csRNA: Connection-Scalable RDMA NIC Architecture in Datacenter Environment

Ning Kang[1,2], Zhan Wang[1], Fan Yang[1], Xiaoxiao Ma[1,2],
Zhenlong Ma[1,2], Guojun Yuan[1], and Guangming Tan[1]
1 Institute of Computing Technology, CAS
2 University of Chinese Academy of Sciences
Beijing, China
kangning18z@ict.ac.cn, wangzhan@ncic.ac.cn, yangfan2020@ict.ac.cn,
{maxiaoxiao, mazhenlong, yuanguojun, tgm}@ncic.ac.cn

***Abstract*—RDMA has been widely deployed in datacenter networking as an ideal optimization strategy in recent years. Due to its mechanisms such as kernel bypass and hardware offloading, RDMA is expected to offer better performance than traditional kernel-based TCP/IP networking. However, the hardware offloading in RDMA requires the RDMA Network Interface Card (RNIC) to manage the connection metadata, and the limited on-chip memory size in RNIC leads to its limited connection scalability. When the RNIC maintains a large number of connections, its performance drops dramatically.**

This paper first finds that the head-of-line blocking in connection metadata management is a major factor affecting RNIC scalability. Based on the findings, we propose csRNA, a connection-scalable RNIC architecture that maintains near-peak performance when connection scales. To achieve the non-blocking RNIC processing path, csRNA utilizes a non-blocking connection scheduler to schedule different connections when blocking. Furthermore, using a non-blocking connection management model, csRNA departs from the conventional RNIC design by returning the prepared connections first. csRNA effectively avoids the performance degradation caused by the head-of-line blocking of connection metadata management when the number of connections increases. We implement and evaluate csRNA and demonstrate that with less on-chip memory occupancy, csRNA could still maintain near-peak performance when scaling up to more than 15,000 connections.

***Index Terms*—Architecture Design, Network Interface Card (NIC), Remote Direct Memory Access (RDMA), Scalability Problem**

I. Introduction

Remote Direct Memory Access (RDMA) has been widely deployed in High-Performance Computing (HPC) environments. In recent years, lots of cloud service providers, such as Microsoft, and Amazon [1], [2], [16], [17], have turned to RDMA to optimize their datacenter networking, acquiring higher throughput and lower latency than traditional TCP/IP networking. RDMA offloads plenty of network tasks into the RDMA Network Interface Card (RNIC), releasing more CPU resources for other data center applications.

While offloading network tasks into the RNIC improves network processing performance, it also introduces some scalability issues. Due to the hardware offloading, the RNIC is required to handle the network connection. Therefore, the connection metadata also needs to be stored in the RNIC to respond to communication requests in time. However, the limited on-chip memory size in the RNIC limits the number of active connections that can be supported for RNIC. As a result, the RNIC can only maintain a limited number of connections.

Recent works [3]–[5] propose their open-sourced RNIC designs. However, they only support a limited number of connections. For example, eRNIC [4] is an embedded RNIC designed by Xilinx, and its connection metadata entries are all stored in RNIC, supporting 127 connection entries at most. The poor on-chip memory greatly limits its connection scalability. On the other hand, commercial RNIC also faces a scalability problem. ConnectX series RNICs deploy a cache for connection metadata storage, which helps it supports up to 2^{16} connections [6]. But these RNICs are closed-sourced, and their design details are unknown to us. Besides, the latest RNIC ConnectX-6 only supports no more than 450 connections to maintain peak performance [13].

In this work, we introduce an RNIC design with high connection scalability. We deploy a connection metadata cache in RNIC to temporarily store connection metadata used in the communication. When the cache misses, RNIC fetches the missing metadata from host memory. In this way, the number of connections is not constrained by RNIC's small on-chip memory, and the RNIC could support more connections without involving on-chip memory incremental.

Though the intuition is simple, deploying the connection metadata cache in RNIC still faces challenges. First, when the cache in RNIC misses, subsequent connection metadata access request is unable to be processed until the previous missed metadata is returned, which causes head-of-line blocking. The head-of-line blocking during connection management causes the entire processing path in RNIC to be blocked, resulting in the peak performance degradation as the number of connections increases. Second, because we need to support a scalable number of connections, we cannot deploy multiple request queues to prevent head-of-line blocking. Therefore, how to design a non-blocking connection management module is also a problem to be solved. Third, the Infiniband specification [7] requires that the processing of the same connection be performed sequentially, otherwise, a semantic error will occur. This requirement indicates that in some cases the blocking

must be guaranteed, which further complicates the problem.

Based on the challenges above, we propose csRNA, a connection-scalable RNIC architecture design that effectively avoids performance degradation as the number of connections increases. First, csRNA includes a non-blocking scheduler that first schedules the prepared connections. Second, csRNA provides a non-blocking connection management module that directly processes the subsequent hit connection request when the present request misses. Third, the connection management module also checks dependencies between connection requests to ensure semantically correctness in the case of non-blocking processing requests. Experimental results show that with less on-chip memory occupancy, csRNA could still maintain near-peak performance when connection scales to over 15,000.

In summary, the main contributions of our work are:

1. We outline csRNA [8], an open-sourced connection-scalable RNIC architecture that supports a scalable number of connections. csRNA utilizes a cache to store connection metadata, and places all the metadata to host memory.

2. We explain why performance degrades as the connection scales. We find that it is the head-of-line blocking problem that causes RNIC's performance degradation as the number of connections increases.

3. We describe a non-blocking RNIC design. The design includes an non-blocking scheduler and a connection management module, which avoids head-of-line blocking when connection metadata cache in RNIC misses.

4. We implement and evaluate csRNA in gem5 to trace its performance gains. Experimental analysis of our proposed csRNA architecture shows that with less on-chip memory occupancy, csRNA could still maintain near-peak performance when connection scales.

II. Background and Related Works

A. RDMA Basics

RDMA NIC (RNIC). RNIC is a network interface card that supports RDMA transmission. RDMA offloads the transport layer onto the hardware to reduce CPU overhead while achieving low latency data transmission. This leaves the RDMA connection maintenance to be done by RNIC. User applications in different nodes exchange the connection metadata during connection establishment and then write the metadata to RNIC, which maintains it ever since.

Transport operation. RDMA supports two kinds of the transport operation, namely two-sided and one-sided operation. The two-sided operation, including send/recv, requires explicit involvement of the remote CPU to complete the data transmission. While one-sided operation allows local applications directly access remote memory, without the involvement of the remote node's CPU. Typical one-sided RDMA operation includes RDMA Read and RDMA Write, which implements direct remote memory reading and direct remote memory writing, respectively. They present their lower latency because of the less involvement of remote CPU.

Service type. RDMA also provides different transport services for applications. Specifically, there are two service modes in RDMA: reliable connection (RC), and unreliable datagram (UD). RC is the only service type that fully supports all of the one-sided operations. Recently, because of the low latency of one-sided operation, RC has received much popularity [1], [2], [14] in datacenter deployment. Thus, we focus on the RC service type in this paper.

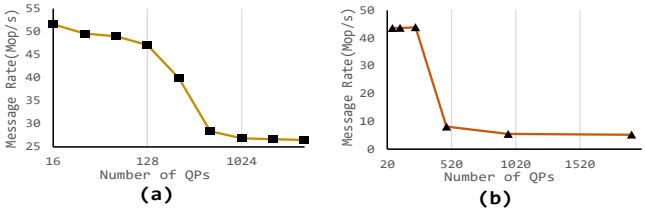

Fig. 1. Message rate changes as the number of connections increases. (a) Message rate results under commercial RNIC (Mellanox ConnectX-4). (b) Message rate results under first-come-first-served policy (gem5).

Queue pair (QP) & queue pair context (QPC). RDMA establishes connections between communication nodes creating by queue pairs (QPs). A QP consists of a send queue (SQ) and a receive queue (RQ), storing sending and receiving descriptors. The descriptors are the indicator of the transmission, including the address and length of data. RNIC uses descriptors to guide the sending or receiving of the data. QP itself also has many attributes like address and length of the QP. As a result, connection metadata is created to hold these attributes. Connection metadata is also called QP context (QPC). A typical commercial RNIC ConnectX-4 requires 256 bytes of metadata to be maintained for one RDMA connection [6], [19].

B. RNIC Connection Scalability Problem in Datacenters

Data centers have high requirements for RDMA connection scalability. Unlike HPC, asymmetric communication often occurs in data center networks. These communication patterns are generally in-cast or out-cast, that is, one node (server) is connected to multiple other nodes (clients). In these networks, the server nodes have to maintain thousands of connections with the clients. For example, in Timestamp Oracle [12], the server nodes need to maintain more than 2800 concurrent connections. With the large-scale deployment of RDMA in data centers, the demand for RDMA's connection scalability capabilities has grown.

However, RDMA's hardware offloading limits its connection scalability. To maintain lower CPU usage and higher packet throughput, RDMA offloads the transport layer onto the RNIC. Therefore, RNIC not only needs to manage the data transmission but also requires maintaining the connection metadata. On the other hand, Infiniband specification [7] states that the maximum number of QPCs is 2^{24}, indicating that the RNIC needs $2^{24} * 256$ bytes = 4GB of space to store QPC. The limited on-chip memory size in RNIC is unable to hold this metadata, limiting the number of connections that the RNIC can manage and limiting its scalability.

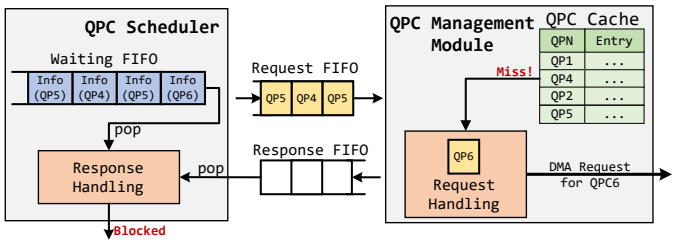

Fig. 2. First-come-first-served connection management mechanism in RNIC. QPC scheduler is a module in RNIC processing path that initiates QPC requests. The waiting FIFO stores the information related to a QPC request (Info), and the contents listed in brackets are the QPC to which the information belongs.

To illustrate the impact of increasing the number of connections on performance, we experiment on Mellanox ConnectX-4 RNIC. We use 4 nodes in the test, specifically, one node as the server node and the other three as the client nodes. The server sends 64 bytes of RDMA Write requests concurrently to the three clients. Fig. 1(a) shows the change in message rate as the number of connections grows. The message rate of the RNIC starts to decline significantly when the number of connections in the server is greater than 256.

C. Related Works

Previous works related to RNIC design lack consideration of the RDMA connection scalability problem. StRoM [3] presents an RNIC supporting the deployment of processing kernels. StRoM stores all of the connection information in on-chip memory, constraining its scalability. 1RMA [22] rearchitects the remote memory access, eliminating the process of establishing connections in data transmission. In comparison, our work tries to improve the connection scalability of the RNIC without changing the semantics of standard RDMA.

III. Head-of-line Blocking Problem in Connection Management

Existing open-sourced RNIC designs [3], [4] store all the QPC in the RNIC chip, so the QPC management module works well even in the first-come-first-served request processing mode. As is shown in Fig. 2, the existing RNIC design processes the QPC request through a request fifo. The fifo receives the QPC access request and pops it to the QPC management module to handle the QPC request. In the meantime, the requester who posts the QPC request also maintains a waiting fifo to temporarily store the information related to the requested QPC. Since the processing of the QPC management module is sequential, when it returns the QPC response, the requester only needs to get the corresponding information from the top of the waiting fifo to start the next stage of processing. This works well for a limited number of QPCs because all of the QPCs sit entirely in the RNIC and each QPC has the same request processing time.

However, based on our analysis, we find that simply deploying the QPC cache in RNIC to increase the supported number of connections would cause head-of-line blocking. According to the Infiniband specification [7], the processing logic within the same QP must be sequential, while there is no sequential requirement between different QPs. If the QPC cache is used, subsequent QPC requests may be blocked by previous QPC requests because the processing time of each QPC request is different. For example, as is shown in Fig. 2, the QPC scheduler posts four QPC requests, requests for QP6, QP5, QP4, and QP5, sequentially processed by the QPC management module. Because QP6 misses, the QPC management module cannot process the subsequent request until the connection metadata of QP6 returns. Although the connection metadata of QP5 and QP4 are stored in the QPC cache, their processing is still blocked until the connection metadata of QP6 is returned. In this procedure, the missing QP6 results in head-of-line blocking, and the entire processing path is also blocked.

To better illustrate this problem, we design and test the first-come-first-served QP connection management strategy as shown in Fig. 2. The experiment is conducted in the gem5 [20] simulator. The size of QPC cache in RNIC is 300 QPC entries. One server sends RDMA Write to the other three client nodes. The result is shown in Fig. 1(b). When the number of connections slightly exceeds the capacity of the cache, the message rate starts to decline sharply. The experiment shows that, when the number of connections grows, the head-of-line blocking can greatly affect RNIC performance.

One solution to the above head-of-line blocking problem is to design an RNIC that supports non-blocking processing, which leads to the following two challenges:

1). How to handle communication requests between different connections without blocking? In previous designs [3], [4], QPC access is done sequentially. Therefore, the information related to QPC requests is simply stored through a waiting fifo. When the QPC response returns, it fetches the related information from the top of the waiting fifo. When RNIC supports non-blocking processing, requests are no longer being processed in first-come-first-served order. How to find the related information according to the returned QPC, to carry out the subsequent non-blocking transmission processing, has become a problem to be solved.

2). How to ensure order dependencies of QPC requests when the requests are returned out of order? The Infiniband specification [7] requires order-preserving for different communication requests in the same QP. When the QPC management module supports out-of-order processing, the order of the QPC response returned is unpredictable. In this case, for different QPC requests of the same QP, subsequent QPC requests may be returned earlier than the previous QPC requests, resulting in subsequent processing sequence disorder and even errors in upper-layer application logic. Therefore, the QP order dependence is also a problem that the QPC management module needs to consider.

IV. Connection-Scalable RNIC Architecture

In the next two sections, we will tackle the challenges raised in Section III. First, we present csRNA, a connection-scalable

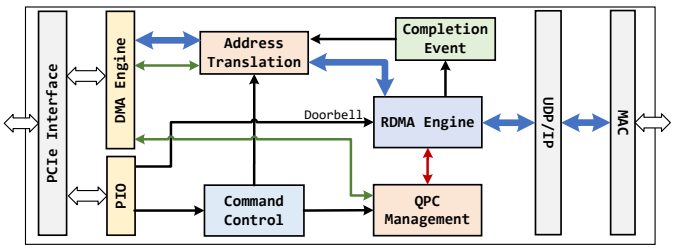

Fig. 3. Overall csRNA Architecture. Blue arrow is data path. Red arrow is QPC request and response. Green arrow is cache miss caused DMA request and response.

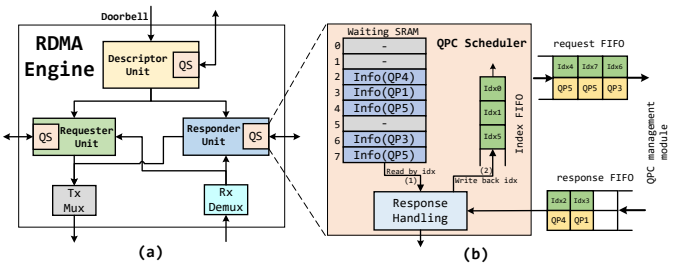

Fig. 4. Non-blocking supports for RDMA Engine. (a) Architecture of RDMA Engine. QS stands for non-blocking QPC scheduler. (b) QPC scheduler that supports non-blocking QPC scheduling.

RNIC architecture that supports a large number of active connections in the datacenter environment (Section IV). csRNA utilizes a non-blocking QPC scheduler to the processing path in the RNIC to schedule the prepared connections first. This scheduler enables the RNIC to process communication requests for different connections without blocking. Second, we propose a non-blocking QPC management module in csRNA (Section V). The module processes the QPC access requests without blocking and resolves the order dependencies between different QPC requests.

In this section, we first introduce the overall design of csRNA non-blocking processing, followed by a detailed description of the QPC scheduler in the RDMA engine.

A. High-level Overview

The architecture of our csRNA is split into 7 main components which are visualized in Fig. 3. Command control module receives the RNIC configuration commands from the host CPU and forwards them to other modules. These commands include initializing the RNIC device, delivering the host memory space allocated for the RNIC, and configuring QPC. QPC management module maintains the RNIC connection metadata configured by the host CPU for data transmission. The module includes a cache to store connection metadata. The cache is called the QPC cache. When cache misses, the module fetches the corresponding QPC from host memory. The detailed description of the QPC Management Module is covered in Section. V. Address translation module is used to translate the virtual address used in RDMA transmission into a physical address and forward it to the DMA module to access host memory. Similar to QPC management module, Address Translation Module also uses a cache to store its page table. RDMA engine handles transmission requests and their responses under RDMA semantics. The RDMA request types currently supported by our design include RDMA write and RDMA read. The RDMA engine starts the transmission task when it receives a doorbell. The doorbell is a 64-bit memory write delivered by the host CPU. This command instructs the RDMA engine to process the request. On the response side, when RNIC receives a packet from the UDP/IP and link-layer (MAC), the RDMA engine determines the QP connection to which the packet belongs based on the information in the packet. The RDMA Engine then uses the QPC to process the received packet. Completion event module, receiving completion message from RDMA engine, notifies host CPU that the processing is complete. DMA module and PIO module, which are connected to PCIe controller, handle access to host memory and receive requests from the host CPU, respectively.

B. Non-blocking Supports for RDMA Engine

In this section, we introduce the detailed design of the RDMA Engine and explain its implementation of non-blocking execution. Fig. 4(a) shows the structure of RDMA engine. Descriptor unit manages the descriptors in the RDMA engine. These descriptors include send descriptors and receive descriptors, which store information to describe the location of the data to be sent or the memory space of the data to be stored at the receiving end. Requester unit handles the communication request, including memory write or read requests generation, sending window maintenance, packet response processing, and packet retransmission. Responder unit processes incoming data read or write requests from the remote end and generates corresponding response packets back to the wire. Tx Mux and Rx Demux are the arbiter and distributor, respectively.

In order to implements its non-blocking processing, RDMA engine deploys a non-blocking QPC scheduler on its processing path. As is shown in 4(a), three units in RDMA engine is equip with QPC scheduler. The structure of the QPC scheduler is shown in Fig. 4(b). Index FIFO stores consecutive, non-repeating numbers as a unique index for a QPC request. We name the number of indexes in the index FIFO as out-of-order capacity. Note that in our implementation the width of index is 8 bits. Waiting SRAM is a block of on-chip memory that stores some information related to QPC requests. This information contains different contents in different units. In the descriptor unit, the information contains the doorbells posted from the host CPU; In the responder unit, the information includes incoming packets received from the wire. And in requester unit's QS unit, the information contains the descriptor acquired from the descriptor unit. Response handling accepts the returned QPC response and forwards the QPC entry along with the Info content to the subsequent processing module.

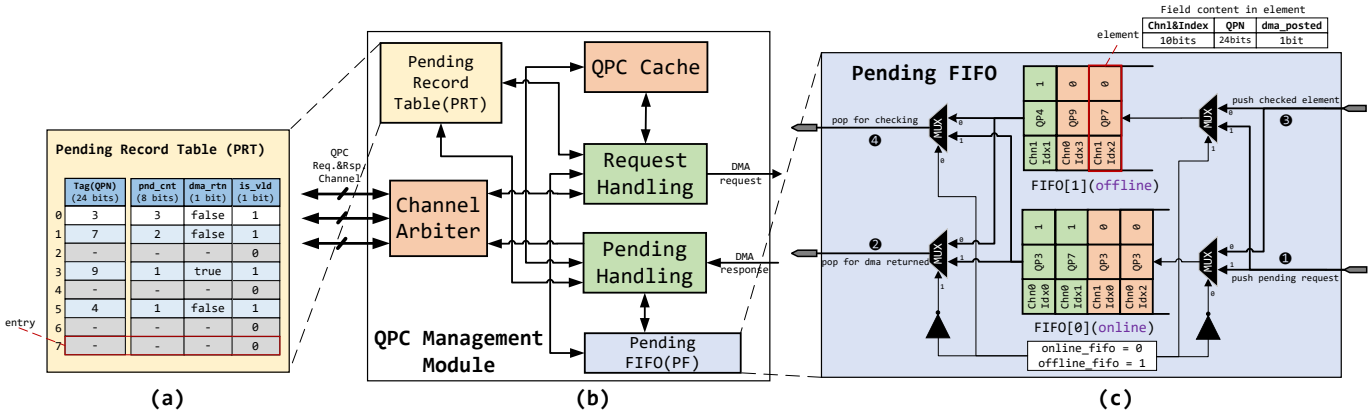

Fig. 5. Non-blocking QPC management module. (a) The structure of pending record table (PRT). (b) The overall architecture of QPC management module that solves the problem of head-of-line blocking. (c) The structure of pending FIFO (PF).

Here is an example of how the QPC scheduler supports non-blocking QPC scheduling. In the initial state, the FIFO stores consecutive numbers starting from 0 as $index$. When a QPC request arrives, the QPC scheduler first acquires an $index$ from the head of the Index FIFO and writes the related information of the QPC request to Waiting SRAM with that $index$ as the address. Next, QPC scheduler pushes the QPC request, along with the retrieved index, to the QPC management module through the request FIFO. After the QPC management module returns the QPC through response FIFO, response handling returns the related information of QPC from the corresponding address in the Waiting SRAM according to the $index$ in QPC response (channel (1) in Fig. 4), and forwards it together with the QPC response to the subsequent processing module. Meanwhile, response handling pushes the $index$ back to the index FIFO for subsequent QPC requests (channel (2) in Fig. 4(b)). By designing a QPC scheduler unit, the RDMA engine processes transmission tasks of different connections without blocking, avoiding the blocking of RDMA transmission processing due to the QPC cache missing.

V. Non-blocking QPC Management Module

In this section, we introduce the design details of the QPC management module, which effectively avoids the problem of head-of-line-blocking in QPC request processing. As mentioned in Section III, the non-blocking QPC management module not only needs to allow out-of-order processing for different QPC requests but also needs to ensure that the QPC management module can maintain the order for the QPC requests of the same QPs.

We design a non-blocking QPC management module that satisfies all of the requirements above. Fig. 5(b) gives the overall structure of the module. Pending FIFO is a unit to hold pending QPC requests that are blocked due to QPC cache missing or due to sequencing requirements. Pending record table records the temporary status of requests in pending FIFO, which is used to determine whether the pending FIFO contains QPC requests for a certain QP, that is, whether the QPC requests need to be sequenced. The request handling unit is used to handle QPC requests from QPC scheduler. Pending handling unit handles QPC requests stored in the pending FIFO. The channel arbiter unit arbitrates requests from multiple QPC schedulers in the RDMA engine and forwards QPC responses back.

A. Pending FIFO (PF)

Pending FIFO is used to store QPC requests that are waiting to be processed. These requests can be classified into two types:

- *Type 1* - QPC cache missing caused QPC requests blocking;
- *Type 2* - order-preserving for the same QP caused QPC request blocking;

To distinguish the two types of requests, pending FIFO adds a 1-bit field dma_posted in the element. As shown in Fig. 5(c), if dma_posted field in an element is 1, QPC request is pending due to QPC cache misses. And the request related to the element has also initiated a read request via DMA. If dma_posted field is 0, it means that the element is blocked due to the order preservation for one QP, and that the request does not make a DMA read request on the requested QPC entry. In addition, an element also contains fields such as QPN, channel number, and index, which are utilized in subsequent processing.

Next, we analyze the differences between the two types of QPC requests. For *Type 1* requests, pending FIFO only needs to ensure that it gets the corresponding element from the header of the FIFO when the response returns. For *Type 2* requests, instead of waiting for a DMA response, it checks whether the QPC requests meet the conditions for processing during idle time when there is no DMA response.

Based on the analysis above, we build the pending FIFO as in Fig. 5(c). The unit contains two sets of interfaces: ❶, ❷ for normal request submission and response processing; ❸ and ❹ are used for checking and processing *Type 2* requests. In addition, pending FIFO contains two FIFOs of the same size to store pending elements and two registers

Algorithm 1 Finite State Machine for Request Handling Unit

```
1: if Incoming QPC request for qpn then
2:    Read qpn entry in PRT;
3:    if Find qpn in PRT then
4:       Push QPC request to PF;
5:       Update entry in PRT with pnd_cnt++;
6:    else if QPC cache hit for qpn then
7:       Read or update the entry in QPC cache;
8:       Return QPC content if this is a read request;
9:    else
10:      Post DMA read for QPC of qpn;
11:      Push QPC request to pending FIFO;
12:      Write an entry for qpn with pnd_cnt=1;
13:   end if
14: end if
```

Algorithm 2 Finite State Machine for Pending Handling Unit

```
1: if QPC DMA response for qpn returns or start_check! =
   0 then
2:    if QPC DMA response for qpn returns then
3:       Pop elem from PT;
4:       if The elem has posted DMA request before then
5:          Read $ update entry of qpn from PRT;
6:          Return QPC & Update start_check & Return;
7:       end if
8:    else
9:       Get one elem from PF for check;
10:      Return if elem has posted DMA request;
11:   end if
12:   Get entry related to elem from PRT & pnd_cnt − −;
13:   if QPC cache hit for this elem then
14:      Update entry in PRT & Return QPC;
15:   else if dma_rtn == true then
16:      dma_rtn = false & Push elem to PF;
17:   else
18:      Push checked elem back;
19:   end if
20:   Update start_check;
21: end if
```

of $online\_fifo$ and $offline\_fifo$ to indicate the type of the two FIFOs. For example, in Fig. 5(c), the values of $online\_fifo$ and $offline\_fifo$ are 0 and 1 respectively, indicating that FIFO[0] is online and FIFO[1] is offline. The output port of the FIFO marked online is connected to ❷ to handle elements of *Type 1*; The input port is connected to ❸ to write back into the FIFO an element that failed to be checked. Check failed means the element previously popped from ❹ is not of *Type 2* or still has a dependency on the same QP. The output port of the FIFO marked offline connects to ❹ to pop the element to be checked; while the input port connects to ❶ to receive the new element. The number of elements in PF is $2 * 3 * ooo\_cap$, where ooo_cap is the out-of-order capacity mentioned in Section IV-B.

B. Pending Record Table (PRT)

Pending record table is used to record QPC request processing of the same QP in pending FIFO. The main purpose of this table is to ensure the order of QPC requests for the same QP. A PRT is essentially a content address memory (CAM) that uses tag fields (QPN in this case) to find matching item contents. There are three fields in an entry. is_vld marks whether the entry is valid or not; pnd_cnt records how many QPC requests for the QP are Pending in the Pending FIFO; dma_rtn indicates that the QPC corresponding to the QP has been stored in the QPC cache, and the subsequent QPC requests corresponding to the QP in the pending FIFO can be directly obtained from the QPC cache. The number of entries in PRT is $3 * ooo\_cap$, where ooo_cap is the out-of-order capacity mentioned in Section IV-B.

During the execution of the QPC management module, both the request handling unit and the pending handling unit access the content of the entry in the PRT. For read request or update request, PRT directly uses QPN to find the corresponding entry; For write entry requests, PRT use the Formula 1 to calculate the address of the entry to be written:

$$address = chnl\_num * ooo\_cap + index \tag{1}$$

, where $chnl\_num$ is the number of request channel in QPC scheduler; $index$ is the unique number given by QPC scheduler to identify the request; ooo_cap is the out-of-order capacity. For example, if the QPC management module receives a QPC request for QP3 with index 0 from QPC scheduler with Channel 0, the address for the PRT entry created by the request is 0.

C. Request Handling

The request handling unit is used to handle QPC requests and write the requests back to channel arbiter or submit them to pending FIFO. After receiving a QPC request, the request handling unit first checks whether the pending record table contains requests waiting for the QPC to ensure sequential QPC access to the same QP. If the QPC is not included in the pending record table, it indicates that the request does not need to consider the problem of sequence preservation. In this case, the QPC is directly accessed from the QPC cache. Hitting requests are written directly back to QPC scheduler. For missed requests, a QPC read request is issued and the pending FIFO and pending record table are updated accordingly. If the pending record table contains the QPC, it indicates that the order preservation of the same QP needs to be considered for the request. The request handling unit writes the request to the pending FIFO and updates the information about the relevant entries in the pending record table.

D. Pending Handling

The pending handling unit is used to handle QPC requests located in the pending FIFO. According to Section V-A, the pending FIFO stores two types of blocked QPC requests. For

requests of *Type 1*, the unit simply waits for the DMA response to return. Since the DMA response of QPC is returned sequentially, the pending handling unit needs to ensure that the QPC request of this type is popped sequentially from the pending FIFO to correspond to the DMA response during the processing of this request type.

For *Type 2* requests, due to QP dependencies, we also need to ensure the order in which requests of this type are processed for the same QP. QPC requests between different QPs can still be out-of-order. To determine and process QPC requests that are independent as quickly as possible, the pending handling unit traverses the elements in the pending FIFO in idle time when no DMA response is back. During the traversal, the pending handling unit first accesses the pending record table to determine whether the request is preceded by a pending QPC request for the same QP. If there is still a predecessor for this QP, skip the checked QPC request and push it back to the pending FIFO. If not, the QP dependency for the request has been removed, and pending handling unit accesses the QPC cache to read the relevant QPC entry. For the hit QPC request, the response returns, and the relevant information in the pending FIFO and pending record table is updated. For a missed QPC request, the request is transformed into *Type 1* QPC request. The related QPC DMA read request is also submitted, and the QPC request is again pushed to the pending FIFO.

E. On-chip Memory Occupancy

Based on the analysis in Section. IV-B and V, we get the relationship between the incremental of on-chip memory and out-of-order capacity under the out-of-order strategy:

$$\begin{aligned} mem_T &= 3 * mem_{QS} + mem_{PF} + mem_{PRT} \\ &= 40 * ooo\_cap\ (bytes) \end{aligned} \tag{2}$$

, where mem_{QS}, mem_{PF}, and mem_{PRT} are the additional on-chip memory occupancy for QPC scheduler, pending FIFO, and pending record table, respectively; ooo_cap is the out-of-order capacity. The Formula 2 shows that, when we increase the out-of-order capacity of each channel by one $index$, 40 bytes of on-chip memory would be added in RNIC. The incremental is much smaller than one QPC entry size of 256 bytes.

VI. Evaluation

To fully evaluate our design, we try to answer the following questions:

- Can the non-blocking optimization strategy proposed in csRNA achieve better performance? (in Section VI-B)
- What is the on-chip memory resource usage of csRNA when it acquires performance gains? (in Section VI-C)

A. Settings

Environment Settings. We implement and evaluate csRNA in the gem5 simulator [20], with the RNIC module consisting of about 7000 lines of code. In the experiment, we use one server to directly connect to three clients through a 100Gbps Ethernet switch with a link delay of 1µs. Each node is tested with 10 CPU cores, each with a frequency of 2GHz. We run each CPU under the gem5's System-Call Emulation (SE Mode) to speed up the test execution time. The PCIe bandwidth is set to 128Gbps, which is the same bandwidth as PCIe 3.0 X 16. The Round Trip Time (RTT) delay for PCIe is set to 500ns [9], [21] for accessing host memory. The clock frequency of the RNIC is set to 1GHz. And the QPC cache replacement policy is set to least recently used (LRU). In the experiment, the message size is set to 64 bytes, which is a typical test case in scalability problem [10], [11].

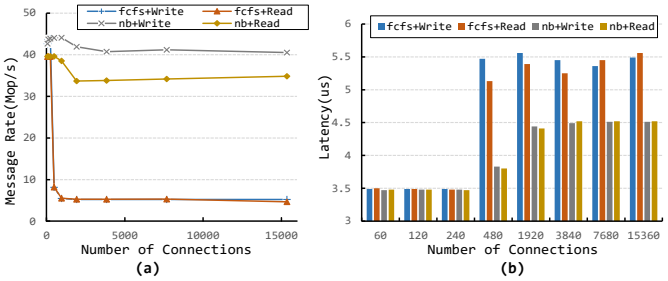

Fig. 6. Connection scalability test. (a) Message rate at different number of QPs. (b) Latency at different number of QPs. fcfs: first-come-first-served QPC management scheme. nb: non-blocking QPC management scheme. Write: RDMA Write transport operation. Read: RDMA Read transport operation.

Software Settings. To run RDMA applications on csRNA, we implement a software stack to provide a verbs interface similar to libibverbs [18], a de facto standard interface for RDMA programming. The software stack includes: 1). Kernel-space driver running on gem5 in SE mode for resource allocation; 2). User-space driver and verbs to interact with RNIC in data path and to provide verbs API; 3). Communication management layer for connection establishment. Our evaluation of csRNA is carried out on this software stack.

B. Connection Scalability Test

First, we evaluate the message rate improvement for csRNA over the first-come-first-served strategy. We test both RDMA Read and RDMA Write transport operations. Under the non-blocking strategy, the size of the out-of-order capacity we use is 32. The QPC Cache has a capacity of 300 entries. In the test, the server establishes the same number of QP connections with each client and continuously sends RDMA communication requests to the client. After receiving the completion event of transmission, the server immediately posts another RDMA communication request to the QP corresponding to the completion event. Fig. 6(a) gives the results of message rate at different number QP connections. The results show that csRNA has better connection scalability under both RDMA Write and RDMA Read transport operations. csRNA eliminates the head-of-line blocking during data transmission, so it achieves near-peak performance message rates even when maintaining over 15,000 active connections.

Next, we evaluate the latency of csRNA. In the test, each process in the server node simultaneously posts a RDMA

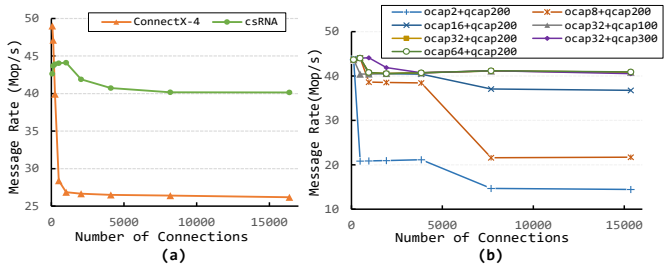

Fig. 7. (a) Message rate comparison between csRNA and commercial RNIC ConnectX-4. The QPC cache capacity in csRNA is 300 QPC entries. (b) Message rate under different out-of-order capacity. ocap: out-of-order capacity. qcap: the number of entries in QPC cache.

request to one of the QP it maintains. When the process receives an RDMA completion event from that QP, it initiates another RDMA request through the next QP. Fig. 6(b) is the changes in transmission latency as the QP connection scales. The latency of first-come first-served strategy (baseline) increases dramatically when the number of connections slightly exceeds the QPC cache capacity. This is because that under the baseline strategy, one communication request may be blocked by the previous missed communication request. On the other hand, when the number of connections increased, the delay of csRNA using the non-blocking strategy also increased slowly, and its latency growth shows certain stability. This is because csRNA eliminates the latency dependence between different communication requests. As the number of connections increases, the access latency of QPC gradually approaches the time to access host memory, which is the QPC access latency when QPC cache miss happens.

In addition, we also compare csRNA with some current works. Fig. 7(a) shows the performance of the commercial RNIC ConnectX-4 and csRNA when connection scales. We test ConnectX-4 under physical system experiment, and csRNA under the gem5 simulator. In order to ensure the fairness of the comparison, we deploy the same test environment as that in gem5 simulation which is mentioned in Section VI-A. Experimental results show that compared with ConnectX-4, csRNA can still achieve near-peak performance as the number of connections increases. Moreover, eRNIC [4] is an open-sourced RNIC provided by Xilinx. eRNIC only supports 127 connections, while csRNA achieves near-peak performance even with more than 15,000 connections.

C. On-chip Memory Usage

To understand the practicability of csRNA, we further test its on-chip memory resource usage. We use the same test as in Section. VI-B to evaluate the performance of csRNA under different out-of-order capacities and QPC cache capacity. The result of the message rate is shown in Fig. 7(b). As the out-of-order capacity increases, the message rate of the RNIC increases when a large number of connections are maintained. This is because as the out-of-order capacity increases, the QPC scheduler could handle more QPC requests. The results show that 32 out-of-order capacity are sufficient for csRNA to achieve near-peak performance during connection scaling, which is about only 1KB on-chip memory occupancy (based on Formula 2). We also test the message rate with 100, 200, and 300 QPC cache entries at the out-of-order capacity of 32. The results show that csRNA can still achieve near-peak performance at 15,000 connections when the QPC cache capacity is 100 QPC entries. eRNIC [4] stores 127 connections on chip, and supports only a maximum of 127 connections. Compared with eRNIC, csRNA consumes less on-chip memory ($100 * 256 + 1KB < 127 * 256$), and maintains near-peak performance even when the connections scales to over 15,000 connections.

VII. Conclusion

RDMA's connection scalability problem caused by limited on-chip memory in RNIC is a well-known challenge. Previous open-sourced works [3], [5], [15] on RNIC design do not consider the connection scalability issues. We propose csRNA that gives an example design of scalable RNIC. csRNA is a connection-scalable RNIC architecture that maintains near-peak performance when connection scales. At its core, we propose an non-blocking QPC scheduler and non-blocking QPC management module in csRNA that solves the head-of-line blocking when QPC cache misses. We implement and evaluate csRNA in the gem5 [20] and the results show that with less on-chip memory occupancy, csRNA could maintain near-peak performance when connection scales to over 15,000.

VIII. Acknowledgement

This work is supported in part by the the National Key Research and Development Program of China (No. 2021YFB0300700), International Partnership Program of Chinese Academy of Sciences (No. 171111KYSB20180011), National Science and Technology Innovation Program 2030 (No.2020AAA0104402), NSFC (No. 61972380), the National key research and development program (No. 2021YFB0300700), the Research and Innovation Program of State Key Lab. of Computer Architecture, ICT, CAS (No. CARCH5407).

References

[1] Zhu, Yibo, et al. "Congestion control for large-scale RDMA deployments." ACM SIGCOMM Computer Communication Review 45.4 (2015): 523-536.
[2] Guo, Chuanxiong, et al. "RDMA over commodity ethernet at scale." Proceedings of the 2016 ACM SIGCOMM Conference. 2016.
[3] Sidler, David, et al. "StRoM: smart remote memory." Proceedings of the Fifteenth European Conference on Computer Systems. 2020.
[4] Xilinx. 2022. ERNIC. Website. https://www.xilinx.com/products/intellectual-property/ef-di-ernic.html
[5] Schelten, Niklas, et al. "A High-Throughput, Resource-Efficient Implementation of the RoCEv2 Remote DMA Protocol for Network-Attached Hardware Accelerators." 2020 International Conference on Field-Programmable Technology (ICFPT). IEEE, 2020.
[6] NVIDIA. 2022. Mellanox Adapters Programmer's Reference Manual (PRM). Website. https://network.nvidia.com/related-docs/user_manuals/Ethernet_ Adapters_Programming_Manual.pdf.
[7] Infiniband. 2022. Infiniband Architecture Specification, Volume 1. Website. https://cw.infinibandta.org/document/dl/8567.
[8] NCSG Group. 2022. csRNA. https://github.com/ncsg-group/csRNA.

[9] Neugebauer, Rolf, et al. "Understanding PCIe performance for end host networking." Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. 2018.
[10] Monga, Sumit Kumar, Sanidhya Kashyap, and Changwoo Min. "Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock." Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021.
[11] Chen, Youmin, Youyou Lu, and Jiwu Shu. "Scalable RDMA RPC on reliable connection with efficient resource sharing." Proceedings of the Fourteenth EuroSys Conference 2019. 2019.
[12] Zamanian, Erfan, et al. "The end of a myth: Distributed transactions can scale." arXiv preprint arXiv:1607.00655 (2016).
[13] Wang, Xizheng, et al. "StaR: Breaking the Scalability Limit for RDMA." 2021 IEEE 29th International Conference on Network Protocols (ICNP). IEEE, 2021.
[14] Kalia, Anuj, Michael Kaminsky, and David G. Andersen. "Design guidelines for high performance RDMA systems." 2016 USENIX Annual Technical Conference (USENIX ATC 16). 2016.
[15] Mittal, Radhika, et al. "Revisiting network support for RDMA." Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. 2018.
[16] Gao, Yixiao, et al. "When Cloud Storage Meets RDMA." 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). 2021.
[17] Li, Yuliang, et al. "HPCC: High precision congestion control." Proceedings of the ACM Special Interest Group on Data Communication. 2019. 44-58.
[18] Mellanox Technologies. 2022. libibverbs. Website. https://github.com/linux-rdma/rdma-core/tree/master/libibverbs.
[19] Mellanox Technologies. 2022. mthca driver. Website. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/infiniband/hw/mthca.
[20] Lowe-Power, Jason, et al. "The gem5 simulator: Version 20.0+." arXiv preprint arXiv:2007.03152 (2020).
[21] Wang, Zeke, et al. "Shuhai: Benchmarking high bandwidth memory on fpgas." 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2020.
[22] Singhvi, Arjun, et al. "1rma: Re-envisioning remote memory access for multi-tenant datacenters." Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication. 2020.