StaR: Breaking the Scalability Limit for RDMA

Xizheng Wang[†], Guo Chen[†*], Xijin Yin[†], Huichen Dai[‡], Bojie Li[‡], Binzhang Fu[‡], Kun Tan[‡]
[†]Hunan University, [‡]Huawei

***Abstract*—Due to its superior performance, Remote Direct Memory Access (RDMA) has been widely deployed in data center networks. It provides applications with ultra-high throughput, ultra-low latency, and far lower CPU utilization than TCP/IP software network stack. However, the connection states that must be stored on the RDMA NIC (RNIC) and the small NIC memory result in poor scalability. The performance drops significantly when the RNIC needs to maintain a large number of concurrent connections.**

We propose StaR (Stateless RDMA), which solves the scalability problem of RDMA by transferring states to the other communication end. Leveraging the asymmetric communication pattern in data center applications, StaR lets the communication end with low concurrency save states for the other end with high concurrency, thus making the RNIC on the bottleneck side to be stateless. We have implemented StaR on an FPGA board with 10Gbps network port and evaluated its performance on a testbed with 9 machines all equipped with StaR NICs. The experimental results show that in high concurrency scenarios, the throughput of StaR can reach up to 4.13x and 1.35x of the original RNIC and the latest software-based solution, respectively.

I. Introduction

As a high-performance network stack, remote direct memory access (RDMA) has moved out from traditional HPC clusters, being widely deployed in cloud data centers [1]–[3]. By offloading the entire network stack from host to the RDMA network interface card (RNIC), RDMA makes applications directly access data in remote memory, largely bypassing the CPU.

In modern cloud data center, large-scale distributed applications are typically built on many machines, requiring frequent network communication using large number of concurrent connections [4]–[6]. However, the performance of RDMA degrades as the number of connections grows. Our testbed measurements (Fig. 1) show that even the latest RNIC can only support less than 450 concurrent connections if to maintain the peak performance (detailed settings described in §II-B).

Essentially, above RDMA scalability[1] problem is a side-effect of offloading. To complete network transmission and data DMA without involving host CPU, the RNIC has to maintain a bunch of connection-related states (DMA-related, networking-related and security-related). As a consequence, when there are too many concurrent connections, the RNIC's limited on-board memory is used up and it has to frequently fetch connection states from host memory through slow PCIe bus, which significantly hurts the performance. Realizing this problem, previous software-based solutions either try to mitigate it (*e.g.*, using large memory pages [7] or connection grouping [8]) or work around it (*e.g.*, using unreliable datagram [9]), still remaining the RNIC scalability problem unsolved.

In this paper, we ask that *can we completely remove the RNIC's scalability constraint?* To address this question, we present *Sta*teless *R*NIC (StaR). Particularly, applications that require high concurrency typically have asymmetric communication pattern, *i.e.*, only one side (called server) in the network communication has many connections while the other side (called client) only has a few [5], [9]–[11]. Observing this, the key insight of StaR is to *move all the connection states to the client side*, and maintain *zero* connection-related states on the server RNIC. Specifically, the client RNIC tracks all the states for the server, and guides the server's RNIC to complete all its data transmission including receiving or sending data packets, notifying application, and generating ACK packets, *etc.*. As such, StaR can significantly improve the performance for those applications with massive fan-in/fan-out on the server side, since the RNIC memory will no longer limit its scalability.

Although the intuition is simple, there face two big challenges in realization: 1) How to complete RDMA functionalities without states on NIC? 2) How to ensure security without states on NIC? We address above challenges in StaR based on the following insights:

- *Carry necessary states in the packets.* Specifically, the client tracks the transmission status for the server, and generates packets carrying necessary states to the server. Then the server's stateless RNIC relies on the connection states carried inside the received packets to complete its RDMA processing.
- *Ensure security in the client.* Since StaR targets data center 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, and only allow legal packets to the server's stateless RNIC.

We have implemented StaR on an FPGA-based 10Gbps NIC prototype, and built a testbed consisting of 9 machines each equipped with a StaR RNIC. Testbed evaluation results show that StaR is able to maintain maximum bandwidth as the number of connections goes up, which brings up to 4.14x and 1.35x performance improvement to upper layer applications compared to the original RNIC and the latest software-based

*Corresponding author

[1]In this paper, we define RDMA scalability as the number for concurrent connections an RNIC can support without throughput degradation.

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

solution, respectively.

II. Problems, Existing Solutions and Insights

A. Preliminaries

Communication between RDMA hosts is based on a pair of work queues (containing a send queue and a receive queue), named queue pair (QP), which will be created and maintained on the RNIC in the communication setup phase. When the application initiates an RDMA operation, it will post a work queue element (WQE) to the send queue or receive queue. The RNIC retrieves the operation from the send/receive queue, then sends or receives the data, and then notifies the application the completion of the operation by posting a completion queue element (CQE) to a dedicated completion queue (CQ).

Applications post RDMA operations through the `verbs` API: SEND, RECV, READ and WRITE. READ/WRITE are memory-access operations which allow remote memory to be written/read without involving the remote CPU, while SEND/RECV are message-passing operations which require the explicit participant of the remote CPU to transmit data. RDMA provides both reliable/unreliable and connected/unconnected transport mode for applications. Specifically, there are three modes in RDMA: *Reliable Connected* (RC), *Unreliable Connected* (UC), and *Unreliable Datagram* (UD). As RDMA being widely deployed in large-scale data center applications, it has to cope with packet loss caused by failures or hardware/software bugs, which is not uncommon even in well-engineered data center networks [12]–[14]. As such, the reliable mode RC is the most widely used in current data center applications [2], [3], [15], [16]. Therefore, we focus on the RC mode in this paper.

The RC mode provides connected and reliable transmission, but also requires to maintain significant states on the RNIC for each connection. There are three kinds of states an RNIC has to maintain for each RDMA connection[2]: 1) DMA-related, 2) networking-related and 3) security-related. DMA-related states are used for memory access, including WQE number, user page size, memory table address, *etc.*. Networking-related states are used for network transmission, including queue pair number (QPN), packet sequence number (PSN), transport timeout, *etc.*. Security-related states are used for security check, including protection domain (PD), memory region (MR) and memory window (MW), *etc.*. A typical RDMA implementation requires 256B states for each RDMA connection [17].

B. RDMA scales poorly

1) Experimental results: Many previous works have already measured the scalability problem in RNIC [7]–[9], [18], [19]. However, their results are based on RNICs released several years before, and it is unclear whether such problem still exists and how severe it is for current high-end RNICs. As such, we conduct the following experiments to evaluate the RDMA scalability on Mellanox ConnectX-6 (CX6) Dx EN NICs, which are the latest RNIC (released at 2019) at the time of writing this paper.

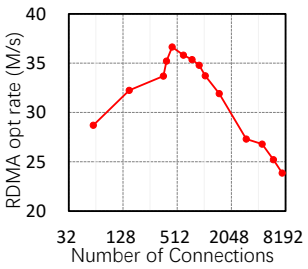


Fig. 1. Performance degrades when the number of concurrent connections grows on Mellanox ConnectX-6 Dx EN NIC.

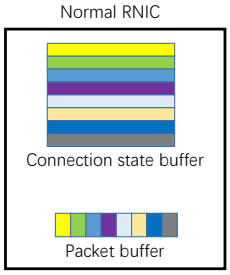


Fig. 2. State and packet buffer in normal RNICs.

Specifically, we connect 1 server machine with 3 client machines (multiple clients to avoid bottleneck at the client NIC), using CX6 NICs with 100Gbps port, through a Huawei 100Gbps switch. The server concurrently issues RDMA READ requests to fetch 32-byte data from all the three clients through multiple RC connections, keeping one out-bounding READ on each connection[3]. To avoid CPU bottleneck (two-socket Intel Xeon E5-2650 CPU), we use multiple threads on the server to issue READ requests fast enough. To evaluate RNIC scalability, we vary the number of concurrent connections and measure the overall throughput at the server (READ operations completed per second). Figure 1 shows the result. When the concurrency is low, the overall throughput grows as the number of concurrent connection grows larger, since there are more out-bounding READ requests on the wire. However, as the number of concurrent connection grows beyond 450, the overall throughput starts to drop fast and is less than 70% of the peak performance when there are more than 3000 connections, while the CPU utilization is still low. As we can see, although CX6 has higher port speed and more on-chip resources than previous CX5 or CX4 NICs, the RNIC scalability problem still exists.

2) Fundamental problems: Due to the cost issue, the RNIC on-chip memory is very small, typically serving as a cache for only part of the connections and the whole states of all connections are stored in the host memory[4]. When the concurrency is high, upon dealing with connections whose states are not stored on the NIC, the NIC has to stall and fetch the states from host memory through slow PCIe bus, which significantly hurts the performance.

As Fig. 2 shows, typically there is a small packet buffer on the RNIC to temporarily hold packets thus to amortize the hardware processing delay for each packet so the RNIC can saturate the bandwidth. When the bandwidth-delay product (BDP) grows, it needs a larger packet buffer which has to

[2]Without explicit specification, RDMA connection refers to RDMA RC connection in the rest of the paper.

[3]Our measurement code is based on the open-sourced code in [18]

[4]Although modern high-end NICs are equipped with large off-chip DRAMs, they are too slow for connection states which is used in every packet's processing, especially for high-speed RDMA network which has 100Gbps (or higher) throughput.

hold more packets. However, packets may belong to different connections, which require different connection-related states to process. As such, the buffer space to hold connection states has to be larger. This issue becomes even more significant when dealing with many small RDMA requests in high-bandwidth network, because there would be a large number of small packets in the packet buffer which may require many connection states, easily causing state miss on the NIC memory. Moreover, connection state cache miss under high concurrency will incur a temporary stall for packet processing (waiting states from PCIe), which in turn needs a much larger packet buffer to hold incoming packets, thus further overloading the burden of NIC memory. Otherwise, without enough memory, the NIC has to stall the pipeline and stop receiving packets, hurting performance.

C. Existing solutions and limitations

Existing solutions mainly focus on constraining the usage of upper-layer software, thus to avoid the scalability issue of the lower-layer RNIC. Particularly, they can be classified into two types:

- *Avoiding generating too many connections by using software middleware to control the application communication pattern [8], [20], [21].* They try to schedule the communication requirements among multiple connections, and purposely serve only part of the connections on the RNIC during a certain period of time, thus to avoid high concurrency on the RNIC. For example, ScalaRDMA [8] classifies connections into several groups, and allows just one group to run at a time by blocking other groups' connections that are about to initiate operations, thus to release the burden of lower-layer RNICs. However, using middleware brings the overhead back to CPU which is contrary to the principle of RDMA. Also, agnostic scheduling between connections in the middleware below may impair upper-layer applications' performance, since there may exist dependency between different connections. For example, in distributed machine learning with a server-worker structure, the next round of the parameter distribution can only begin after the last round of the parameter calculation is completed (assuming the most widely used synchronous mode). This requires the server node to distribute parameters and receive all the responses in a timely manner. However, if scheduling RDMA connections in the middleware agnostic to this application requirements, the performance will be hurt significantly although the RNIC concurrency is not that high (see results in §V).
- *Using RNIC in unreliable mode instead of reliable mode [9], [22], [23].* Although RNIC does not need to maintain connection-related states in the unreliable mode, thus has no scalability issue, it may cause two-fold problems: 1) either it may have very low performance when the network is lossy (which is not uncommon even in well-engineered data centers [12]) due to lack of transmission reliability, 2) or it may incur very high CPU cost because it has to deal with transmission reliability back in the software [24].

D. Opportunities

We observe that for RDMA applications that require high concurrency in the RNIC, not both sides of a connection may have high concurrency at the same time. Particularly, there are often some RNICs which have huge fan-in/fan-out while other RNICs only keep a handful of connections. For example, servers in Timestamp Oracle [25] typically need to keep more than 2800 concurrent connections, meanwhile, because of the small number of servers, the clients only need to maintain a few connections. Such asymmetric traffic pattern offers us a great opportunity to design a novel asymmetric RNIC state-maintain mechanism. Specifically, we do not maintain states on both sides of a connection. Instead, we could move both sides' states of the connection to the RNIC side with only a few connections (called client side), and let the other side which has a large number of connections to be stateless. There exists work [26] before to apply this idea to TCP/IP protocol stack. Inspired by them, we design StaR[5] to solve RDMA scalability problem.

III. StaR Design

A. Overview

StaR RNICs are totally compatible to current RDMA API, requiring no changes to existing applications[6]. As mentioned before, the principle of StaR is to maintain the connection states of the high-concurrency side's NIC on the other low-concurrency side's NIC. The architecture of StaR is shown in Fig. 3. During communication, one side's NIC is totally stateless (called server) and the other side's NIC is stateful (called client). The stateful side tracks the data transmission states for both sides, and the stateless side's NIC works according to the information embedded in the incoming packets, maintaining no connection states.

In the rest of this section, we first describe the seven packet types used in StaR. Then we dive into the detailed data transmission steps of StaR. Next, we introduce StaR's security mechanisms. Finally, we discuss several other design points and limitations in StaR.

B. Packet types

There are seven types of packets in StaR, as summarized in Table I. Specifically:

- WQEP, CQEP and EA. These three packets are used to control the start and the end of an operation. When the server initiates an operation, it encapsulates a WQEP (WQE Packet) and sends it to the client, containing information such as the operation type, data address, and data length. The server NIC does not record any states for this operation and remains stateless during data transmission. When the transmission is complete, the client

[5]We presented a preliminary idea of StaR in an earlier workshop paper [27].

[6]Except for a minor API extension in an optional connection setup mode. See details in §III-C1.

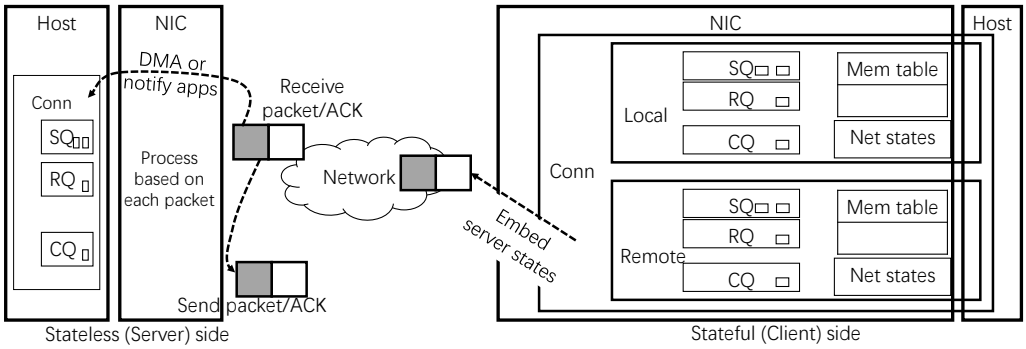


Fig. 3. Stateless RDMA processing in StaR NIC: Maintain all states at one side.

Packet Type	Description
WQEP	Sent by stateless side, to initiate events
CQEP	Sent by stateful side, to notify event completion
EA	Sent by stateful side, to ack WQEP or CQEP
RD	Sent by stateful side, to carry data to stateless side
RDA	Sent by stateless side, to ack RD
GD	Sent by stateful side, to fetch data from stateless side
GDA	Sent by stateless side, to ack GD and carry data to stateful side

TABLE I
STAR PACKET TYPES.

encapsulates a CQEP (CQE Packet) and sends it to the server. The server returns a EA (Event Acknowledgment) packet according to the received CQEP and generates a CQE to notify applications. CQE is also generated at the client after receiving the EA for confirmation. It is worth noting that the client confirms the delivery of WQEP or CQEP by receiving EA. This is because only the client stores states, so it is necessary for reliability.

- RD, RDA. RD (Receiving Data) is the packet that carries data to server, which also carries the states required by the server to help DMA the received data and organize the corresponding RDA (RD ACK) back. The RD/RDA is a packet pair. The client can retransmit RDs (if necessary) by tracking received RDAs to ensure reliability.
- GD, GDA. GD (Getting Data) is the packet sent by the client to the server to get data from server host memory (through DMA), which also guides the server to encapsulate the data packet (GDA) and send it to the client. Similar as RD/RDA, the client can track received data packets (GDAs) to ensure reliability (thus we call data packet GDA, standing for GD ACK).

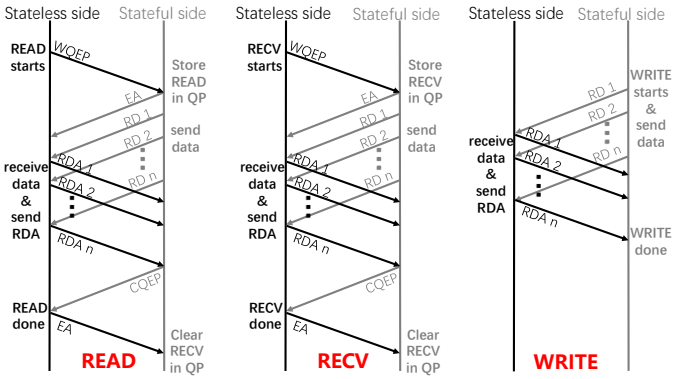
C. Data transmission

1) Connection setup: When setting up a connection, the two sides negotiate to determine which side is stateless. Once determined, the stateful side receives all the initial states (*e.g.*, QPN, page size, PD) from the stateless side and stores it on the board. Both sides' role do not change until the connection is closed. We devise two alternative modes to specify which side is stateless during connection setup: a) *explicit mode*, where RDMA users directly specify the stateless/stateful role of the RNIC through an extended API when creating connections, and b) *auto-negotiation mode*, where RNICs on both sides negotiate the role by themselves according to the current concurrency degree on each side.

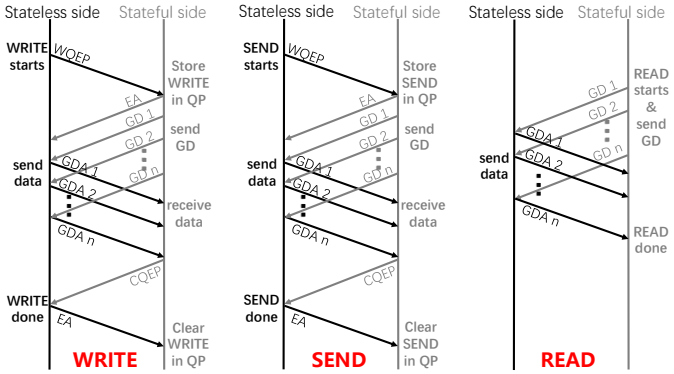
In the auto-negotiation mode, RDMA users need no explicit specification during connection setup. Both NICs automatically send the number of stateful connections currently maintained on-board to each other when a new connection is about to be established. If the number of connections on one side (or both) exceeds a certain threshold (threshold configurable), the two sides will negotiate that the side with fewer connections works as the StaR stateful side and the other one works as the StaR stateless side. After negotiation, the stateless side transfers all the initial connection states to the stateful side through network in a reliable manner (with explicit ACK), and remains a copy in local host memory. After connection setup is finished, both sides are ready for posting WQEs and transmitting data.

2) Posting WQE: When an operation request is posted from the host to the RNIC in the form of a WQE, the stateful side and the stateless side will behave differently. On the StaR stateful side, the WQE is stored on the RNIC waiting to be processed, which is the same as original RNIC. On the StaR stateless side, upon getting a WQE from the host, the RNIC directly packs it into a WQEP and sends it to the other (stateful) side, without storing it locally. We have modified the RDMA lib, so the necessary information used for packing WQEPs is passed from the host when applications initiate a WQE, requiring to store no states on the RNIC. Next, when the stateful side receives the WQEP, the RNIC parses the WQEP and gets the necessary information for data transmission (*e.g.*, the starting memory address and the length of the data). After that it starts to get/send data from/to the stateless side (introduced in §III-C3).

Note that WQEP may be dropped in the network, in which case the operation request may get lost and the application is not notified. Therefore, when the application initiates an operation request on the stateless side, besides generating a WQEP, we start a timeout timer (recorded in our modified RDMA lib software) to monitor whether this WQEP has been



(a) Receiving.



(b) Sending.

Fig. 4. Steps of receiving and sending data on the stateless side.

successfully received by the stateful side. Moreover, we let the stateful side immediately return a EA back to the stateless side when receiving a WQEP. When the stateless side's RNIC receives a EA, the EA is directly passed to our software lib and the corresponding timeout timer will be canceled. Otherwise, when the timeout occurs, the software lib will re-post the WQE to the RNIC and a WQEP will be sent out again. If the WQEP has not been successfully received after multiple timeouts, a CQE indicating operation request error will be delivered to the application. Note that EA is not on the critical path of initiating operation and transmitting data, so it does not hurt the performance.

The first several steps in Fig. 4 show how it works when posting WQEs on the stateless side.

3) Sending/Receiving data: We introduce the data transmission steps based on two scenarios, *i.e.*, when the stateless side *receives or sends* data.

Receiving data on the stateless side. There are three conditions that the stateless side receives data, *i.e.*, a) stateless side proactively calls READ, b) stateful side posts SEND and stateless side correspondingly posts RECV, c) stateless side passively receives data when stateful side posts WRITE. Fig. 4(a) shows the working steps on the two sides. When the stateful RNIC has received the WQEP from the remote side (or WQE from local application), it can easily find the memory address of the data going to be sent to the stateless side according to the information embedded in the WQEP (or WQE). Then, the stateful RNIC encapsulates the data into RD packets and sends them to the stateless side. Each RD packet contains the destined memory address, data length, and other necessary states, such as QPN, PSN, *etc.*. When the stateless side receives an RD packet, it performs two operations, *i.e.*, DMA data to the host memory and return RDA, which can be easily done using the information contained in the RD packet. Specifically, the memory address and data length are used to encapsulate the DMA descriptor to transfer data from RNIC to host memory. The RDA packet has the same header as the RD packet but with the destination and source IP/port/QPN reversed. Detailed packet format is introduced in §IV. The stateful side tracks data transmission according to the RDAs, and retransmits RDs if necessary (when packets lost) for reliability.

Sending data on the stateless side. Similarly, there are three conditions that the stateless side sends data, as shown in Fig. 4(b). When the stateful RNIC has received and parsed the WQEP from the remote side (or WQE from local application), the stateful side generates GD packets according to the WQEP (or WQE) to fetch data from the stateless side. Each GD packet contains necessary DMA-related and networking-related information. According to the carried information, the stateless side can fetch the target data from host memory and pack it into GDA packets and send them back to the stateful side. The stateful side receives the data in GDA packets, tracks data transmission according to the GDAs, and retransmits GDs to fetch data again if necessary (when packets lost) for reliability.

4) Generating CQE: After all the data in a WQE has been successfully transmitted, if the WQE is posted by the stateful side, its RNIC can directly generate the CQE and push it into the corresponding CQ (in local host memory), which is the same as the original RNIC. However, if the WQE is posted by the stateless side, the stateful side's RNIC will generate the CQE for the stateless side and encapsulates it into a CQEP and sends it to the stateless side. When receiving CQEP, the stateless side's RNIC retrieves the CQE from the packet, DMAs it to the corresponding CQ, and returns a EA back, all according to the information carried in the CQEP. If EA has not been received, the stateful side will retransmit the CQEP.

D. Security mechanisms

Security model: Our security model assumes that all the NICs (including the hardware and the corresponding driver/lib software) in the network are deployed and controlled by trusted data center operators. This assumption to the trusted computing environment is obviously stronger than current RDMA. We discuss the feasibility and limitation in practice in §III-E3. Based on above assumption, StaR adopts the same memory protection mechanisms as in current RDMA (PD, MR, MW)[7], except that StaR moves the protection mechanisms on the stateless side to the other stateful side. Note that due to the lack of authentication and encryption,

[7]Detailed RDMA security mechanisms can be found in [28].

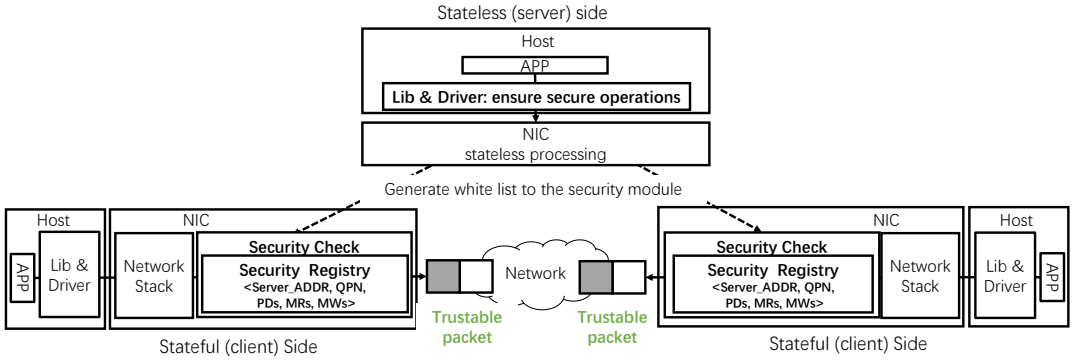


Fig. 5. Ensure security for stateless processing.

current RDMA can not cope with malicious attackers who may manipulate packets in the middle or generate deliberately forged packets from uncontrolled hosts, which is the same as StaR.

Current RDMA prevents malicious memory accesses from remote side relying on checking if the packet contains information that matches the maintained states. For example, the RNIC will only process and DMA the data in a packet only if it has the right access tokens (*e.g.*, *rkey*) corresponding to the target data memory region (registered before transmission). However, without maintaining those states, the StaR stateless side's RNIC is not able to do security check upon receiving packets from the network. As such, we design a mechanism that conducts security check at the client side, which ensures the packets sent to the stateless side's RNIC are all legal.

Particularly, we insert a hardware *security check* module in every NIC, which checks every packet before sending it out to the network. Security rules (stored in the security registry table), which are used to filter packets, will be loaded into the module during connection setup and memory region registration. Fig. 5 overviews the security architecture. Now we dive into the details that how security registries are updated and packets are checked.

1) Updating security registry table when establishing/closing connections or registering/deregistering memory regions: The security registry table contains a set of 5-tuple rules, *i.e.*, *<Server Address (Server_ADDR), Queue Pair Number (QPN), Protection Domain (PD), Memory Region (MR), Memory Windows (MW)>*[8], which are the whitelist indicating that for packets destined to a certain server, which memory regions are legal to access according to the connection (QPN) it belongs to and the corresponding PD, MR and MW. On the stateless side, when the application establishes a connection and binds it to a PD (*e.g.*, through `ibv_create_qp(pd, ...)`), the lib/driver will intercept the call and insert a corresponding whitelist rule to the security check module on the corresponding stateful sides' NIC, indicating which PD the QP belongs to. When a new MR is registered to a PD (*e.g.*, through `ibv_reg_mr(pd, ...)`), the lib/driver will update the corresponding rule on the stateful sides' NIC, indicating which PD the MR belongs to and its access tokens. Similarly, the rules will be updated or removed when the MR is deregistered or the connection is closed. StaR relies on reliable and secure network transmission (*e.g.*, the mature SSL) to ensure the lib/driver can update rules successfully and correctly on the other side's NIC.

2) Checking packets before sending them out of the NIC: Every packet is checked before sent out from the stateful side's NIC. If there is no matched whitelist rule (*e.g.*, packets from certain QPs try to access unauthorized MRs), packets will be discarded by the NIC and will not be sent out. Note that the security module exists on every NIC including the stateless side's, however, there would be rules only on the stateful side and stateless side will not perform rule checking (more details in §IV-B3). As the stateful side has low concurrency (so the number of rules is small), it will not be the performance bottleneck.

E. Discussions and limitations

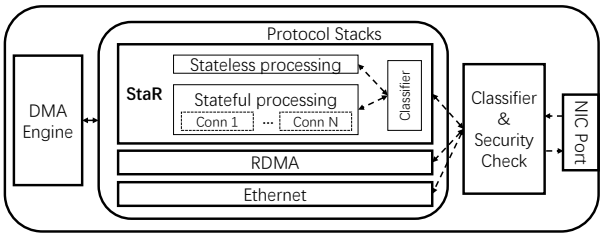


Fig. 6. Multiple stacks on StaR RNIC.

1) Stateless and stateful StaR on the same NIC, as well as other stacks: All the description in this paper on the stateless and stateful side of StaR is in terms of one connection. Note that a StaR NIC can simultaneously hold stateless and stateful connections as Fig. 6 shows (the StaR box on the top). Packets are classified to the right processing logic according to a certain header field. As discussed in §III-C1 before, a connection on a StaR NIC can either be specified or auto-negotiated to be stateless or stateful, and different connections

[8]Note that each tuple may consist of multiple fields. For example, a MR contains the registered memory address, the corresponding *rkey* and *lkey*, and the access restrictions such as read-only, write-only.

on the NIC can work in different mode (stateless or stateful). Moreover, other stacks such as normal RDMA and Ethernet can be integrated into StaR NIC for user to choose, with a classifier and security module (§III-D) in the front, as shown in Fig. 6. Note that the security module does not affect the performance of other stacks, since their packets have different formats (differentiate by certain fields) which will not be processed by the StaR stack (neither the StaR stateless logic), so these packets do not go through the whitelist rules. We have implemented a normal RDMA stack for comparison on our StaR NIC (see §IV).

2) Performance trade-off, stack selection and mode/stack switch: StaR eliminates the scalability issue on the stateless side, but adds latency for posting WQEs and delivering CQEs, trading off an extra RTT for packing WQEP/CQEP and transmit to the other side, and also burdens the stateful side. For applications with low-concurrency or with high-concurrency on both sides, StaR may have inferior performance. Thus for such communication, users can select normal RDMA stack on StaR NIC. Moreover, after connection setup, currently a connection will use the (if) selected StaR stack and remain the stateless/stateful mode until it is finished. However, traffic patterns may change dynamically, *e.g.*, the number of concurrent connections may rapidly grow on the low-concurrency side during communication. To optimize the performance under such scenarios, we can add dynamic mode/stack switch for StaR. Specifically, StaR NICs can periodically monitor the concurrency degree on both ends during communication. If the concurrency degree changes, we can dynamically switch the stateless/stateful side in the connection, or even switch the connection to use normal RDMA stack, all keeping transparent to the upper-layer applications. We plan to study the dynamic mode/stack switch in our future work.

3) Security consideration and incremental deployment: To protect the stateless side, StaR security mechanism requires all machines to be equipped with only StaR NICs that have the security check module. Such precondition is feasible for building new data centers, but may be too strong for existing data centers since it is difficult to replace all NICs at once. To incrementally deploy StaR in existing data centers, there are two possible options. 1) Configure a dedicated network partition that is separate to other network parts (*e.g.*, using VLANs), and replace all the NICs in the partition to StaR NICs. 2) Add a kernel security module to the hosts which use conventional NICs, to filter malicious packets destined to StaR stateless servers (may incur performance overhead).

IV. Implementation

Since commercial RDMA NICs have fixed stacks which cannot be modified, we have implemented the StaR NIC from scratch based on a Xilinx FPGA board with an `xcku040-ffva1156-2-e` FPGA chip, four 10Gbps SFP+ ports (only one of them is used in our implementation), and 8 lanes of PCIe Gen 3 bus. Our FPGA implementation contains 3031 lines of Verilog code. Also, we implement the StaR software lib based on the FPGA board driver provided by Xilinx. Next, we first introduce the detailed packet format used in StaR implementation, then, we describe our hardware (FPGA) and software (lib) implementation, respectively.

A. Packet format

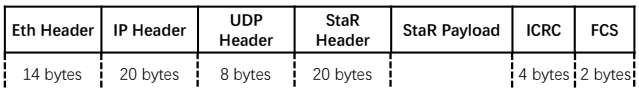


Fig. 7. StaR packet format.

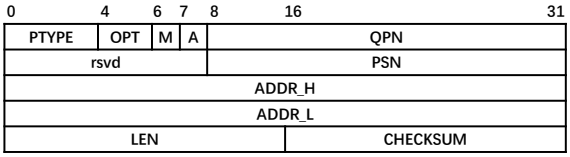


Fig. 8. StaR packet header.

The format of the whole StaR packet is shown in Fig. 7. We use a dedicated UDP port to identify StaR packets. For fast classification, StaR packets to stateless and stateful processing logic have different UDP ports. The specific format of the StaR header is shown in Fig. 8, which is similar to the Infiniband header. PTYPE indicates the data packet type. OPT indicates the operation type (*e.g.*, SEND/RECV/WRITE/READ). M indicates the states migration, only used for the stateless side to transfer states during connection setup. A indicates whether it needs to reply an ACK (to trigger sending ACKs on the stateless side). For RD/RDA and GD/GDA packets, ADDR_H and ADDR_L together form a 64-bit memory address for the transmitted data (ACKs just echo back the data address). For WQEP/CQEP packets, the ADDR_H and ADDR_L fields are not used and WQE/CQE is carried in the payload. LEN indicates the payload length. QPN, PSN and CHECKSUM are the same as in the Infiniband header.

B. Hardware implementation

We have implemented all StaR RNIC's hardware processing logic in the FPGA chip. Our hardware implementation architecture is shown in Fig. 9. From the left part of the figure to the right, our FPGA implementation contains the following modules.

1) DMA module: DMA module consists of a DMA engine (IP core), a DMA buffer and our customized DMA processing logic. The DMA processing logic contains a scheduler, arbiter and several queues for buffering DMA requests/responses and send events. The DMA processing logic coordinates the DMA requests and DMA responses. The DMA requests from the *StaR processing logic* will be checked for validity, scheduled, and then executed by the DMA engine after queuing. The executed request is buffered and the module notifies the *StaR processing logic* via inserting send events into the send event queue when the corresponding DMA response returns, *i.e.*, the DMA request has finished (return success when the data has been DMAed, otherwise return error).

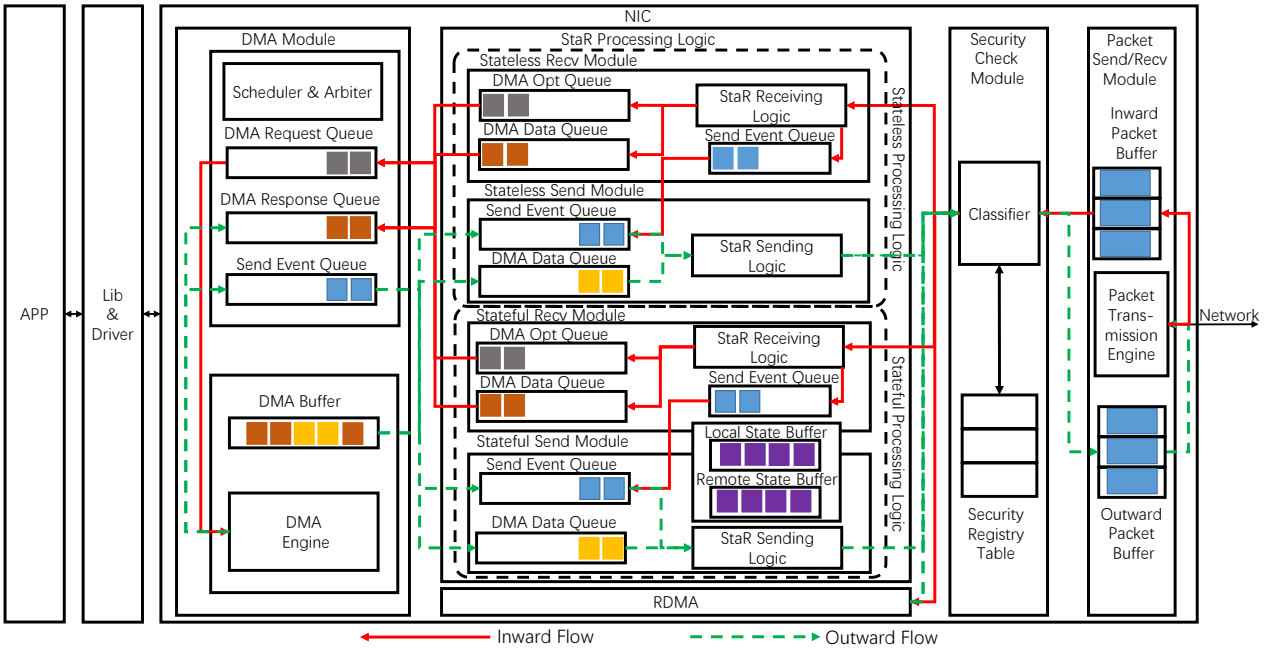


Fig. 9. StaR NIC implementation.

2) StaR processing logic: This part processes the core StaR stack logic, consists of two independent sub-parts, *i.e.*, *stateless processing and stateful processing*. In general, the two sub-parts have the same architecture, consisting of a recv module and a send module, performing three common functionalities: a) receiving and parsing packets from network, b) DMA data and WQE/CQE from/to host memory, and c) sending packets to network. In the *recv module*, after the StaR receiving logic receives and parses the packet and has something to DMA (data or WQE/CQE), it will generate DMA requests to the DMA module, by putting DMA operation metadata in the DMA Opt Queue and pushing DMA data (if any) in the DMA Data Queue. In the *send module*, send events from host DMA (*e.g.*, local WQEs) or recv module (*e.g.*, ACKs triggered by received packets) are pushed in the send event queue, then processed and packed with data (if any) from DMA data queue by the StaR sending logic into the form of packets, and passed to the security check module. The difference between stateless and stateful processing sub-parts is that packets and DMA operations are whether processed based on the information carried in the incoming packets or based on the states stored in the on-chip state buffer (local state buffer and remote state buffer).

3) Security check module: Security check module consists of a classifier module and a security registry table. It checks both the incoming and outgoing packets. Particularly, all the incoming and outgoing packets are checked by the classifier module first. For *outgoing packets*, if not targeting any StaR stateless processing on other NICs (differentiating by specific UDP port), they will directly bypass the security module since they will not cause security issue for stateless processing. Otherwise, outgoing packets with stateless targets will go through the security registry table, and will be sent out only if there is a matched whitelist rule already installed. Since the security rule is relatively complex containing multiple tuples and fields (see §III-D), we implement them using exact match based on high-speed on-chip SRAM. For *incoming packets*, all packets except for those targeting the stateful processing logic (differentiating by specific UDP port) will bypass the security module. For packets targeting the StaR stateful logic, we use the same protection mechanisms as in normal RDMA (PD, MR, MW, *etc.*), and check packets based on states stored locally in the security module.

4) Packet send/recv module: This module sends and receives packets through a 10Gbps SFP+ port. We implement it based on an existing FPGA IP Core.

5) RDMA processing logic: For fair comparison under the same hardware environment, we have implemented a simplified normal RDMA stack in parallel with StaR stack. The RDMA stack is largely similar to the StaR stateful processing logic in implementation (but only store one side's states). We omit the detailed description here.

Resource	Utilization	Available	Utilization%
LUT	39338	242400	16.23
LUTRAM	6700	112800	5.94
FF	36138	484800	7.45
BRAM	140	600	23.33

TABLE II
FPGA RESOURCE CONSUMPTION FOR STAR STATELESS NIC.

FPGA resource consumption. For clarity, we separately evaluate the resource consumption of our implementation on the stateless part and the stateful part. Particularly, we implement two dedicated NICs that only process StaR stateless stack (denoted as StaR stateless NIC) and StaR stateful

Resource	Utilization	Available	Utilization%
LUT	40277	242400	16.62
LUTRAM	6702	112800	5.94
FF	36882	484800	7.61
BRAM	178.5	600	29.75

TABLE III
FPGA RESOURCE CONSUMPTION FOR STAR STATEFUL NIC.

stack (denoted as StaR stateful NIC), respectively. Note that each NIC is complete, having all other modules implemented including DMA, packet send/recv, security check, *etc.*, Table II and Table III show the resource consumption on our FPGA.

C. Software implementation

We have implemented a simplified user-level lib compatible with current RDMA API (emulate Linux `libibverbs`) based on the FPGA board driver provided by Xilinx. We add a flag to `ibv_create_qp()` for users to specify stateless or stateful mode when establishing a connection. Moreover, we modify the code inside several APIs (including `ibv_reg_mr()`, `ibv_post_send()`, *etc.*) so that StaR NIC can load security rules and pack/send WQEP packets.

V. EVALUATION

We evaluate the performance of StaR in a real testbed consisting of 9 machines, and compare it with normal RDMA as well as the latest software-based solution.

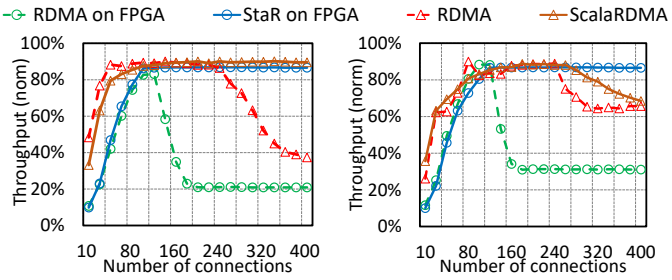
A. Settings

1) Testbed environment: We build a small cluster, which includes 9 physical machines. Each machine has two sockets of Intel Xeon E5-2650 CPU and 64GB memory, and equipped with a 10Gbps StaR NIC (FPGA board) and a 10Gbps Mellanox ConnectX-3 NIC. All NICs are connected to an Arista 7050S-64 switch with 64 10Gbps ports.

2) Methods compared: Besides StaR, three other methods are compared: a) normal RDMA implemented on our FPGA board (denoted as RDMA on FPGA), b) normal RDMA using Mellanox ConnectX-3 NIC, which has the same 10Gbps port speed as our FPGA board (denoted as RDMA), c) ScalaRDMA [8] using Mellanox ConnectX-3 NIC[9].

For both StaR and RDMA on FPGA, each connection consumes 256 bytes of states, which is the same as Mellanox ConnectX-3 NIC [29]. Based on the on-chip SRAM capacity of our FPGA and considering the clock frequency after fitting, we set the total capacity of the state buffer on StaR (stateful side) and RDMA FPGA NIC to be 30KB, which can hold the states of 120 connections on-chip. For ScalaRDMA, we set the group capacity to be 240 connections (which is the measured on-chip state buffer size of Mellanox CX3 NIC). A new group is only created after the previous groups are full. The timeslot for serving a group is 20us, and when the timeslot is over, ScalaRDMA will switch to another group and serve connections in it.

[9]We implement a simplified ScalaRDMA according to the paper [8], with the help of discussion with the paper authors.



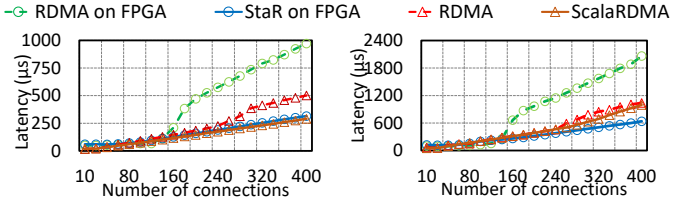
(a) RPC (b) Distributed Machine Learning

Fig. 10. Throughput under two workloads as the number of concurrent connections varies.

3) Workloads evaluated: We mimic two application workloads: a) simple RPC and b) distributed machine learning. In simple RPC, 8 machines work as clients, each posting three SENDs to send three 32-byte data packets to another server machine (mimic remote call). The server immediately posts a SEND to send a 32-byte data packet back to a client (mimic return value) after receiving all the three packets from it. Each client continuously repeat above remote call again after receiving the return value from the server. In distributed machine learning, the packet transmission process is the same as that in RPC. However, all clients are required to be synchronized, *i.e.*, the server will simultaneously return the response packets to all clients after receiving all clients' three packets, thus to mimic the training process in synchronous Parameter-Server architecture [11]. In StaR, the server works fixedly in stateless mode and all the clients work fixedly in stateful mode. We vary the number of concurrent connections by running multiple connections between a client and the server. To avoid the interference of CPU, we ensure that the overall CPU utilization is low both at the server and all the clients, under all the evaluated concurrency degrees.

B. Throughput

Figure 10 shows the throughput (normalized to link bandwidth) under both RPC and distributed machine learning workloads. For both RDMA and RDMA on FPGA, the throughput drops when the number of connections exceeds the on-chip memory capacity (∼240 for CX3 NIC and 120 for the FPGA), due to state miss. On the contrary, as the concurrency degree grows, StaR reaches the maximum throughput and remains at the peak, not suffering from the scalability issue. When the number of concurrent connections grows, StaR has a throughput ∼4.13x and ∼2.2x higher than RDMA on FPGA and RDMA, under RPC workload, and ∼2.9x and ∼1.4x higher than RDMA on FPGA and RDMA, under distributed machine learning workload, respectively. When the concurrency degree is low, StaR performs inferior because the extra RTT incurred for delivering WQEP/CQEP packets. Moreover, our FGPA implementation is not as efficient as the commercial RNIC, leading to inferior performance of StaR and RDMA on FPGA under low concurrency. Under such case, user can explicitly choose normal RDMA or the StaR NIC can automatically



(a) RPC (b) Distributed Machine Learning

Fig. 11. Throughput under two workloads as the number of concurrent connections varies.

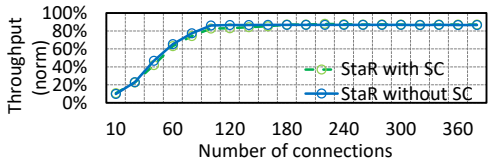


Fig. 12. Security check module overhead

negotiate to normal RDMA mode as introduced in §III-C1 (auto-negotiation is not enabled in the experiment).

ScalaRDMA can keep the peak performance as StaR under simple RPC workload (Fig. 10(a)). This is because connections are not related to each other, so the connections in one working group are enough to saturate the maximum bandwidth. However, switching between groups incur CPU overhead, which may cause throughput degradation in network with higher link bandwidth (not evaluated due to the lack of enough hardware). Nevertheless, when there is dependency among connections, the grouping mechanism in ScalaRDMA will hurt the performance. As shown in Fig. 10(b), under distributed machine learning workload, the throughput of ScalaRDMA begins to drop when the number of connections exceeds 260, and remains only less than 75% of StaR under 400 connections. The reason is that it is difficult for ScalaRDMA middleware to estimate the exact time for each group to complete operations. Particularly, a timeslot too long may cause unnecessary waiting (connections in other groups which have packets to send need to wait for connections in the working group which have no packets to send), and a timeslot too short may incur excessive switching overhead (on-chip connection state update, thread scheduling, *etc.*), both impairing the performance. StaR is not limited to the application patterns, keeping superior performance under both scenarios.

C. Latency

Fig. 11(a) shows the average latency for a client finishing a remote call (3 data packets) and receiving the return value from the server (1 data packet) under both simple RPC and distributed machine learning workloads. Similar to the throughput, when the number of connections exceeds the on-chip memory capacity, the latency of RDMA and RDMA on FPGA rapidly grows due to state miss. However, the latency of StaR grows slowly (almost linearly) as the concurrency degree grows, since state miss has not occurred. At the highest concurrency level in our evaluation, RDMA and RDMA on FPGA have latency ∼1.5x and ∼2.9x higher than StaR under RPC workload, and ∼1.6x and ∼3.2x higher than StaR under distributed machine learning workload, respectively. As introduced before, ScalaRDMA performs well under simple RPC workload. However, ScalaRDMA's performance is inferior under distributed machine learning workload due to group scheduling overhead, which has latency ∼1.5x higher than StaR when there are 400 concurrent connections.

D. Overhead of security check

We evaluate the performance overhead of security check by intentionally installing and removing the security module in StaR, respectively. Fig. 12 shows the throughput of StaR under simple RPC workload, with and without the security check (SC) module. Each connection generates a whitelist rule in the security registry table in the stateful side's NIC (only one PD and one MR is used for data transmission in each connection). Since looking up entries in on-chip SRAM is very fast, the performance overhead is low. Fig. 12 shows that the throughput are almost the same for StaR with and without security check module.

VI. Conclusion

The RDMA scalability problem caused by the limited on-chip NIC memory has bothered the community for several years. Although previous works try to mitigate or avoid the problem by limiting the RDMA usage pattern, the problem remains unsolved. In this paper, we propose StaR that totally removes the RDMA NIC memory constraint which causes the scalability issue. By moving states to the other communication end, StaR makes the RDMA NIC on the bottleneck side totally stateless. As such, it can process RDMA packets fast enough even when there is a large number of concurrent connections. We have implemented StaR on a 10Gbps FPGA board and evaluated its performance on a 9-machine testbed. Results show that StaR can significantly improve the performance under high concurrency scenarios, reaching up to 4.13x and 1.35x throughput compared to the original RDMA NIC and the latest software-based solution, respectively. To the best of our knowledge, StaR is the first work that fundamentally solves the RDMA scalability problem by making the bottleneck RDMA NIC stateless.

Acknowledgement

We thank Peng Cheng and Yongqiang Xiong for their inspiring discussion on the idea, and our shepherd Ang Chen and the anonymous reviewers for their valuable comments and suggestions on improving this paper. This research was partially funded by the National Natural Science Foundation of China (No.61872132), the Natural Science Foundation of Hunan Province for Excellent Young Scholars, the Training Program for Excellent Young Innovators of Changsha, the Fundamental Research Funds for the Central Universities, and Huawei.

REFERENCES

[1] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 523–536. ACM, 2015.

[2] 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.

[3] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 44–58. 2019.

[4] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25:1223–1231, 2012.

[5] 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.

[6] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[7] 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.

[8] 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.

[9] 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.

[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] 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.

[12] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. volume 45, pages 139–152. ACM, 2015.

[13] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM Computer Communication Review*, 45(4):183–197, 2015.

[14] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and Mitigating Packet Corruption in Data Center Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan M G Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *acm special interest group on data communication*, 45(4):537–550, 2015.

[16] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 313–326, 2018.

[17] *InfiniBand architecture volume 1, general specifications, release 1.2.1*. InfiniBand Trade Association, 2008.

[18] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 437–450, 2016.

[19] 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.

[20] Dian Shen, Junzhou Luo, Fang Dong, Xiaolin Guo, Kai Wang, and John CS Lui. Distributed and optimal rdma resource scheduling in shared data center networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 606–615. IEEE, 2020.

[21] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[22] 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.

[23] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.

[24] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *ACM Transactions on Database Systems (TODS)*, 44(4):1–45, 2019.

[25] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655*, 2016.

[26] 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.

[27] Pulin Pan, Guo Chen, Xizheng Wang, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. Towards stateless rnic for data center networks. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 57–63, 2019.

[28] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. Redmark: Bypassing {RDMA} security mechanisms. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[29] Mellanox Adapters Programmer's Reference Manual (PRM). https://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf.