Making QUIC Quicker With NIC Offload

Xiangrui Yang
National University of Defense Technology
CN

Lars Eggert
NetApp

Jörg Ott
Technical University of Munich
DE

Steve Uhlig
Queen Mary University of London
UK

Zhigang Sun
National University of Defense Technology
CN

Gianni Antichi
Queen Mary University of London
UK

ABSTRACT

This paper aims at defining the right set of primitives a NIC shall expose to efficiently offload the QUIC protocol. Although previous work already partially tackled this problem, it has only considered one specific aspect: the crypto module. We instead dissect different QUIC implementations, and perform an in-depth analysis of the cost associated to many of its components. We find that the kernel to userspace communication, the crypto module and the packet reordering algorithm are CPU hungry and often the cause of application performance degradation. We use those findings to define an architecture for offloading QUIC and discuss the associated challenges.

CCS CONCEPTS

• **Networks** → **Network performance analysis**; *Network measurement*.

KEYWORDS

Network Profiling, Measurements, Acceleration

ACM Reference Format:
Xiangrui Yang, Lars Eggert, Jörg Ott, Steve Uhlig, Zhigang Sun, and Gianni Antichi. 2020. Making QUIC Quicker With NIC Offload. In *Workshop on Evolution, Performance, and Interoperability of QUIC (EPIQ'20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3405796.3405827

1 INTRODUCTION

The ever-increasing traffic workloads and the gradual slowdown in CPU performance improvements are making end-host networking progressively challenging [15, 19, 20]. Recently introduced Network Interface Cards (NICs) with programmable hardware components, e.g., network processor, FPGA, can help by easing the host CPU from expensive computation tasks [16]. For this reason, nowadays they are becoming commonplace in datacenter networks [17]. Recognizing this aspect, researchers have been looking into the role of programmable NICs in the context of TCP offload [7, 29], load balancing [8], consensus protocols [26] or key-value stores [28], to name a few.

In this work, we explore their role in the context of QUIC [27], a new transport protocol which is likely to serve a large fraction of bytes on the Internet soon [14]. Although QUIC has proven to improve the performance of connection-oriented web applications [27], it has also demonstrated to be CPU hungry, requiring up to 3.5 more CPU cycles than TCP with TLS [27]. For this reason, to fully realize its potential, defining new primitives for offloading (part of) it on new programmable NICs becomes of primary importance. While previous work advocating for QUIC offload [13] has focused its attention on one only specific component of the protocol, i.e., the crypto module, we argue that to have a complete picture, it is required an in-depth analysis of the costs associated to all of its components first.

We dissected four different implementations of QUIC (§ 2) to better understand and compare the impact on CPU utilization of different functions associated with the protocol: crypto, connection setup & tear-down, ACK and packet reordering processing, packet I/O and packet header formatting. We found that data copy between user and kernel space accounts for 50% of the total CPU usage related to the protocol. Moreover, in the presence of a kernel-bypass optimization, crypto operations become the new bottleneck, by pushing the CPU resources usage up to 40% per connection. Finally, the ratio of out-of-order packets, alongside the specific algorithm being used to cope with packet reordering, has a significant influence on the total CPU usage[1]. With this in mind, we propose an hardware/software co-design to accelerate QUIC (§ 3). We share the challenges in designing the prototype on commodity FPGA-based NICs and discuss possible pathways to overcome them.

In summary, the main contributions of this paper are:

- We present a measurement campaign carried over different implementations of QUIC to evaluate the cost of its building blocks in a number of scenarios, i.e., out-of-order packets, losses.
- We share the lessons learned from the measurement analysis and propose a hardware/software co-design of QUIC that can be implemented on commodity NICs.
- We discuss the challenges associated to our design and propose solutions to overcome them.

EPIQ'20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-8047-8/20/08.
https://doi.org/10.1145/3405796.3405827

[1] The measurement results and scripts are open-sourced and will be maintained at https://github.com/Winters123/QUIC-measurement-kit

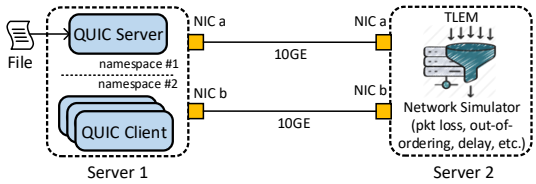


Figure 1: Testbed.

Table 1: The testbed settings.

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

2 MEASUREMENTS AND ANALYSIS

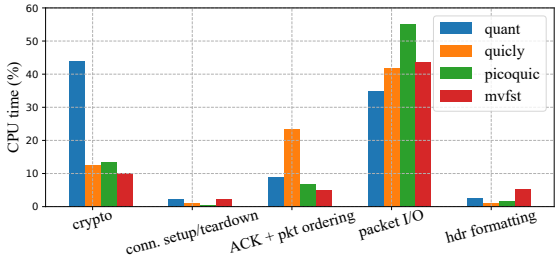
Nowadays, it is already possible to find many different implementations of QUIC [4]. They mainly differ by the programming language being adopted, i.e., Java, C, Rust, and the draft version they comply with, i.e., 20, 23, 25, 27. To quantify the potential bottlenecks in QUIC implementation, we had first to pick some of them, potentially implemented with the same programming language in order to avoid as much as possible performance discrepancies due to language-related compilation. In this regard, we decided to focus our attention on Quant [3], Quicly [5], Picoquic [2] and Facebook's Mvfst [1]. This is because, (1) they all comply to the latest IETF QUIC draft, e.g., 27; (2) they are all open-source, which is an important aspect as it allows us to add appropriate instrumentation into their source code; (3) they are all implemented in C/C++, a relative low-level language compared to the other available implementations. This last aspect allows us to avoid as much as possible overheads created by the programming language abstractions. Finally, as Quant provides support for the netmap fast packet I/O framework [33], it gives us a good comparison point to understand the actual performance implications on the protocol when using the standard Socket APIs versus kernel bypass techniques.

2.1 Testbed Settings

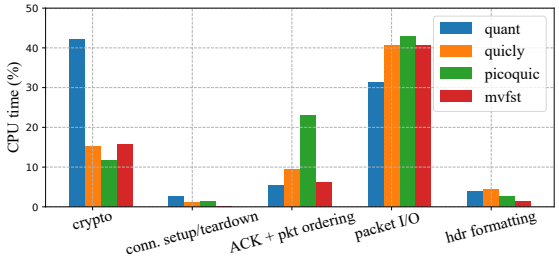
The testbed configuration being used in our tests is shown in Figure 1. It consists of two dual socket Dell PowerEdge R440 servers running Ubuntu 18.10 (from now on we call them A and B). They are connected via dual port 10G Intel NICs (UDP GSO features are disabled from NICs). We install both QUIC server and client on A and we pin both processes to different cores. We use B to introduce a number of controlled traffic perturbations, i.e., loss, delay, re-ordering, with the help of the TLEM toolkit [34]. In this setting, traffic departing from A reaches B and then returns back to A. Details are shown in Table 1. As a sample application, we transfer 50 MegaBytes from the server to the client and we repeat the same test fifteen times. During the test, more rounds are applied but the result doesn't change significantly.

2.2 Single Connection Scenario

Here, we focus our attention on the performance of QUIC when only a single connection is used. We introduce a static 1ms delay with



(a) server



(b) client

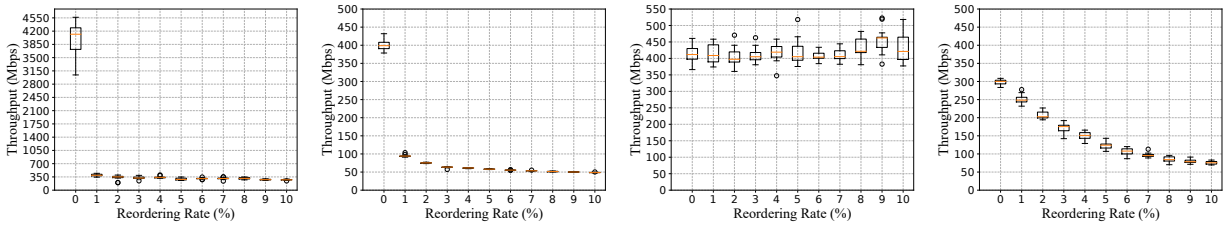
Figure 2: CPU usage breakdown of both the QUIC server and client.

TLEM to emulate the presence of a well-behaved network and we measure the throughput as well as both client and server CPU utilization. This first simple test can serve as baseline. The throughput is calculated by instrumenting the code of each QUIC implementation. The CPU utilization has been obtained with perf [12], an open source profiler tool. The results are shown in Table 2. Each implementation consumes similar amount of CPU time percentage (50%) except Quant (90%). While Quicly, Picoquic and Mvfst achieve similar throughput (<500Mbps), Quant reaches 4121Mbps on average, which is around 10x higher than the other three. This performance gap is related to the fact that Quant is configured in kernel-bypass mode using netmap, while the other three implementations only support standard socket APIs.

Table 2: Maximum throughput vs CPU usage.

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

To gain a better insight into the four implementations, we break down the overall CPU usage into the most representative functions associated with the protocol: crypto, connection setup & tear-down, ACK and packet reordering processing, packet I/O (socket system calls, netmap processing) and QUIC packet header formatting. The CPU profiling results of both the server and client are shown in Figure 2. In Quicly, Picoquic and Mvfst, the predominant CPU usage is packet I/O, contributing over the 40% of the CPU time. In contrast, for Quant it accounts only for 30%. The difference in CPU cost for packet I/O correlates with the use of netmap instead of standard socket APIs. Interestingly, over 40% of CPU time in Quant is spent



(a) quant (b) quicly (c) picoquic (d) mvfst

Figure 3: The throughput of Quant, Quicly, Mvfst and Picoquic under difference levels of out-of-order packets.

in dealing with the crypto functions. Specifically, `enc_aead()` and `dec_aead()` calls, responsible for encrypting and decrypting each packet, require up to 89% and 92% of the CPU time consumed by the crypto part on the server and client, respectively.

The cost of out-of-order packets. Here we profiled the four implementations in the presence of out-of-order packets. TLEM allows to hold a given amount of packets for a fixed time period, while letting others passing through. By carefully controlling this functionality, and ensuring that the holding time is smaller than the packet loss timeout set in the QUIC end points and the sequence gap between reordered packets and normal packets is within the packet number threshold, it is possible to fairly assess the cost of out-of-order packets. In Figure 3, we show the throughput of all four QUIC implementations when increasing the reordering rate, i.e., holding time in TLEM. Interestingly, while Picoquic does not experience any performance degradation, the other three show a drastic reduction in throughput, even when only 1% of the total packets exchanged between server and client are reordered. By looking at the server congestion window size, we measure a 30/50x decrease with respect to a normal scenario, which in turn justify the huge throughput degradation. This suggests that Quant, Quicly and Mvfst, in this scenario, treat the out-of-order packets as lost, even though they arrive before the expiration of the packet loss timeout. To confirm this, we access the available packet loss counters in the QUIC server and saw an increasing value. We believe this might be related to the inability of Quant, Quicly and Mvfst to handle out-of-order packets timely, which in turn causes the packet loss timeout to expire and a consequent shrink of the congestion window. In contrast, Picoquic appears to be more resilient in this situation, at the cost of about 26% CPU usage, which is indeed consumed by the packet reordering function.

The importance of the packet reordering engine. The previous test has shown an important insight. Being able to deal as timely as possible with packet reordering prevents from treating out-of-order packets as losses, thus preventing the congestion window to drop with consequences on the overall throughput. To better clarify the importance of the packet reordering engine, we performed more tests on Picquic as it provides two different algorithms that deal precisely with this problem. Specifically, a new version (*commit ID: 2e5c3c3*) uses a self-balancing binary search tree (splay tree), while an older (*commit ID: f4802c3*) employs a linear search to reorder packets upon arrival. Here, we ran a set of experiments that impose an increasing percentage of packet loss. We show the obtained results in Figure 4. We can see that the throughput drops more

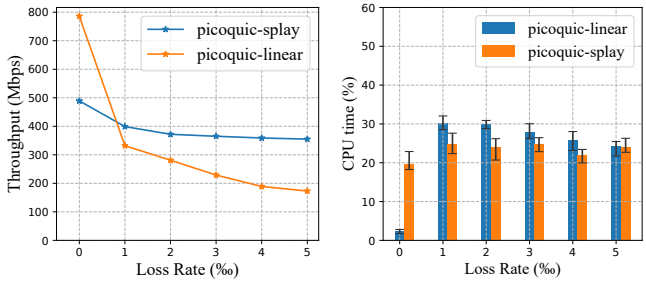


Figure 4: Picoquic throughput with different packet reordering algorithms (left) and the CPU time cost by reordering packets (right) under packet loss.

rapidly when using linear searching, though it performs better in the best case scenario (no packet reordering). CPU profiling results indicate that this is mainly caused by the complexity difference between linear and splay tree search: $(O\left(n^2\right)$ vs $O\left(logn\right)$, given n the number of frames being handled. Linear search performs better than splay tree without packet reordering since it always hits the target in the first round of the loop $(O\left(1\right))$. In contrast, the splay tree needs to be traversed every time a packet is received in order to find the leftmost leaf.

2.3 Multiple Connections Scenario

Here, we consider the situation where multiple QUIC clients interact with the same server. The tests take into consideration only Quicly, Picoquic and Mvfst, as at the time of writing this paper, Quant does not support multiple connection establishments on the server side. We launched only one server process and pinned it to a specific core which had no other working process on it. Our server counts a total of 40 cores: this means that with our setup, when we instantiate more than 39 connections, some clients will have to share the same core. In our tests, we made sure the server never had to share resources with any other process, and as long as we had cores idle we kept new clients on free cores. We ran our tests using an increasing number of clients, i.e., connections. Every client, when activated, was requesting 50MB of data from the server. We repeated the same test fifteen times.

In Figure 5 we show the per-connection average throughput we obtained. The error bars represent 95th percentile and minimum. In Figure 6, we show instead the aggregate throughput. Here we can see that in both Quicly and Picoquic, it increases linearly (with different rates) until the number of connections is bigger than 50. In contrast, the maximum throughput of Mvfst is achieved only after

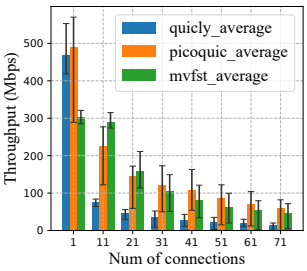


Figure 5: Per-flow average throughput

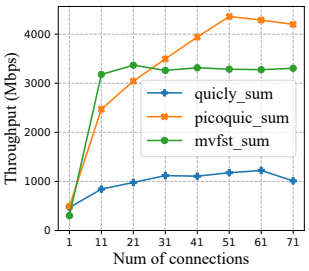


Figure 6: Total average throughput.

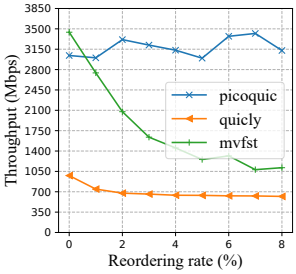


Figure 7: Average aggregate throughput with different degrees of out-of-order packets.

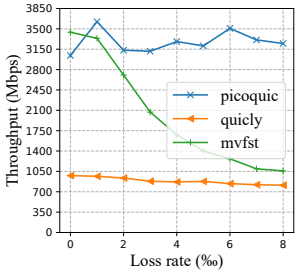


Figure 8: Average aggregate throughput with different degrees of packet loss.

10 connections. Interestingly, while Quicly, Picoquic and Mvfst have shown similar results in the single-flow scenario (Table 2), here Picoquic and Mvfst outperform Quicly of about 4x when the number of connections exceeds 40. This indicates that it is also possible to achieve high throughput without using any kernel-bypass techniques but instead relying on multiple connections. Finally, we also measured the CPU usage in the multi-connection scenario and saw that the major difference compared to the single connection scenario is that the CPU time spent on packet sending, e.g., pkt formatting, crypto, pkt I/O, has increased by about 10%.

Packet out-of-order and loss. Here we carried a similar set of experiments we did for the single-connection scenario. We used a total of 21 connections, launched simultaneously. In Figure 7, we show that increasing the amount of out-of-order packets impacts negatively on the aggregate throughput, similar to the single-connection scenario. In Figure 8, we show instead the behavior of the two implementations when increasing the packet loss rate. As a confirmation of our previous assumption regarding the importance of the packet reordering engine, it is possible to see that Quicly shows almost equal aggregate throughput when either 1% of packet loss or out-of-order packets are enforced. Again, we stress that when creating packet reordering we made sure that *no packets* would be received by the QUIC end host *after* the packet loss timeout has expired. This result strengthens our explanation that the failing to handle out-of-order packets timely causes packet loss and further leads to the decrease in the performance, e.g., throughput. Picoquic is proved instead to be more resilient by producing an aggregate throughput that fluctuates around 3.2Gbps. The throughput of Mvfst however, was heavily influenced by both packet out-of-order and packet loss, which indicates a potential bug in dealing with traffic perturbations.

2.4 Lessons Learned

Here we recap the main finding from the measurement campaign carried out in the previous section.

Lesson #1: Data copy between user and kernel space costs around 50% of total CPU usage. This can be avoided by using kernel-bypass techniques as adopted in Quant.

Lesson #2: In the presence of a kernel-bypass optimization, crypto operations become the new most expensive operation, requiring up to 40% of CPU resources per connection.

Lesson #3: The algorithm dealing with packet reordering is performance-critical. A slow search can cause to treat out-of-order packets as losses, thus pushing the congestion window to drop with consequences on the throughput.

3 TOWARDS ACCELERATING QUIC

Here, capitalizing on the lessons learned from section §2, we discuss an host-NIC co-design capable of offloading the most expensive QUIC operations.

3.1 Design Guidelines

We identified three main characteristics that shall be guaranteed in the first place:

1. Providing NIC-support for AEAD operations. QUIC encrypts packets using AEAD (Authenticated Encryption with Associated Data) algorithms and keys that are negotiated through the TLS handshake [30]. Both packet and header protection are guaranteed with those algorithms [6]. Our measurement campaign has highlighted the high costs associated to crypto operations (up to 40% of the overall CPU usage), and specifically discovered that approximately 75%-80% of the CPU usage related to crypto operations are consumed by the `aead_enc()` and `aead_dec()` methods. This is an important aspect as the mentioned functions are stateless, thus well positioned to be moved in the NIC data plane.

2. Moving packet reordering in the NIC. Once AEAD operations are moved into the hardware, it is possible to provide support for packet reordering directly in the NIC. Indeed, once both data and header are decrypted, it is possible to quickly identify the packet sequence number and then design a reordering algorithm taking into account the limited resources available in off-the-shelf programmable NICs.

3. Keeping control operations in the host CPU. Control plane related functions, i.e., TLS handshake, QUIC negotiation, generally requires expensive stateful processing which is often not possible in resource constrained environments such as the ones provided by off-the-shelf programmable NICs. Furthermore, according to our analysis they are not as performance critical to justify an hardware implementation. Such a split-design between hardware and

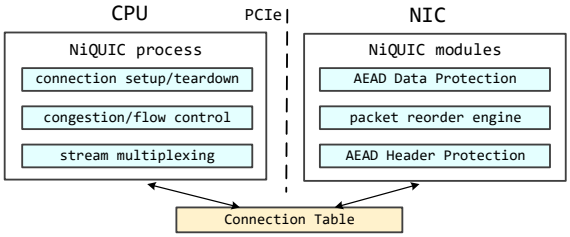


Figure 9: NiQUIC architecture.

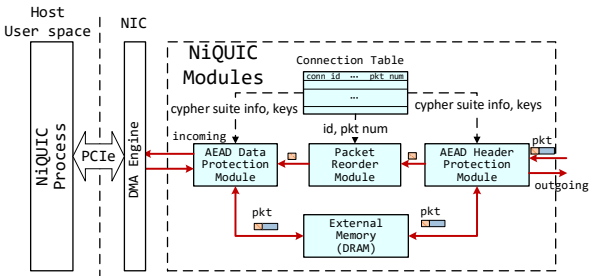


Figure 10: Key Modules in NiQUIC.

software, however, need to provide an efficient synchronization mechanism between host CPU and NIC to work.

3.2 Architecture

In Figure 9, we present a high-level view of our architecture, NiQUIC, based on the guidelines discussed above. The NIC is in charge of AEAD operations, enabling both packet and header protection, as well taking care of reordering. The host CPU, in contrast, deals with control plane operations, i.e., connection establishment/tear-down, stream multiplexing, congestion control. The synchronization between hardware and software is guaranteed through the connection table.

When the first packet belonging to a new connection arrives, the host handles version negotiation as well as cryptographic and transport handshakes. Here the hardware let all the packets going up to the stack without interfering with them. Once the connection is established, the host will add a new entry in the connection table to inform the NIC about the presence of a new flow as well as its connection ID and associated keys/cipher suite information for both header and packet protection. This approach allows the software to keep track of every offloaded connection and to decide if a specific flow is worth offloading or not. This becomes important considering the costs of writing to the connection table and the limited amount of memory available in the hardware. For instance, a short connection, e.g., a single HTTP GET request, might be well served and managed directly by the host CPU. In such cases, the host would not write anything in the connection table on the NIC.

The hardware design is shown in Figure 10. When a packet arrives from the wire, the NIC first checks if its connection ID is present in the connection table. If not, the packet is sent directly to the host CPU. Otherwise, appropriate keys are retrieved and the header is decrypted. Once this is done, the NIC can access the packet number and use it in the reordering module. This will most likely require the access to an off-chip memory such as DRAM which can be used to store out-of-order packets. The last step relates to the AEAD data protection module which is in charge to further decrypting the packet in plain text. Noted that in the opposite direction, i.e., from the host CPU to the wire, the reordering module is bypassed.

3.3 Challenges

Here we discuss the challenges in implementing the proposed architecture in a split hardware/software co-design.

Hardware/Software synchronization. In our design this is obtained with the connection table. However, commodity programmable NICs provide limited memory resources [7, 29], thus imposing a first challenge when designing this module. To alleviate the pressure on memory while allowing the server to make the most of NIC offloading, two approaches can be considered: (1) offloading only long connections and let mice flows to be entirely handled by the host CPU; (2) reduce the per-entry cost of the table by adopting appropriate hashing techniques.

AEAD protection modules. Encryption and decryption are functions commonly considered hardware-friendly [32] as they are completely stateless and require only a series of mathematical operations to be performed on the input data. They might however introduce a significant amount of latency on the packet processing compared to the CPU counterpart. This is because hardware accelerators work generally at clock speeds of just hundreds of MHz [24, 31, 36], ten times slower than commodity CPUs. This can in turn affect the overall NIC throughput. To mitigate this problem, we will consider leveraging the possibility to parallelize multiple modules and let incoming packets being process by the first available decoder/encoder. Studying the trade-offs between memory requirements and performance will be a matter of future work.

Packet reordering. Typical software implementations of packet reordering engines employ a splay-tree type of data structure [2, 3, 5]. This is, however, not efficient in hardware. The reordering process requires a memory read, to find where the newly arrived packet shall be placed, and potentially multiple writes, to reorder the various packets. While the read can be easily computed in hardware, problems arise when the reordering takes place: a new out-of-order packet might trigger multiple writing on the tree to move nodes up or down. Additionally, if two consecutive packets hit the same branch, it might not be possible to pipeline properly the read and write operations with consequences on the overall performance. We propose instead to use a TCAM that within just one clock cycle allows to map a key, i.e., decrypted packet number in QUIC, to a value, i.e., the address where the packet shall be stored in the memory. In this way, the reordering process would be guaranteed by construction: simply requesting reads on consequent packet numbers in TCAM, by which packet addresses can be retrieved perfectly in order. Additionally, an efficient timeout mechanism need to be implemented to avoid packets being stuck into the NIC for too long, waiting for out-of-order packets to arrive. Another

challenge might be related to TCAM saturation: entries being saturated when severe packet out-of-order occurs. Inspired by Large Receive Offloading (LRO) [18], a potential solution to this would be to merge N adjacent packets into a single large packet and associate to it just one packet number, which would lower the TCAM usage by $N - 1$.

4 RELATED WORK

The closest work to the one presented in this paper was proposed by Hay et al. [21], where the authors show the benefits on the QUIC protocol of offloading both crypto and segmentation in hardware. In this paper we take a step back, by first dissecting different QUIC implementations and then proposing a new offload architecture based on the obtained results. We now review the works more closely related to both measurements and offloading architectures.

QUIC measurements. In the recent past, the research community has produced a number of works measuring QUIC performance [9, 10, 13, 14]. Some of them focus on the differences between QUIC and HTTP/TCP [9, 10]. Others, investigated instead more on the impact of QUIC on the host CPU/memory but taking into account either only a specific implementation [14], or a building block [13], i.e., crypto.

Networking protocol offloading. Offloading the entire or part of the networking stack on reconfigurable hardware is a well explored concept, especially in the context of TCP [7, 22, 29, 35]. This work is a first step towards the understanding of the implications for the QUIC protocols.

Crypto operations offloading. Several academic and industrial works have explored the possibility of accelerating TLS using specialized hardware [23–25, 32]. Our proposed architecture does not require full TLS offloading, but instead, considering the limited available resources in off-the-shelf programmable NIC [7, 29], we argue for the possibility of implementing only AEAD operations in hardware.

5 CONCLUSIONS & LIMITATIONS

In this paper, we dissected four different implementations of QUIC. By analyzing the CPU cost of different protocol building blocks, i.e., crypto, connection setup & tear-down, ACK and packet reordering processing, packet I/O, packet header formatting and checksum, we identified three main bottlenecks: (1) kernel network stack; (2) crypto operations; (3) packet reordering. With this in mind, we propose an hardware/software co-design to accelerate QUIC and share the challenges in designing the prototype on commodity FPGA-based NICs. We conclude the paper by discussing potential pathways to overcome them. However, this paper currently doesn't investigate the performance influence brought by features like UDP Generic Segmentation Offload and packet pacing [11] which may improve or have potentially improved the overall performance.

ACKNOWLEDGEMENTS

We would like to acknowledge the anonymous reviewers for their invaluable feedback on this paper. This research is supported by the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL: sdn EnAbled MeasuRement for alL project (Project Reference EP/P025374/1). Yang is also supported by the China Scholarship Council.

REFERENCES

[1] 2020. Mvfst repository. https://github.com/facebookincubator/mvfst/commit/5a203c90db34b15f3cecb7a71659ebb1f93bde6f.
[2] 2020. Picoquic repository. https://github.com/private-octopus/picoquic/commit/2e5c3c31478f3696ac125c084abe36205b8b4626.
[3] 2020. Quant repository. https://github.com/NTAP/quant/commit/8f63023ab04daa1b30a5de31e91f3c18dcb7f6f9.
[4] 2020. QUIC IETF Working Group Implementations. https://github.com/quicwg/base-drafts/wiki/Implementations.
[5] 2020. Quicly repository. https://github.com/h2o/quicly/commit/6637712da98fa11a6d90c6148cabb5266b8e61ec.
[6] 2020. Using TLS to Secure QUIC. https://quicwg.org/base-drafts/draft-ietf-quic-tls.html.
[7] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *Networked Systems Design and Implementation (NSDI)*. USENIX.
[8] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling End-Host Network Functions. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[9] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. HTTP over UDP: an Experimental Investigation of QUIC. In *Symposium on Applied Computing (SAC)*. ACM.
[10] Sarah Cook, Bertrand Mathieu, Patrick Truong, and Isabelle Hamchaoui. 2017. QUIC: Better for what and for whom?. In *International Conference on Communications (ICC)*. IEEE.
[11] Willem de Bruijn and Eric Dumazet. 2018. Optimizing UDP for content delivery: GSO, pacing and zerocopy. In *Linux Plumbers Conference*.
[12] Arnaldo Carvalho De Melo. 2010. The new linux perf Tools. In *Linux Kongress*.
[13] Manasi Deval and Gregory Bowers. 2019. Technologies for accelerated QUIC packet processing with hardware offloads. In *Intel Corporation Patent*. US Patent.
[14] Lars Eggert. 2020. Towards Securing the Internet of Things with QUIC. In *Decentralized IoT Systems and Security (DISS)*. The Internet Society.
[15] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *International Symposium on Computer Architecture (ISCA)*. ACM.
[16] Daniel Firestone. 2019. Building hardware-accelerated networks at scale in the c/guloud. https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf.
[17] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mark Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish K. Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX.
[18] Leonid Grossman. 2005. Large receive offload implementation in neterion 10GbE Ethernet driver. In *Linux Symposium*.
[19] Nikos Hardavellas. 2012. The rise and fall of dark silicon. In *;login:, Volume: 37*. USENIX.
[20] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward dark silicon in servers. In *Micro, Volume: 31, Issue: 4*. IEEE.
[21] Joshua Hay, Maciej Machnikowski, Gregory Bowers, Natalia Wochtman, Joanna Muniak, and Manasi Deval. 2019. Accelerating QUIC via Hardware Offloads through a Socket Interface. In *The Technical Conference on Linux Networking (Netdev)*.
[22] Yatin Hoskote, Bradley A Bloechel, Gregory E Dermer, Vasantha Erraguntla, David Finan, Jason Howard, Dan Klowden, Siva G Narendra, Greg Ruhl, James W Tschanz, Sriram Vangal, Venkat Veeramachaneni, Howard Wilson, Jianping Xu, and Nitin Borkar. 2003. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *Journal of Solid-State Circuits, Volume: 38, Issue: 11*.
[23] Xiaokang Hu, Changzheng Wei, Jian Li, Brian Will, Ping Yu, Lu Gong, and Haibing Guan. 2019. QTLS: High-Performance TLS Asynchronous Offload Framework with Intel QuickAssist Technology. In *Principles and Practice of Parallel Programming (PPoPP)*. ACM.
[24] Takashi Isobe, Satoshi Tsutsumi, Koichiro Seto, Kenji Aoshima, and Kazutoshi Kariya. 2010. 10 Gbps implementation of TLS/SSL accelerator on FPGA. In

International Workshop on Quality of Service (IWQoS). IEEE.

[25] Xiaowei Jiang. 2019. Cooperative TLS acceleration. US Patent.

[26] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[27] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[28] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Symposium on Operating Systems Principles (SOSP)*. ACM.

[29] YoungGyoun Moon, SeungEon Lee Lee, Muhammad A. Jamshed, and Kyoung-Soo Park Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[30] Paul Morrissey, Nigel Smart, and Bogdan Warinschi. 2008. A modular security analysis of the TLS handshake protocol. In *Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer.

[31] Netronome. 2018. Netronome AgilioTM CX 2x40GbE Intelligent Server Adapter. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.

[32] Boris Pismenny, Ilya Lesokhin, Liran Liss, and Haggai Eran. 2016. TLS offload to network devices. In *The Technical Conference on Linux Networking (Netdev)*.

[33] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *Annual Technical Conference (ATC)*. USENIX Association.

[34] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. 2016. Very high speed link emulation with TLEM. In *Local and Metropolitan Area Networks (LANMAN)*. IEEE.

[35] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio Lopez-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *Field Programmable Logic and Applications (FPL)*. IEEE.

[36] Noa Zilberman, Yury Audzevich, Adam G. Covington, and Andrew W. Moore. 2014. NetFPGA SUME: toward 100 Gbps as research commodity. In *Micro, Volume: 34, Issue: 5*. IEEE.