states maintained on the other side to complete its network communication including receiving or sending data packets, notifying application, and generating ACK packets, *etc.*. Many DCN applications requiring large concurrent connections have asymmetric communication pattern, *i.e.*, only one side (called server) has huge fan-in/fan-out traffic while the other side (called client) only has a few connections [5, 10–12]. To name a few: Parameter servers (*server*) and workers (*client*) in distributed machine learning systems [12], and result-aggregators (*server*) and document-lookupers (*client*) in web-search back-end services [10]. As such, StaR can significantly improve their performance originally bottlenecked by the RNIC scalability at the server side.

We have implemented StaR in NS3 simulator. Evaluation results show that compared to original RNICs, StaR can improve the aggregate throughput by more than 160x under targeted stress tests, and bring more than 4x throughput improvement for upper-layer applications.

2 CHALLENGES AND INSIGHTS

We need to carefully address several challenges brought by moving connection states to the other side. Particularly:

- *How to complete RDMA functionalities without states on NIC?* RNIC has two major functionalities, *i.e.*, DMAing data from/to application memory and transmitting data through network. Correspondingly, we need to figure out how to: 1) determine the address to write/read data to/from the application memory; 2) notify applications when operations are completed; 3) complete reliable in-order network transmission and congestion control; 4) encapsulate packets (*e.g.*, filling IP and MAC addresses).
- *How to ensure security without states on NIC?* We can verify whether an application's operation to NIC is secure in driver software. However, without maintaining any states, RNIC cannot conduct security check on received packets. Therefore, the NIC actions triggered by the received packets may cause adversarial consequences such as accessing illegal memory address and generating malicious traffic to the network.

We address above challenges in StaR based on the following insights:

- *Carry necessary states in packets by trading off some network bandwidth.* Specifically, the client tracks the stack processing states for the server, and generates packets carrying those necessary states. Then the stateless side relies on the connection states carried inside the received packets to complete its RDMA stack processing. As §3.1 shows, the header size consumed by those states is acceptable, only incurring small bandwidth overhead.
- *Ensure security in the client.* Since StaR targets DCN scenario where all physical machines are managed by a single operator, we can ensure security by controlling the packets sent out by the client NICs. Particularly, we add a security module to each NIC, which is actually a match-action table that checks all the outbound packets. If a NIC is specified as the client side of a StaR connection, its security table will be installed with entries only allowing legal packets to the corresponding server. In addition, switches in DCN can prevent source address spoofing.

3 STAR DESIGN

We now discuss in details about how StaR conducts stack processing and security mechanism, respectively, followed by a detailed discussion on StaR's performance.

3.1 Stack Processing

Overview: Fig. 2 overviews the stateless processing in StaR. StaR offers applications an interface to specify which side to be stateless before connection starts (*e.g.*, through connection option). Then during transmission, application can benefit from StaR NIC's high-scalability without any constraints/changes beyond the standard RDMA usage.

Specifically, during connection setup, the stateless side (called server) driver will transmit all its connection states to the other stateful side (called client). Original RDMA stack creates a pair of work queues, called queue pair (QP), to maintain the stack processing context of each RDMA connection (including memory-access related states and network-transmission-related states). As such, the stateless side transfers the whole QP context to the client side during connection setup (maintaining a copy in the local host software), and the client can track and help the server NIC to finish all the stack processing.

Whenever the server application initiates an RDMA operation by posting a work queue element (WQE) to its QP on the host, the WQE will be directly encapsulated into a packet and transfered and stored in the QP maintained on the client side, rather than being stored at the local NIC. Then the client-side stack can generate corresponding packets and trigger server NIC's operation according to its WQEs in the QP. For example, a client can generate packet with DMA addresses according to the RECV/READ WQEs in the remote QP, so that the data can be directly DMAed by the server NIC. Similarly, it can get server's data using DMA addresses from the SEND/WRITE WQEs in the remote QP. It incurs cost, indeed, to send WQEs to the other side before transmitting the actual data. However, as introduced later, this has almost no impact (for receiving operations) or very small

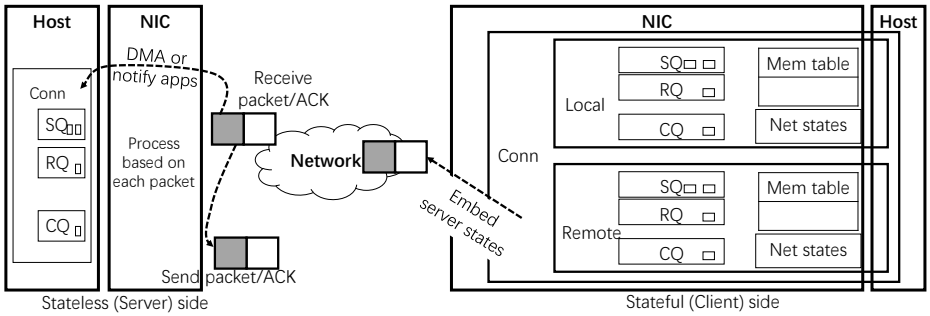


Figure 2: Stateless RDMA processing in StaR NIC: Maintain all states at one side.

impact (for sending operations) on the end-to-end delay in data center scenario.

Since RDMA applications operate on virtual memory space, original RNIC maintains a memory table (Mem table in Fig. 2) to translate the user-registered buffer address to the actual physical memory address. As such, in StaR, the server side will also transmit its memory table to the client during connection setup. Then, whenever the application registers/deregisters a user buffer later, the server driver will encapsulate it into packets and send to the client, which then update the entries in the memory table maintained in the client NIC. Note that registering/deregistering buffer is always not on the critical path of data transmission, so above procedure will not impair performance.

RDMA applications typically use TCP to setup and transmit necessary information during connection setup. As such, we omit the detailed discussion to connection setup, which is easy to understand, and focus on the data transmission in the rest of the section.

Packet types: StaR relies on the following types of packets to complete stateless RDMA processing:

1) *WQEP*, *CQEP* and *EA*. WQEP is the packet that transmits the WQE to the client, and CQEP is the packet that transmits the CQE to the server after its operation has been finished. A typical WQE/CQE with one memory region is less than several 10s of bytes [13]. WQEP and CQEP have a per-connection sequence number tagged by the driver software, and EA (Element ACK) is the ACK that clients return to servers to notify the WQEP/CQEP has been received.

2) *RDP* and *RA*. RDP (Receiving Data Packet) is the packet that contains the data going to be received by the stateless server. The data DMA address (8B) and its length (2B) are carried in each RDP's header. RDP has a sequence number, and RA (Receiving ACK) is the ACK that servers return to clients after receiving each RDP. RAs' header are generated using the header information carried in each RDP (*e.g.*, MAC and IP addresses for this connection). Such header information is relatively small. For instance, current RNIC requires only 44B states to generate a packet's Ethernet header and IP header [13].

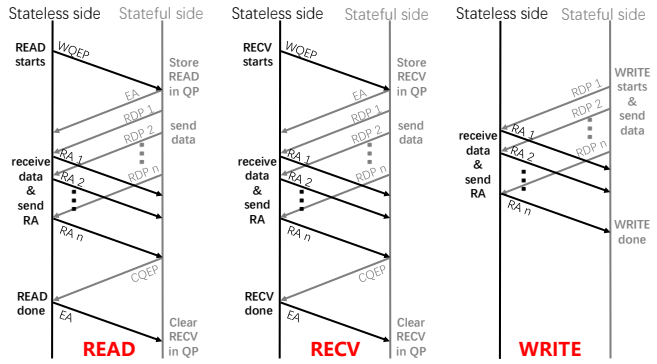
3) *GD* and *SDP*. GD (Getting Data) is the packet that a client sends to a server to trigger it to send data, which carries the data DMA address (8B) and length (2B), and the header information to encapsulate data packets (similar to RDP). SDP (Sending Data Packet) is the packet triggered by GD, containing the data sending from the server to the client.

Due to space limitation, we omit the detailed structure of each packet type. We now introduce the procedure of data reception and sending in StaR, respectively, based on how StaR handles the four RDMA verbs (WRITE/READ and SEND/RECV).

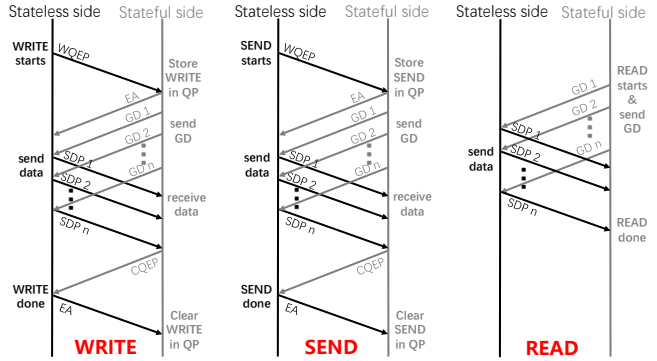
Receiving on the stateless side: There are three conditions that the stateless (server) side receives data, *i.e.*, the server application proactively READs or RECVs the data sent by the client, or the server passively receives data which the client WRITEs. Fig. 3(a) shows the processing procedure on the two sides.

Specifically, when an application on the server initiates a READ/RECV, the driver will encapsulate the WQE into a WQEP packet and notify the NIC to send it out to the client. After receiving the WQEP, the client decapsulates the WQE and stores it in the QP maintained for the server, and immediately generates an explicit acknowledgment (EA). Note that EA is necessary since WQEPs may get lost (discussed in §3.3). Then, according to the READ/RECV WQE (also depending on its own SEND WQEs), the client will generate a series of data packets (RDPs). Upon receiving RDPs, the server NIC can directly DMA the data to the corresponding memory address and generate acknowledgments (RAs) back using information contained in RDP headers. When all the data in the READ/RECV has been successfully received (tracked by the client through RAs), the client will send a CQEP, which the server can use to notify applications.

It is much simpler for the server to receive data WRITE by the client. Particularly, the server only needs to DMA data and generate RAs according to the RDP headers controlled by the client.



(a) Receiving.



(b) Sending.

Figure 3: Procedure of stateless receiving and sending.

Since the client tracks the whole transmission state, it can apply any congestion control or loss recovery mechanisms.

Sending on the stateless side: Similar as receiving, there are also three conditions that the stateless (server) side sends data, *i.e.*, the server proactively WRITEs or SENDs the data, or the server passively sends data triggered by the client READs, as shown in Fig. 3(b).

When an application on the server initiates a WRITE/SEND, the driver will encapsulate the WQE into a WQEP packet and notify the NIC to send it out to the client. After decapsulating the WQE and generating EA, the client will generate several GDs to retrieve data from the server. Upon receiving GDs, the server NIC can directly fetch the data and encapsulate it into packets (SDPs) using information contained in GD headers. The client will send a CQEP to notify server the completion of WRITE/SEND when all the data has been received (by tracking the SDPs). The procedure that the client READs data from the server is simple and similar to the WRITE case in Fig. 3(a), so we omit the discussion here.

Congestion control. For StaR stateless sender, all the congestion control calculation is done at the receiver (client) side. StaR can implement various window based congestion control algorithms (*e.g.*, DCTCP), calculating congestion window according to received SDPs. Since the size of SDP is determined by each GD, receiver can control the sender's sending rate by adjusting the GDs. Note that rate based congestion control is not applicable for StaR stateless sender as it needs per-connection states for pacing the rate.

3.2 Security Mechanism

Fig. 4 overviews the mechanism in StaR to ensure the security of NIC stateless processing. For local operations, StaR driver will check their security to prevent malicious operation to the NIC. This is typical in device driver, so we omit the discussion here. We focus on how to prevent malicious behavior triggered by the received packets.

Specifically, each NIC is enforced with an extra security check module, and all the traffic will pass through this module before sent out. Initially, all StaR traffic (packets with format introduced before) are blocked by the security module in case of threatening the stateless NIC. Note that a NIC in stateless mode only reacts to StaR packets and other packets are safe. Whenever a StaR connection is setting up, an entry will be inserted into the client's security module indicating legal packets to the server, *e.g.*, legal memory access ranges in GDs/RDPs in this connection. Note that we ensure the entries in the security module to be only modified by the stateless server through control in local NIC drivers. During the connection lifetime, the entries may also be updated by the stateless server for control plane operations, *e.g.*, registering/derigstering user buffer.

Above security mechanism can gracefully prevent stateless NIC from disrupting the host or the network. Also, different connections within a NIC can be prevented from accessing others' memory regions. However, similar as original RDMA, applications should carefully use its own registered buffers within an RDMA connection thus to prevent overwriting data or getting the outdated data.

3.3 Discussions on Performance

No state-fetching bottleneck on data transmission: In StaR, the stateless server receives/sends every data packet only based on each received RDP/GD. Since per-packet stateless processing can be well pipelined and parallelized in NIC hardware, there would be no performance bottleneck with any large scale. In contrary, in original RDMA, if a received packet belongs to a connection whose state is not on-chip, the processing has to be stalled until the NIC fetched its state from the host. Similarly, when original RNIC is going to send a packet triggered by a SEND/WRITE WQE, it has to first fetch the corresponding connection state before start the sending procedure.

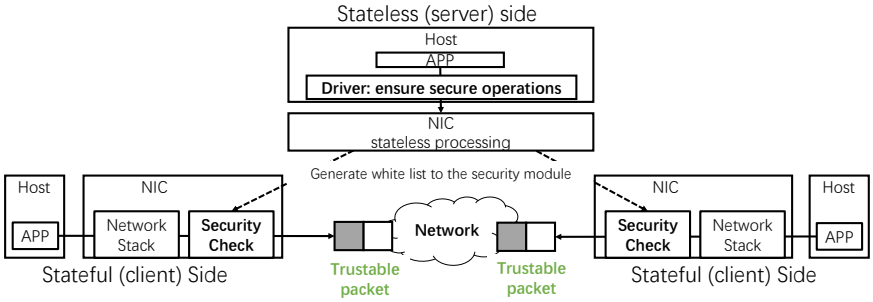


Figure 4: Ensure security for stateless processing in StaR NIC.

Cost on maintaining WQEs at the remote side: This mainly incurs three costs:

1) *Extra network delay for transmitting WQEs.* As Fig. 3(a) shows, when receiving data from WRITE, servers need not post any WQE so there is no extra delay. For READ, StaR also incurs no extra network delay before data transmission as the READ request and the READ WQE are combined in the same WQEP. For RECV, StaR indeed needs an extra 1/2 round-trip-time (RTT) to first transmit the WQE to the client. However, the impact on performance is negligible since typical RDMA applications always prepost multiple RECV WQEs and then notify the sender to start sending data.

Note that when READ/RECV is done, applications on StaR need an extra RTT to get notified by CQEPs from the client. However, a network RTT without software involvement in DCN is negligible (*e.g.*, <8 μs across switches [14]). Moreover, high-performance RDMA applications often proactively poll certain memory addresses to track the transmission status [3]. , hence this notification delay will not impair such applications' performance.

When sending data, transmitting WQEs incurs an extra RTT. StaR bears this tradeoff for stateless processing thus to improve the throughput under massive concurrent sending. There is no extra delay for application notification because original sender RNIC also waits for the receiver's ACK so it knows the sending operation has finished.

2) *Processing overhead on generating WQE packets.* Besides network delay, generating WQE packets also incurs processing overhead. While in original RDMA WQEs are directly fetched and stored in the NIC, StaR driver needs to first encapsulate them into WQE packets before passing to the NIC. However the WQEs can be preposted and well pipelined, which has no impact on the application performance (see results in Fig. 5(b)). Moreover, the processing overhead is small compared with the subsequent data transmission, because WQEPs are small-size control packets which need no complicated transport logic. Specifically, currently StaR adopts no congestion control for WQEPs, and simply encapsulating the StaR/UDP/IP/Ethernet header for them before delivered to the NIC.

3) *Handling WQE loss.* WQEs may be lost during transmission. StaR needs to handle it on the stateless side since the remote stateful side may not even know whether there are any WQEs posted if they get lost. StaR deals with WQE loss in the driver software thus to keep stateless processing in the NIC. Particularly, each EA for WQE is directly passed to the driver by StaR NIC. As the lossless network where RDMA is typically deployed on has extremely low loss rate [5], this rarely impacts the host performance. Moreover, the EA processing does not delay the data transmission procedure (see Fig. 3).

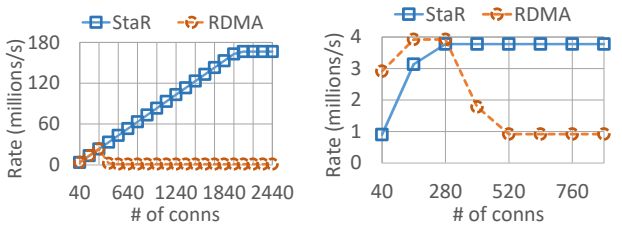
Auto adaption between stateless and stateful mode: To further optimize the cost discussed before, each StaR NIC can be implemented with two processing logics, *i.e.*, one for original RDMA connections and one for stateless connections. StaR will automatically choose the original RDMA mode for the newly created connection if current concurrent connections does not exceed the on-chip memory limit. And for more connections, it may turn into the stateless mode if the on-chip memory has been used up. We note that automatic mode transition for active connections may offer better performance as connections come and go, which we plan to study further in future works.

4 EVALUATION

Our evaluation tries to answer the following questions: *1) How does StaR perform under dedicated stress test (§4.1)? 2) How does StaR improve the performance of upper-layer applications (§4.2)?*

Simulation settings: We evaluate StaR and original RNIC (denoted as RDMA) in NS3 simulator. A server and multiple clients are directly connected to a 100Gbps switch, with a 12μs base network RTT between them. MSS is 1.4KB. The PCIe latency is set to be 1μs for RNIC to access states stored in host memory or host to deliver WQE to RNIC. For RDMA, its RNIC on-chip memory can store up to 300 connections' states (based on Fig. 1). Since our evaluation targets the RNIC scalability problem caused by on-chip states, to avoid the interference of congestion control and loss recovery, we set the

switch queue to be large enough to avoid PFC, ECN or packet loss. We do not turn on the capability of mode auto-adaption in StaR, thus to better evaluate its overhead.



(a) Stress test. (b) RPC application.

Figure 5: Aggregate throughput (MOPS) at the server under stress test and RPC applications.

4.1 Stress test

We simulate the same scenario as our testbed measurements in Fig. 1. Various number of clients concurrently WRITE to one server. All WRITEs from one client are through one long-lived RDMA connection. In each connection, there is only one outbounding 8-byte WRITE at any moment, and the client immediately initiates another 8-byte WRITE after the former one has finished. In StaR, the server is the stateless side.

Results: As shown in Fig. 5(a), as the number of concurrent connections grows, the aggregate WRITE rate in RDMA first grows to about 23M operations per second (OPS) and then quickly drops to about 1M OPS due to connection state missing. In contrary, the overall WRITE rate in StaR can scale well as the number of connection grows, reaching the 100Gbps bandwidth limitation (about 166M OPS). Note that due to the RTT and PCIe latency settings, the absolute results in this simulation are different from those in the testbed (Fig. 1), but the relative trends are the same.

4.2 Application throughput

We simulate a remote procedure call (RPC) application, where multiple clients concurrently call the remote procedure in one server. Each client sends a 2.8KB RPC request to the server (through SEND/RECV), and the server responses an 1.4KB RPC result (through SEND/RECV) after receiving the request (assuming zero RPC computation time). The client immediately repeats the RPC call after the former one completes. All RPCs between a client and the server use one long-lived RDMA connection. We assume all the RECV WQEs are pre-posted which does not affect the RPC performance. In StaR, the server is the stateless side.

Results: Fig.5(b) shows the overall RPCs finished per second in the server. StaR can always keep the maximum 3.8M OPS RPC rate (limited by the network bandwidth) as the number of concurrent connections grows beyond 280. In contrary, the RPC rate in RDMA drops to about 0.9M OPS due to state missing on the NIC. Note that StaR has lower RPC rate than RDMA when the concurrency is small, because each SEND in StaR requires an extra RTT to transmit the WQE first. However, this problem can be solved by the auto adaption scheme discussed in §3.3. The maximum RPC rate of StaR is a little lower (~2%) than the maximum in RDMA (for 160 to 300 connections) due to the overhead of extra headers and control packets.

5 RELATED WORK

StaR shares the same idea with Trickles [15] on moving connection states to the other side. However, Trickles only focuses on TCP states for software network stack, and this TCP-only solution is not fully stateless since the stateless side still needs to calculate sending rate according to TCP congestion control information of the connection. On the contrary, StaR is the first stateless NIC for RDMA, which maintains all necessary memory-access-related and network-related states on the other side and carries them in packet header, to make the server side RNIC completely stateless.

[16] builds an HTTP service based on "stateless" TCP, which is actually an TCP interface working on UDP sockets. The application has to do a bunch of network processing such as congestion control and loss recovery, which is contradicted to RDMA's design rationale.

6 CONCLUSION

This paper presents StaR. To the best of our knowledge, it takes the first step to fundamentally solve the RNIC's scalability problem, by moving all the connection states to the client side. As such, StaR server can maintain zero connection-related states while all the RDMA data plane processing is still done by NIC hardware. Preliminary simulations show StaR's potential to significantly improve the RNIC's performance under large concurrency, which might bring performance and architecture revolutions to the upper-layer large-scale RDMA applications.

7 ACKNOWLEDGMENTS

We thank the APNet reviewers for the helpful comments on improving this paper. We thank Yongqiang Xiong and Peng Cheng for valuable discussions on the idea of making one side's NIC stateless. This work was partly supported by the National Natural Science Foundation of China under grant 6187060280, Fundamental Research Funds for the Central Universities of China, and Huawei Innovation Research Program.

REFERENCES

[1] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 202–215. ACM, 2016.

[2] *Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE).* InfiniBand Trade Association, 2012.

[3] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[4] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[5] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, GA, 2016. USENIX Association.

[6] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for RDMA in datacenters. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 357–371, 2018.

[7] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 19:1–19:14, New York, NY, USA, 2019. ACM.

[8] Open MPI: Open Source High Performance Computing), 2019. https://www.open-mpi.org/.

[9] Anuj Kalia, Michael Kaminsky, and David G Andersen. Datacenter RPCs can be General and Fast. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2019.

[10] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding Up Distributed Request-response Workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 219–230, New York, NY, USA, 2013. ACM.

[11] Rajesh Nishtala, Hans Fugal, Steven M Grimm, Marc P Kwiatkowski, Herman Lee, Harry C Li, Ryan Mcelroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. pages 385–398, 2013.

[12] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[13] Linux Cross Reference. Linux/include/linux/mlx4/qp.h. http://lxr.free-electrons.com/source/include/linux/mlx4/qp.h.

[14] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[15] Alan Shieh, Andrew C Myers, and Emin Gün Sirer. Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI)*, pages 175–188, 2005.

[16] David A Hayes, Michael Welzl, Grenville Armitage, and Mattia Rossi. Improving HTTP performance using "stateless" TCP. In *Proceedings of the 21st international workshop on Network and operating systems support for digital audio and video*, pages 57–62. ACM, 2011.