, we limit its role to the encryption and decryption of short-header packets (i.e., packets exchanged after connection establishment). This limitation also maintains the connection establishment latency of the original implementation.
- *QUIC Protocol Handling:* This stage is where all QUIC-specific logic is implemented. The only modification needed for implementation in order to fit into our architecture is to support crypto offload. This requires two minor modifications. First, on connection establishment, crypto secrets are communicated asynchronously with the Crypto stage. Second, rather than passing all packets to crypto processing, the implementation checks if a Crypto stage is active, and if it is, it handles the packet appropriately (i.e., tags the packet with necessary metadata for encryption or skips the decryption step).
- *Application:* This stage handles the logic of the application that uses QUIC for communication (e.g., web server logic).

Communication between stages. Inter-stage communication can be split into a data plane and a control plane. Communication in the data plane includes passing data packets between stages along with any metadata specific to that packet (e.g., connection information or a packet number). Control plane communication includes communicating crypto secrets between the QUIC Protocol Handling stage and the Crypto stage. All communication is performed using a lightweight RPC library based on shared memory. All messaging is lock-free, where data in both data-plane and control-plane messages are queued between stages and processed in a FIFO manner. Under some pipeline stage configurations, some stages might have higher throughput than their subsequent stages, potentially creating congestion inside the system. To avoid this problem, we create stalls by forcing the thread responsible for the higher-throughput stage to sleep when it detects that the length of the queue to the subsequent stage exceeds a configurable threshold.

Pipeline Configuration. We define a *Pipeline Configuration* as the order in which stages are executed and their parallelism configuration (i.e., which ones are executed in series and which are executed in parallel). The best-performing Pipeline Configuration is implementation-dependent. It depends on the nature of the bottleneck and the benefits added by parallel execution compared to the added communication cost. We manually identified the best Pipeline Configuration for the two implementations we studied. We leave automatically and dynamically adapting Pipeline Configuration to future work.

4 Implementation

Our implementation requires light-weight threads that support efficient inter-thread communication library, on top of an optimized network stack. While we can achieve these requirements on top of Linux, Shenango libOS provides a runtime that supports all these features out of the box [24]. In particular, Shenango offers a lightweight and efficient implementation of a UDP/IP stack and a lightweight threading library. Moreover, Shenango pipelines packet processing, dedicating a core to IO processing. This IOKernel passes packets to lightweight threads that handle UDP/IP and application processing. We extend that paradigm to separate packet processing into the stages discussed above.

We implement our design to accelerate two QUIC implementations: `mvfst` and `quicly`. As discussed earlier, we chose `mvfst` and `quicly` because they provide two opposite points in the QUIC implementation design space. `quicly` provides a very lightweight implementation where most of the CPU time is spent on IO and crypto. Further,`quicly` is already stack agnostic, making it trivial to create a separate pipeline for IO and UDP/IP processing. Moreover, `quicly` provides an API to offload crypto operations, making it easy to implement it in a separate thread. On the other hand, `mvfst`'s implementation has a high overhead, making it the main bottleneck, allowing us to demonstrate the value of pipelining. For both implementations, we use a baseline pipeline configuration, referred to as *Simple Pipelining*, which has only two stages, one for IO and one for all other packet processing operations, including UDP/IP, crypto, and protocol handling. Note that our simple pipelining configuration uses kernel bypass for handling IO. We manually identify the best pipeline configuration for each QUIC implementation, which we discuss next.

quicly. On the server side, `quicly` has only two stages: *(i)* IO and *(ii)* all other stages performed sequentially in a single thread. On the client side, it has three stages: *(i)* IO, *(ii)* UDP/IP and QUIC header parsing, and *(iii)* Crypto and QUIC Protocol Handling. All other configurations yielded poorer performance. We use `quicly`'s qperf application. Modifications included changes of socket calls and event processing. Moreover, at the client, we launch a dedicated thread for receiving and parsing the header of packets.

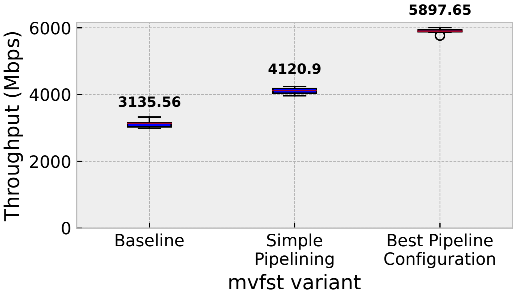

Figure 2: Throughput of `mvfst`.

mvfst. We created symmetrical pipelines for `mvfst`'s client and server. The first stage has IO and Crypto running on a busy-polling thread. The second stage has the UDP/IP and QUIC protocol handling. Modifying `mvfst` is much more involved than `quicly`. `mvfst` uses folly [3] as a wrapper around the socket API and fizz [2] TLS implementation. Our implementation makes modifications to `mvfst`, fizz, and folly. We use `mvfst`'s tperf benchmarking application.

5 Evaluation

This evaluation aims to answer the following questions, focusing on single-connection performance:

(i) How much benefit does each implementation gain from simply optimizing its use of Linux's UDP stack?
(ii) How much does pipelining improve performance? Can it be done incrementally?
(iii) How much performance improvement can we achieve by increasing the size of the unit of crypto?

Evaluation setup: We use two pairs of machines in Cloudlab [17]. The first pair are xl170 machines with an Intel E5-2640v4 CPU and two dual-port Mellanox ConnectX-4 25 Gbps NICs each. We use only a single port on a single NIC. We used this pair to produce all results reported earlier in this paper as well as all `mvfst` experiments in this section. To experiment with larger MTUs, we use the second pair of machines with an AMD 7302P CPU and two dual-port Mellanox ConnectX-5 25 Gbps NICs each. We use a single port. We use that pair for all `quicly` experiments in this section. `quicly` performed comparably in both setups. All experiments used Ubuntu 20.04 with Linux 5.4 kernel and DPDK 22.03. Our setup can support a maximum MTU size of 4000 bytes. All implementations use OpenSSL for crypto.

mvfst. Figure 2 shows the throughput of `mvfst` under different configurations. The baseline implementation of `mvfst` supports GSO and GRO, which are natively supported. Simple pipelining improves throughput by 31.4%. The moderate improvement demonstrates that the Linux stack is not a major

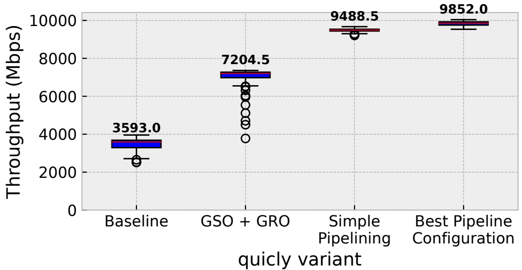

Figure 3: Throughput of `quicly`**.**

bottleneck for QUIC throughput. The best pipeline configuration improves throughput by an additional 43% (or 88% compared to the baseline). The best pipeline configuration also has only two stages. However, the IO stage also handles Crypto. Using the best pipeline configuration only on the server or client does not yield performance improvements compared to simple pipelining. One way to interpret these results is that `mvfst`, with the best pipeline configuration, achieves only 73% the throughput of `MSQuic` or `picoquic-dpdk`. However, our point is that a user that relies on `mvfst`-specific features can achieve 88% improvement in performance by using its best pipeline configuration.

quicly. Figure 3 shows the incremental improvement observed in `quicly`'s implementation. The baseline only uses GSO at the server. We optimize the client to support GRO, improving throughput by 2×. Like with `mvfst`, we find that a significant improvement in throughput typically requires changes at both the client and the server. Moreover, simple pipelining yields a modest improvement 31%, almost identical to the improvement achieved by simple pipelining in `mvfst`. However, the best pipeline configuration of `quicly` is only 3% better than the simple configuration. The best pipeline configuration of `quicly` is 32% and 22% better than `picoquic-dpdk` and `MSQuic`, receptively. `MSQuic` uses coarse-grain pipelining, limiting the possibility of performance improvements in Linux compared to the best configuration of `quicly`. On the other hand, `picoquic-dpdk` can be improved if it is pipelined. Note that the objective of this paper is not to exhaustively pipeline all implementations. Instead, our objective is to demonstrate the value of pipelining.

Impact of the crypto unit size. Table 1 shows that most of the CPU in `quicly`'s optimized implementation is spent on crypto. Thus, optimizing crypto is likely to have the biggest impact on `quicly`. To observe its impact, we changed the MTU size used in our experiments from 1500 bytes to 4000 bytes. Increasing the MTU size improves performance, achieving 14.6 Gbps under the simple pipelining configuration and 15.4 Gbps under the best pipeline configuration. This is the only configuration that allows us to outperform TCPLS, only marginally by 2%. This improvement comes at the cost of using additional CPU cores to pipeline QUIC processing. We predict that these improvements can be achieved by other implementations as well, but leave tuning them to future work.

6 Related Work

There has been interest in improving the throughput of QUIC using novel software architectures [11, 27] and hardware offloads [29]. It is also important to note that aside from the work on `picoquic-dpdk` [27], earlier work has dealt with older implementations of QUIC that yielded substantially lower throughput [25, 29] than the state of the art.There have been efforts to analyze the performance of multiple QUIC implementations, comparing them to TCP in different low-throughput settings [19, 22, 28, 30] and examining the impact of encryption acceleration in CPUs [21]. However, such settings are out of scope for this work because our main focus is improving single-connection throughput using a software-only implementation. Pipelining the stack over two stages is a pretty common tactic. TAS decouples application processing from network stack processing [20]. For QUIC, MsQuic has two stages, one for the application and protocol handling (including crypto), and the other for IO and UDP/IP [11, 12]. NetChannel proposes breaking the stack into finer-grained stages [13]. This work identifies the stages and demonstrates their value in different QUIC implementations.

7 Conclusion

Current QUIC implementations emphasize following the protocol standard while offering different levels of performance. Optimizing performance requires understanding the fundamental bottlenecks in the QUIC protocol and following best practices that mitigate their effect. This paper demonstrates the value of such guidelines, focusing on single-connection throughput. We find that the throughput of the crypto engine is limited by its fixed per-packet overhead. While different crypto engines yield different performance, the overhead per packet (e.g., initialization, memory operations, function calls, etc) forms a fundamental roadblock. Increasing the unit size of crypto is impossible under the current standard. If QUIC's throughput improves, the community needs to start exploring alternative designs for high-throughput QUIC connections. For example, encryption can be done on large MTUs that are then fragmented at the sender and reassembled at the receiver. Alternatively, the QUIC standard can be modified to implement crypto on larger units of data (e.g., at the stream level).

We hope that this paper will serve as an example that implementation guidelines can be developed for QUIC, to help the users of the implementation, as well as developers, cater to specific application needs. Moreover, it can help highlight shortcomings in the protocol design, potentially leading to further refinements of the protocol. We hope that this work will encourage more such guidelines as well as further improvements to the single-throughput performance of QUIC.

Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank Cloudlab [17] for providing us with the infrastructure to run these experiments. This work was funded in part by NSF grants CNS-2212098 and NeTS-1816331; and a Google Research Award.

References

[1] benchmark: A microbenchmark support library. https://github.com/google/benchmark. Accessed: 2023-06-22.

[2] fizz: C++14 implementation of the TLS-1.3 standard. https://github.com/facebookincubator/fizz. Accessed: 2023-06-22.

[3] folly: An open-source C++ library developed and used at Facebook. https://github.com/facebook/folly. Accessed: 2023-06-22.

[4] fusion: AES-GCM engine. https://github.com/h2o/picotls/pull/310. Accessed: 2023-06-22.

[5] MsQuic Performance Dashboard. https://microsoft.github.io/msquic/. Accessed: 2023-06-22.

[6] mvfst: An implementation of the QUIC transport protocol. https://github.com/facebookincubator/mvfst. Accessed: 2023-06-22.

[7] OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. Accessed: 2023-06-22.

[8] quicly: A modular QUIC stack designed primarily for H2O. https://github.com/h2o/quicly. Accessed: 2023-06-22.

[9] Schannel SSP. https://learn.microsoft.com/en-us/windows-server/security/tls/tls-ssl-schannel-ssp-overview. Accessed: 2023-06-22.

[10] Ietf quicwg - implementations wiki. https://github.com/quicwg/base-drafts/wiki/Implementations, 2023. Accessed: 2024-01-09.

[11] Nick Banks. Making MsQuic Blazing Fast. https://techcommunity.microsoft.com/t5/networking-blog/making-msquic-blazing-fast/ba-p/2268963, 2021. Accessed: 2024-01-09.

[12] Nick Banks. MsQuic API. https://github.com/microsoft/msquic/blob/main/docs/API.md, 2023. Accessed: 2024-01-09.

[13] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μ s tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 767–779, 2022.

[14] Yuchung Cheng and Neal Cardwell. Making linux TCP fast. In *Netdev conference*, 2016.

[15] Bo Cui, Brain Will, Karen Shemer, Stephen Doyle, and Guodong Zhu. IETF QUIC Acceleration using Intel QuickAssist Technology (Intel QAT). Technical report, Intel, 2023.

[16] Willem de Bruijn and Eric Dumazet. Optimizing udp for content delivery: Gso, pacing and zerocopy. In *Linux Plumbers Conference*, 2018.

[17] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[18] Alessandro Ghedini. Accelerating UDP packet transmission for QUIC. https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/, 2020. Accessed: 2024-01-09.

[19] Benedikt Jaeger, Johannes Zirngibl, Marcel Kempf, Kevin Ploch, and Georg Carle. Quic on the highway: Evaluating performance on high-rate links. In *International Federation for Information Processing (IFIP) Networking 2023 Conference (IFIP Networking 2023)*, 2023.

[20] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[21] Marcel Kempf, Benedikt Jaeger, Johannes Zirngibl, Kevin Ploch, and Georg Carle. Quic on the fast lane: Extending performance evaluations on high-rate links. *Computer Communications*, 223:90–100, 2024.

[22] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. How quick is quic? In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, 2016.

[23] Kazuho Oku and Jana Iyengar. QUIC vs TCP: Which is Better? https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency, 2020. Accessed: 2024-01-09.

[24] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[25] Florentin Rochet, Emery Assogba, Maxime Piraux, Korian Edeline, Benoit Donnet, and Olivier Bonaventure. Tcpls: Modern transport services with tcp and tls. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 45–59, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Nikita Tyunyayev. *Improving the performance of picoquic by bypassing the Linux Kernel with DPDK.* PhD thesis, UCL - Ecole polytechnique de Louvain, 2022.

[27] Nikita Tyunyayev, Maxime Piraux, Olivier Bonaventure, and Tom Barbette. A high-speed quic implementation. In *Proceedings of the 3rd International CoNEXT Student Workshop*, CoNEXT-SW '22, page 20–22, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Peng Wang, Carmine Bianco, Janne Riihijärvi, and Marina Petrova. Implementation and performance evaluation of the quic protocol in linux kernel. In *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '18, page 227–234, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Xiangrui Yang, Lars Eggert, Jörg Ott, Steve Uhlig, Zhigang Sun, and Gianni Antichi. Making quic quicker with nic offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ '20, page 21–27, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Alexander Yu and Theophilus A. Benson. Dissecting performance of production quic. In *Proceedings of the Web Conference 2021*, WWW '21, page 1157–1168, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z Morley Mao, Feng Qian, and Zhi-Li Zhang. QUIC is not Quick Enough over Fast Internet. *arXiv preprint arXiv:2310.09423*, 2023.

[32] Johannes Zirngibl, Philippe Buschmann, Patrick Sattler, Benedikt Jaeger, Juliane Aulbach, and Georg Carle. It's over 9000: Analyzing Early QUIC Deployments with the Standardization on the Horizon. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC '21)*, pages 261–275, 2021.