# Towards Connection-Scalable RNIC Architecture

**Ning Kang**
Institute of Computing Technology
**Zhan Wang** (wangzhan@ncic.ac.cn)
Institute of Computing Technology
**Fan Yang**
Institute of Computing Technology
**Xiaoxiao Ma**
Institute of Computing Technology
**Zhenlong Ma**
Institute of Computing Technology
**Guojun Yuan**
Institute of Computing Technology
**Guangming Tan**
Institute of Computing Technology

---

**Research Article**

**Keywords:** Architecture Design, Network Interface Card (NIC), Remote Direct Memory Access (RDMA), Scalability Problem

**Posted Date:** June 22nd, 2023

**DOI:** https://doi.org/10.21203/rs.3.rs-3076067/v1

**Additional Declarations:** No competing interests reported.

---

# Towards Connection-Scalable RNIC Architecture

Ning Kang[1,2], Zhan Wang[1], Fan Yang[1], Xiaoxiao Ma[1,2], Zhenlong Ma[1,2], Guojun Yuan[1], Guangming Tan[1]

[1]Institute of Computing Technology, CAS, Beijing, 100089, China.
[2]University of Chinese Academy of sciences, Beijing, 100089, China.

Contributing authors: kangning18z@ict.ac.cn; wangzhan@ncic.ac.cn; yangfan2020@ict.ac.cn; maxiaoxiao@ncic.ac.cn; mazhenlong@ncic.ac.cn; yuanguojun@ncic.ac.cn; tgm@ncic.ac.cn;

**Abstract**

RDMA is a widely adopted optimization strategy in datacenter networking that surpasses traditional kernel-based TCP/IP networking through mechanisms such as kernel bypass and hardware offloading. However, RDMA also faces a scalability challenge with regard to connection management due to limited on-chip memory capacity in the RDMA Network Interface Card (RNIC). This necessitates the storage of connection context within RNIC's memory and considerable performance degradation when maintaining a large number of connections.

In this paper, we propose a novel RNIC microarchitecture design that achieves peak performance with scalable connections. First, we model RNIC and identify two key factors that degrade performance when connections scale: head-of-line blocking when accessing the connection context and connection context dependency in transmission processing. To address the head-of-line blocking problem, we then combine a non-blocking connection requester and connection context management module to process prepared connections first, which achieves peak message rate when connections scale. Besides, to eliminate connection context dependency in RNIC, we deploy a latency-hiding connection context scheduling strategy, maintaining low latency when the number of connections increases. We implement and evaluate our design, demonstrating its successful maintenance of peak message rate (66.4Mop/s) and low latency (3.89us) while scaling to over 50k connections with less on-chip memory footprint.

**Keywords:** Architecture Design, Network Interface Card (NIC), Remote Direct Memory Access (RDMA), Scalability Problem

## 1 Introduction

Remote Direct Memory Access (RDMA) is a network technology widely deployed in High-Performance Computing (HPC) and Datacenter. It employs kernel bypass to provide a high throughput and low latency network. In addition, RDMA offloads a large number of network tasks to the RDMA Network Interface Card (RNIC), releasing more CPU resources for other datacenter applications. In recent years, lots of cloud service providers, including Microsoft, Alibaba, and Amazon [1, 2, 19, 20], have turned to RDMA to optimize their datacenter networks.

Offloading network tasks into the RNIC improves network performance, but it also brings some scalability issues. On the one hand, RNIC needs to manage network connections to quickly respond to communication requests, which requires the RNIC to store the connection context. On the other hand, due to the limited on-chip memory size of RNIC, the number of active connections it supports is also limited. Therefore, the size of on-chip memory limits the number of connections supported by RNIC, which is the source of connection scalability problem.

Recent works [3–5] present their open-sourced RNIC designs. However, these designs are limited in terms of connection support. For instance, eRNIC [4] is Xilinx's embedded RNIC, and its connection context is all stored in RNIC. It only supports a maximum of 127 connections, which limits the connection scalability of eRNIC. In addition, commercial solutions also face scalability problem. The ConnectX series RNICs deploy caches in RNIC for storing connection context and supports up to $2^{16}$ connections [7]. But these RNICs are closed-sourced, and their design details are unknown. Moreover, the latest ConnectX-6 can only sustain peak performance with no more than 450 connections [14].

To ensure connection scalability, this paper proposes a design that utilizes existing open source RNIC designs for caching connection contexts. We implement a connection context cache in the RNIC to temporarily store the connection context. When the cache is hit, RNIC directly retrieves connection context from its local cache. In case of a cache miss, RNIC obtains the lost context from host memory. This approach allows RNIC to support a greater number of connections without being constrained by its limited on-chip memory.

Although the concept is straightforward, our analysis identified three challenges in designing a connection context cache for RNIC: 1). When a cache miss occurs, subsequent connection context requests are blocked until the lost context is returned, resulting in head-of-line blocking. This blocks the processing path of RNIC during transmission and reduces its peak message rate as the number of connections increases. 2). The Infiniband specification [8] mandates sequential processing on the same connection to avoid semantic errors, which requires guaranteed blocking in certain cases and adds complexity to the problem. 3). In the event of a connection context cache miss, RNIC must retrieve the context from host memory for subsequent transmission processing. This results in additional PCIe access latency and an increase in RNIC's processing latency.

To address the aforementioned challenges, we propose csRNA, a connection-scalable RNIC architecture design that effectively avoids performance degradation as the number of connections increases. First, csRNA is equipped with a non-blocking requester and connection management module to ensure timely processing of subsequent connection context requests. Second, RNIC checks the dependencies between context requests within the same connection in the connection context cache to ensure semantic correctness in case of non-blocking processing requests. Finally, we deploy a latency-hiding mechanism for accessing connection context to maintain low latency even when context cache misses. Experimental results show that with a small on-chip memory footprint, csRNA can still achieve a message rate of 66Mops and a latency of 3.89us even with over 50k connections.

In summary, the main contributions of our work are:

1. We propose csRNA [9, 25], a connection-scalable RNIC architecture that supports a scalable number of connections. By implementing a connection context cache, we evaluate the effects of conventional approaches on system performance and identify their underlying causes.

2. We describe a non-blocking RNIC design to enhance message rate. The design includes a non-blocking requester and a connection management module, which avoids head-of-line blocking in the event of the RNIC's connection context cache misses.

3. We introduce a latency-hiding mechanism to optimize latency. This mechanism overlaps context access with normal processing by segmenting the context content, thereby hiding the access time of missing connection context.

4. We implement and evaluate csRNA in gem5 to assess its performance benefits. Our experimental analysis demonstrates that even with reduced on-chip memory usage, csRNA can maintain peak performance as connection scales exceed 50k.

## 2 Background and Related Works

### 2.1 RDMA Basics

**RDMA NIC (RNIC).** RNIC is a network interface card that facilitates RDMA transmission. RDMA enables the transport layer to be offloaded onto the hardware, resulting in reduced CPU overhead and low latency data transmission. As a result, RNIC is responsible for maintaining the RDMA connections. During connection establishment, user applications on different nodes exchange connection context and then pass it on to RNIC for maintenance. The context is then stored by RNIC for future use.

**Transport operation.** RDMA supports two types of transport operations, specifically 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. One-sided operation allows local applications to access remote memory directly, without the CPU involvement of the remote node. A typical one-sided RDMA operation includes RDMA Read and RDMA Write, which implement direct remote memory read and write, respectively. They have lower latency due to the less involvement of the remote CPU.

**Service type.** RDMA also provides a variety of transportation services for applications. Specifically, RDMA has two service modes :RC (Reliable Connection) and UD (Reliable Datagram). RC is the only service type that fully supports all one-side operations. Recently, RC has gained widespread popularity [1, 2, 15] in datacenter

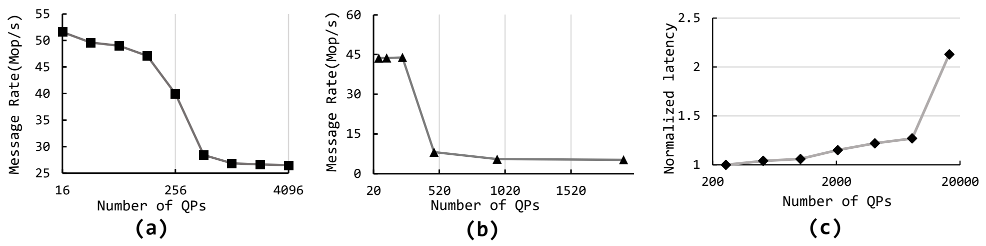

**Fig. 1** Performance changes as the number of connections increases. (a) Message rate results under commercial RNIC (Mellanox ConnectX-5). (b) Message rate results under first-come-first-served policy (gem5). (c) Latency results under different number of QPs (Mellanox ConnectX-5).

deployments due to the low latency nature of one-side operations. Therefore, this paper focuses on the types of RC services.

**Queue pair (QP) & queue pair context (QPC).** RDMA establishes connections between communication nodes by creating queue pairs (QPs). A QP consists of a send queue (SQ) and a receive queue (RQ), storing send and receive descriptors. Descriptors are transmission indicators, including address and data length. The descriptor is also called work queue elements (WQEs). RNIC uses descriptors to guide the sending or receiving of data. QP itself also has many attributes, such as address and length of the SQ and RQ. As a result, the connection context is created to hold these attributes. Connection context is also called QP context (QPC). A typical commercial RNIC ConnectX-4 requires 256 bytes of context to be maintained for a RDMA connection [7, 22].

## 2.2 Connection Scalability Issues

RDMA connection scalability is a critical requirement for datacenters. Unlike HPC, data center networks exhibit asymmetric communication patterns. These patterns typically involve in-cast or out-cast scenarios, where a server node communicates with multiple client nodes. The server node in such a network has to sustain thousands of connections with the clients. For instance, in the Oracle's Timestamp system [13], the server node was responsible for managing over 2800 concurrent connections. As RDMA becomes more prevalent in large-scale datacenters, the need for RDMA's connection scalability capabilities also grows.

However, RDMA has a trade-off between offloading hardware and maintaining connection scalability. It uses RNIC for handling the transport layer, which reduces CPU usage and increases packet throughput. But this also means that the RNIC has to take care of both sending data and keeping track of the connection state. The Infiniband specification [8] states that the RNIC can have at most $2^{24}$ QPCs. This means that the RNIC needs $2^{24} * 256$ bytes $= 4$GB of space to store all QPCs. But the RNIC only has a small amount of memory on its chip, which is not enough to store all the QPCs. This limits how many connections RNIC can handle and affects its connection scalability.

We evaluate the performance impact of increasing the number of connections using Mellanox ConnectX-4 RNIC. We use 4 nodes in the experiment, with one node as

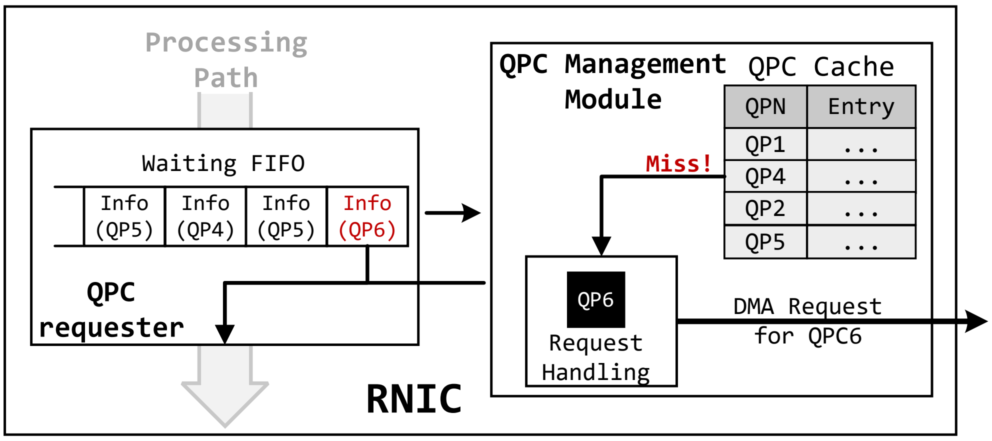

**Fig. 2** RNIC uses a first-come-first-served mechanism to manage QPC requests. The QPC requester module is a component of the RNIC processing path that initiates QPC requests. The waiting FIFO stores the information associated with each QPC request (Info), and the values in parentheses indicate the QPC that the information belongs to.

the server and the other three as the clients. The server sends 64-byte RDMA Read requests concurrently to the three clients. Fig. 1(a) shows how the message rate varies with the increasing number of connections. The RNIC message rate starts to decline when the server has more than 256 connections.

## 3 Problems When Deploying QPC Cache

Current open-source RNIC designs [3, 4] store all QPC entries within the RNIC chip to enable efficient management, even in first-come-first-served (FCFS) request processing mode. As illustrated in Fig. 2, these designs process QPC access request through a request fifo. The fifo receives the request and forwards it to the QPC management module for processing. Concurrently, the requester responsible for initiating the QPC request also maintains a waiting FIFO to temporarily store relevant information. As the QPC management module processes sequentially, upon receiving the QPC response, the requester can retrieve the corresponding information from the top of the waiting fifo to proceed with subsequent processing steps. This approach is effective for a limited number of QPCs as they are entirely located within the RNIC and have identical request processing times.

Integrating a QPC cache into an RNIC may resolve the limitation in the number of QPCs supported. However, the current RNIC design (FCFS) is not conducive to the use of a QPC cache. The remainder of this section aims to illustrate the implications of deploying a QPC cache on existing designs.

On the one hand, current RNIC designs adopt a first-come-first-served QPC processing policy, but integrating QPC caches directly can cause head-of-line blocking and affect RNIC's performance. According to the Infiniband specification [8], the processing logic within the same QP must adhere to order-preserving, whereas no such requirement is applicable to processing between different QPs. Adding QPC cache may result in subsequent requests being blocked by previous ones due to varying processing times. Fig. 2 demonstrates this scenario where the QPC requester requests QPC for subsequent processing. The QPC requester is a component in processing path that posts QPC request to access QPC. In this example, QP6 is missing, causing subsequent requests to be blocked until QPC6 returns. Although the connection contexts of QP5 and QP4 are stored in the QPC cache, their processing remains blocked until the connection context of QP6 returns. The head-of-line blocking problem can further cause the performance degradation of RNIC.

To clarify this further, we evaluate the message rate performance of the first-come-first-served strategy when the QPC cache is missing. We implement the FCFS QPC management module in gem5 simulator and incorporate the QPC cache into the RNIC design. The size of the QPC cache used in the experiments is 300 entries. Fig.1(b) displays the results of sending RDMA Write operations from one server node to 10 client nodes. When the number of connections slightly exceeded the cache size, RNIC's message rate dropped dramatically. These experiments indicate that RNIC's message rate may suffer significantly from head-of-line blocking.

On the other hand, when the QPC cache misses, it can result in increased latency during transfer processing. Before RNIC starts the data transmission, it has to retrieve QPC to acquire the relevant information. If the QPC cache does not contain the corresponding QPC, the RNIC retrieves it from the host memory, causing a latency increments by a PCIe Round Trip Time (RTT). The current end-to-end latency for RDMA is a few microseconds, whereas the typical PCIe RTT latency is around 500 nanoseconds, which is a non-negligible latency overhead. We conduct tests on the average latency of Send in UD service type, transmitting 1-byte messages in one direction. We test the latency incremental of the RNIC on the requesting side. As illustrated in Fig. 1(c), when the number of connections exceeds the QPC cache capacity, the overall end-to-end latency starts to increase.

After analyzing the integration of QPC cache into RNICs, we have identified three challenges for optimizing connection scalability:

**C1.** How to handle communication requests between different connections without blocking? In previous designs [3, 4], QPC access is done sequentially. As such, information related to QPC requests is simply placed into a waiting FIFO. Once QPC responses are received, relevant information is then retrieved from the top of the waiting FIFO. However, with the support of non-blocking processing by the RNIC, requests are no longer processed in a first-come-first-served order. This has brought up the issue of how to locate relevant information based on the returned QPC, in order to carry out subsequent non-blocking transmission processing.

**C2.** How to ensure the order dependencies of QPC requests when requests are returned out of order? According to the Infiniband specification [8], order preservation is required for different communication requests within the same QP. However, when the QPC management module supports out-of-order processing, the order of the QPC response returned becomes unpredictable. This can result in subsequent QPC requests being returned earlier than previous ones, causing processing sequence disorder and even errors in the application. Therefore, maintaining dependence on the QP order is also an important consideration for the QPC management module.

**C3.** How to minimize the latency increase caused by the deployment of the QPC cache? When a QPC cache is missing, the RNIC must retrieve the QPC from the host's memory and access the corresponding WQE for any subsequent operations. However, this process of fetching the missing QPC from memory results in increased latency for data transfer. Therefore, reducing the processing latency in the case of integrated QPC cache is also a problem we need to consider.

## 4 Connection-Scalable RNIC Architecture

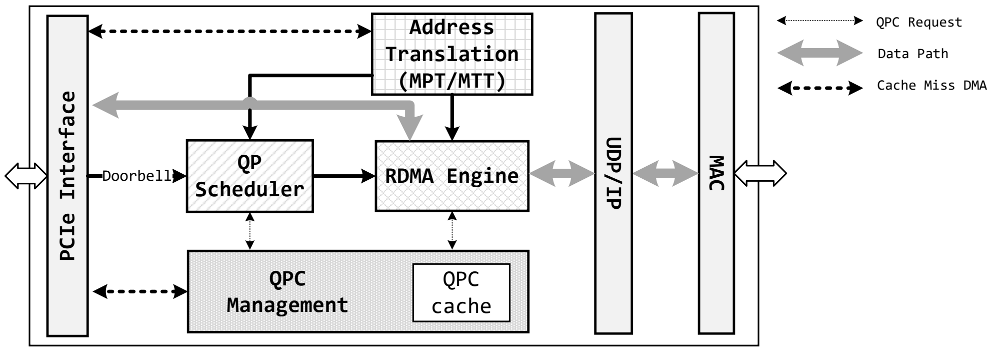

**Fig. 3** Overall csRNA Architecture. The RNIC utilizes PCIe to attach to the host CPU, and employs Ethernet to connect to the fabric.

In this section, we first introduce the overall design of csRNA, followed by a detailed description of optimization strategies.

### 4.1 High-level Overview

The architecture of our csRNA is divided into 4 main components that are visualized in Fig. 3. The QPC management module maintains the RNIC connection context configured by the host CPU for data transmission. Both QP scheduler module and RDMA engine module post the QPC request to the QPC management module. The module includes a QPC cache to store the connection context. When the cache misses, the module fetches the corresponding QPC from the host memory. The detailed description of the QPC management module is covered in Section 4.3. The address translation module ensures that RNIC is able to manage and maintain virtual addresses. The module includes memory protection table (MPT) and memory translation table (MTT), used to validate and translate the virtual address, respectively. Similar to the QPC management module, the address translation module also uses a cache, which is used to store the page table. This part of cache optimization is beyond the scope of our work. The QP scheduler module is responsible for selecting the next QP to serve, on the transmit path. This module receives doorbells from host memory and schedule corresponding QP for transmission. Then it retrieves the WQEs containing certain amount of data and pass it to the RDMA engine. The RDMA engine handles transmission requests and their responses under RDMA semantics. RDMA request types currently supported by our design include RDMA write and RDMA read. The RDMA engine starts the transmission request when it receives a WQE. This instructs the RDMA engine to process the request. On the response side, when RNIC receives a packet from UDP/IP, the RDMA engine determines the 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.

## 4.2 Key Design Ideas

In this section, we will address the challenges raised in Section 3. We present csRNA, a connection-scalable RDMA NIC architecture. csRNA proposes three optimization strategies:

1. A non-blocking QPC management module is proposed that handles non-blocking connection information access requests, and solves the sequential dependency problem between different QPC requests (**C1, C2**).

2. An out-of-order QPC Requester module is designed to schedule ready QPCs first. This approach allows RNIC to handle non-blocking transmission requests for different connections (**C1**).

3. A delay-concealing QPC scheduling mechanism is developed at the transmitter. It divides the QPC information such that the missing QPC read time overlaps that of WQE's, resulting in optimized processing delay. (**C3**).

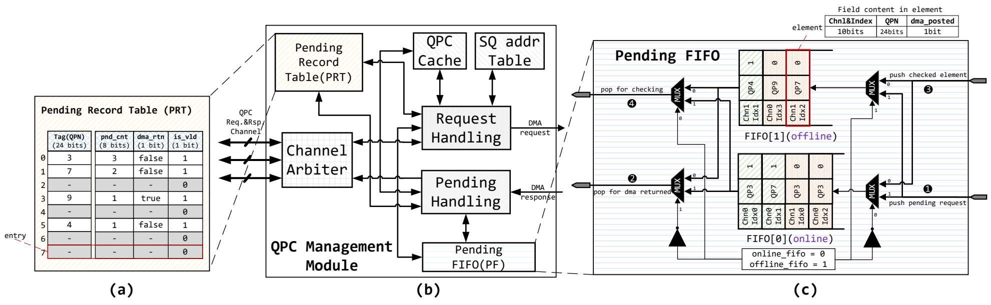

**Fig. 4** Non-blocking QPC management module. (a) The structure of pending record table (PRT). (b) The overall architecture of QPC management module that solves the problem of head-of-line blocking. (c) The structure of pending FIFO (PF).

## 4.3 Non-blocking QPC Management Module

As mentioned in Section 3, 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 order for QPC requests from the same QP.

We design a non-blocking QPC management module that meets all of the above requirements. Fig. 4(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 sequencing requirements. The 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 process QPC requests from the QPC requester. The pending handling unit processes the QPC requests stored in the pending FIFO. The channel arbiter unit arbitrates requests from multiple QPC requester in the RDMA engine and forwards QPC responses back.

### 4.3.1 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 differentiate between the two types of requests, the pending FIFO includes a 1-bit field called *dma_posted* within the element. As displayed in Fig. 4(c), if the *dma_posted* field is set to 1, the QPC request is pending due to the QPC cache missing. Additionally, the request associated with the element has initiated a read request via DMA. Conversely, if the *dma_posted* field is 0, it signifies that the element is blocked due to order preservation for a single QP, and the request does not prompt a DMA read request on the requested QPC entry. Moreover, an element includes other fields such as QPN, channel number, and index, which are leveraged in subsequent processing.

Next, we will examine the distinctions between the two types of QPC requests. Concerning *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. On the other hand, *Type 2* requests do not wait for a DMA response but rather evaluate whether the QPC requests fulfill the processing requirements, that QPC cache has the required QPC.

**Algorithm 1:** Finite State Machine for Request Handling Unit

```
if Incoming QPC request for qpn then
    Read qpn entry in PRT;
    if Find qpn in PRT then
        Push QPC request to PF;
        Update entry in PRT with pnd_cnt++;
    end
    else if QPC cache hit for qpn then
        Read or update the entry in QPC cache;
        Return QPC if this is a read request;
    end
    else
        Post DMA read for QPC of qpn;
        Push QPC request to pending FIFO;
        Write an entry for qpn with pnd_cnt=1;
    end
end
```

Based on the analysis above, we construct the pending FIFO as shown in Fig. 4(c). The unit contains two interfaces: ❶, ❷ for normal request submission and response processing; ❸ and ❹ are used for checking and processing *Type 2* requests. In addition, the pending FIFO contains two FIFOs of the same size to store pending elements and two registers of $online\_fifo$ and $offline\_fifo$ to indicate the type of the two FIFOs. For example, in Fig. 4(c), the values $online\_fifo$ 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 *Type 1* elements; the input port is connected to ❸ to write back to the FIFO an element that has not been checked. Failure to check means that the element previously popped from ❹ is not *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 4.4.

### 4.3.2 Pending Record Table (PRT)

The 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 content. There are three fields in an entry. *is_vld* indicates 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 subsequent QPC requests corresponding to the QP in the pending FIFO can be obtained directly from the QPC cache. The number of entries in PRT is $3 * ooo\_cap$, where *ooo_cap* is the out-of-order capacity.

**Algorithm 2:** Finite State Machine for Pending Handling Unit

```
if QPC DMA response for qpn returns then
    $elem < -PF.pop\_return()$;
    if $elem.dma\_posted == 1$ then
        Read & update $PRT[qpn]$;
        $start\_check = 1$ if no DMA response;
        Return $QPC[qpn]$;
        Return;
    end
end
else if $start\_check! = 0$ then
    $elem < -PF.pop\_check()$;
    if $elem.dma\_posted == 1$ then
        $PF.push\_checked(elem)$;
        Return;
    end
end
$entry < -PRT[elem.qpn]$;
$entry.pnd\_cnt - -$;
if QPC cache hit for this elem then
    update $PRT[elem.qpn]$;
    Return $QPC[elem.qpn]$;
end
else if $entry.dma\_rtn == true$ then
    $entry.dma\_rtn = false$;
    $PF.push\_request(elem)$;
end
else
    $PF.push_checked(elem)$;
end
Update $start\_check$;
```

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 channels in the QPC requester; *index* is the unique number given by the QPC requester 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 the QPC requester with channel 0, the address for the PRT entry created by the request is 0.

### 4.3.3 Request Handling

The request handling unit is used to process QPC requests and write the requests back to the channel arbiter or submit them to the 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 issue of sequence preservation. In this case, the QPC is accessed directly from the QPC cache. Hitting requests are written directly back to the QPC requester. 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 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 on the relevant entries in the pending record table.

### 4.3.4 Pending Handling

The pending handling unit is used to handle QPC requests in the pending FIFO. According to Section 4.3.1, the pending FIFO stores two types of blocked QPC requests. For *Type 1* requests, the unit simply waits for the DMA response to return. Since the DMA response of the 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 on the request has been removed, and the 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 DMA read request from QPC is also submitted, and the QPC request is again pushed to the pending FIFO.

## 4.4 Non-blocking Supports for QPC Requester

In this section, we introduce the detailed design of the RDMA Engine and explain its implementation of non-blocking execution. Fig. 5(a) shows the structure of the RDMA engine. Descriptor unit manages 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

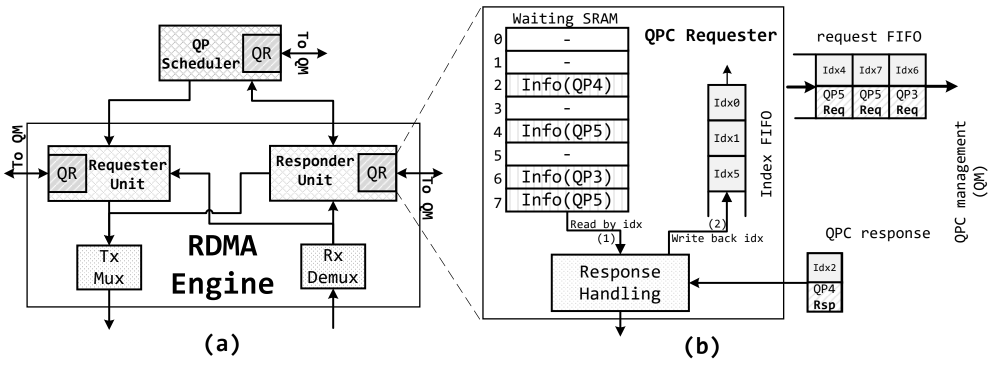

**Fig. 5** Non-blocking supports for RDMA Engine. (a) Architecture of RDMA Engine. QR stands for non-blocking QPC requester. (b) QPC requester that supports non-blocking QPC processing.

receiving end. The requester unit processes communication requests, including memory write or read request generation, sending window maintenance, packet response processing, and packet retransmission. The responder unit processes incoming data read or write requests from the remote and generates corresponding response packets back to the wire. Tx Mux and Rx Demux are the arbiter and distributors, respectively.

In order to implement its non-blocking processing, the RDMA engine deploys a non-blocking QPC requester on its processing path. As shown in Fig. 5(a), three RDMA engine units are equipped with QPC requester. The structure of the QPC requester is shown in Fig. 5(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, which is mentioned in Section 4.3.2. Note that in our implementation, the index width 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 doorbells posted from the host CPU; In the responder unit, the information includes incoming packets received from the wire. And in the requester unit's QR unit, the information includes 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.

Here is an example of how the QPC requester supports non-blocking QPC processing. In the initial state, FIFO stores consecutive numbers starting at 0 as *index*. When a QPC request arrives, the QPC requester first pops an *index* from the Index FIFO and writes the relevant information of the QPC request to the Waiting SRAM with that *index* as the address. Next, the QPC requester pushes the QPC request, along with the retrieved *index*, to the QPC management module via the request FIFO. After the QPC management module returns the QPC through response FIFO, the response handling returns the relevant information of QPC from the corresponding address in the Waiting SRAM according to the *index* in the QPC response (channel (1) in Fig. 5), and forwards it along 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. 5(b)). By designing a QPC requester unit, the

RDMA engine handles transmission tasks of different connections without blocking, avoiding blocking RDMA transmission processing due to the QPC cache missing.

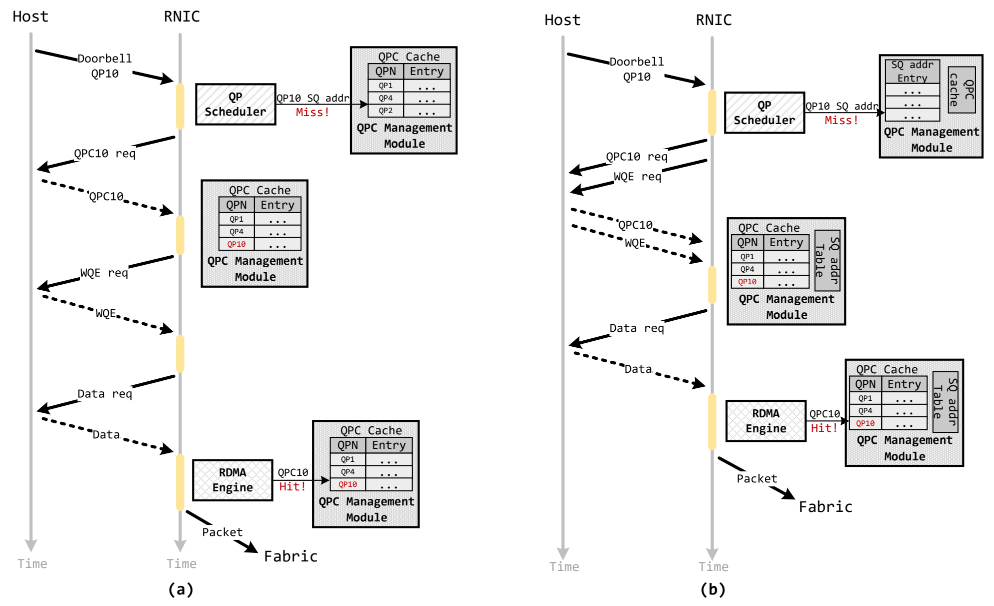

**Fig. 6** Latency comparison between original strategy and latency-hiding scheduling strategy. (a) Processing example of QPC10 misses under original strategy. (b) Processing example of QPC10 misses under latency-hiding strategy.

## 4.5 Latency-hiding QPC Scheduling

To further optimize latency at the sender when the QPC cache is missing, csRNA implements two optimizations: 1) Separating QPC information to remove dependence on accessing both QPC and WQE, and 2) Developing a QPC pre-fetching mechanism to reduce latency caused by QPC cache missing.

First, the csRNA divides the QPC. SQ address and offset are part of the QPC. Before sending the WQE read request, the RNIC must retrieve the SQ address and offset from the QPC. As shown in Fig. 4(b), csRNA deploys a SQ address table, which stores all SQ addresses and their offsets in the RNIC. The csRNA splits the QPC, maintaining the SQ's address and offset on the RNIC, while the remainder of the QPC information is cached on the QPC cache. As a result, the subsequent WQE read at the sender is not dependent on the QPC cache missing.

In addition to QPC splitting, csRNA also introduces a QPC pre-fetching mechanism. QPC pre-fetching involves initiating a read operation for the QPC of a QP after initiating a WQE read request, ensuring that subsequent processing won't encounter QPC cache misses.This process enables QPC and WQE access to overlap, concealing processing delays caused by QPC cache misses on the sender side.

Fig. 6 illustrates how the latency-hiding QPC scheduling strategy optimizes latency in the case of QPC cache misses. As described in Fig. 6(a), when the host CPU sends a doorbell attached to QP10, the RNIC schedules QP10 in the QP scheduler module (as described in Section. 4.1). The QP scheduler module reads the QPC information corresponding to QP10 to obtain the SQ address and offset information and reads the WQE. In this case, the QPC10 is missing from the QPC cache, so the RNIC needs to read QPC10 from the host memory and return it before it can read the WQE. Once the QPC data is retrieved, the RDMA engine module can read QPC10 directly from the cache for packet encapsulation.

Fig. 6(b) depicts the mechanism of RNIC when the latency-hiding scheduling strategy is employed. As the optimized RNIC stores the SQ addresses and offset information of all QPs, it can directly initiate WQE read requests even if the QPC10 is not cached. Meanwhile, the QPC Management module initiates a read request to QPC10 after sensing that the SQ address table has been read to ensure that subsequent access to QPC10 is not missed. The SQ address table eliminates the dependency between WQE read requests and QPC10 read requests, allowing the access latency to overlap.

## 4.6 The Placement of QPC

In this section, We discuss the optimal storage location for QPC. Non-blocking QPC management mechanism and latency-hiding QPC scheduling strategy effectively reduce transmission processing latency. However, as the number of connections increases, the QPC cache hit ratio decreases. If QPC is stored in host memory, each cache miss consumes PCIe bandwidth to obtain QPC entries, which affects the data transmission throughput. To avoid this problem, we suggest storing QPC in onboard DDR. In this way, when QPC cache misses, RNIC can directly access QPC entries from onboard DDR without consuming PCIe bandwidth. Furthermore, the onboard DDR has much lower access latency than the host memory [10, 26], which also helps in mitigating the processing latency caused by QPC cache miss.

## 4.7 On-chip Memory Occupancy

Based on the analysis in Section. 4.4 and 4.3, we get the relationship between the incremental of on-chip memory and out-of-order capacity under the out-of-order strategy:

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

, where $mem_{QR}$, $mem_{PF}$, and $mem_{PRT}$ are the additional on-chip memory occupancy for QPC requester, 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.

Optimizing csRNA latency also entails additional storage overhead. As outlined in Section 4.5, csRNA stores all the necessary information to read the work queue element on the RNIC, including the SQ address and WQE offset. Each QP takes up 10 bytes of on-chip memory. A comprehensive breakdown of the total memory footprint for this aspect is provided in Section 5.

## 5 Evaluation

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

- Can the non-blocking QPC management mechanism and latency-hiding QPC scheduling strategy achieve better performance? (in Section 5.2)
- Under what parameter configuration can csRNA achieve better performance? (in Section 5.3)

### 5.1 Settings

**Table 1** RNIC on-chip memory used by connection information

| RNIC Type | eRNIC | SRNIC | ConnectX-5 | csRNA |
|---|---|---|---|---|
| On-chip memory usage | 32KB | 2.5MB | 2MB[18] | 1.4MB |
| Max. Number of QPs | 127[4] | 10K[28] | 128K[23] | 128K |

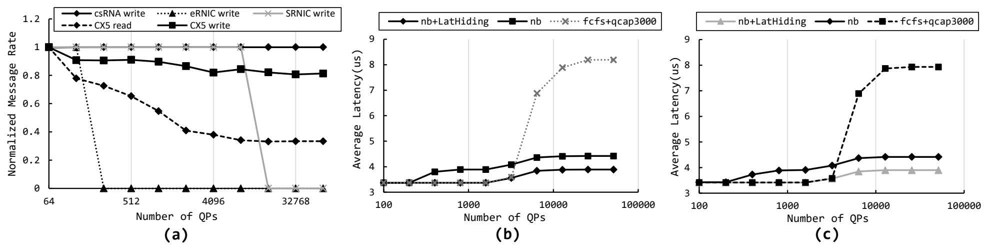

**Fig. 7** Connection scalability test. (a) Normalized message rate changes at different number of QPs. (b) Latency changes with different optimizations (RDMA write). (c) Latency changes with different optimizations (RDMA read). CX5: ConnectX-5. fcfs: first-come-first-served QPC management scheme. nb: non-blocking QPC management scheme. LatHiding: Latency-hiding QPC scheduling strategy. qcap: the number of entries in QPC cache.

**Environment Settings.** We implement and evaluate csRNA in the gem5 simulator [24], with the RNIC module consisting of about 9000 lines of code. As part of the experimental setup, we use a single server that connects to 10 clients via a 100Gbps Ethernet switch, with a network link delay of 1µs. Each node we test has 10 CPU cores, each with a frequency of 2GHz. To expedite test execution time, we utilize the System-Call Emulation (SE Mode) of gem5. The PCIe bandwidth for the experiment is 128Gbps, which matches the bandwidth of PCIe 3.0 X 16. We set the Round Trip Time (RTT) delay for PCIe to 500ns [10, 26] when accessing host memory. The RNIC clock frequency we set to 1GHz, and utilize least recently used (LRU) as the QPC cache replacement policy. Notably, we set the message size to a typical test case of 64 bytes, as identified in connection scalability problems [11, 12]. Unless stated otherwise, we assume that the QPC cache has a capacity of 300 QPC entries in our experiments.

**Software Settings.** To enable RDMA applications on csRNA, we design and implement a software stack that offers a verbs interface similar to libibverbs [21], which is a de facto standard interface for RDMA programming. The software stack consists of three components: 1) A kernel-space driver that runs on gem5 in SE mode and allocates resources for RDMA operations; 2) A user-space driver and verbs library that interacts with the RNIC in the data path and provides the verbs API to the applications; 3) A communication management layer that handles connection establishment between RDMA endpoints. We use this software stack to evaluate the performance of csRNA.

## 5.2 Connection Scalability Test

We conducted a comparative evaluation of csRNA against existing solutions, including eRNIC, SRNIC, and Mellanox ConnectX-5, by measuring their message rates as the number of connection scales. In order to highlight the connection scalability problem, we employed a normalized message rate, where the maximum message rate achieved by each RNIC is set to 1. csRNA leverages a QPC cache and SQ address table on the RNIC, which is capable of storing 300 QPC entries and an additional 128K SQ addresses, respectively. The amount of on-chip memory space required by the QPC in each RNIC is shown in Table. 1. Despite maintaining more active connections, csRNA occupies less memory footprint. Fig. 7(a) gives message rate results for different RNICs. Both eRNIC and SRNIC have a fixed connection capacity. Thus, as the number of connections exceeds their limit, we set their message rates to 0. In addition, we test the connection scalability of ConnectX-5 and observe a decrease in peak message rate with an increase in the number of connections for both RDMA write and RDMA read. As ConnectX-5 is a commercial network card, we lack information on its microarchitecture design and performance degradation reasons. Our results (shown in Fig. 7(a)) demonstrate that compared to the existing RNIC design, csRNA can maintain a peak message rate even when reaching 51.2K connections.

We also test the latency performance. In latency testing, each process on the server side sends a transport request from a QP that it maintains. When the process receives a transfer completion event from RNIC, it sends a transfer request to the next QP it maintains. This means that QPs between different processes can initiate access requests at the same time, creating competition for accessing the QPC management module. We test latency performance for first-come-first-served (FCFS), non-blocking (NB), and latency-hiding (LatHiding) policies. Under the first-come-first-served access policy, QPC cache capacity is 3000 QPC entries. Under other optimization strategies, the cache has a capacity of 300 QPC entries. Fig. 7(b) shows the change in transmission delay as the number of QP connections increases under the RDMA write transmission type. In the figure, we see that with the FCFS strategy, transmission latency increases dramatically when the number of connections slightly exceeds the QPC cache capacity. Under NB policy, the transmission delay increases by one PCIe RTT when the number of connections exceeds the QPC cache capacity. This is because the QPC cache does not hit and RNIC needs to read QPC information from the host memory. When the latency-hiding QPC scheduling strategy is used, RNIC can maintain the original latency performance even if the QPC cache on the sender does not hit. When the number of connections exceeds 3000, the QPC cache on the receiving end also starts to fail to hit. At this time, RNIC transmission latency starts to rise slowly, but the latency-hiding QPC scheduling strategy is still better than the performance of FCFS and NB strategy. Fig. 7(c) shows the test results of RNIC under the RDMA read transmission type, which has similar performance characteristics to RDMA write.

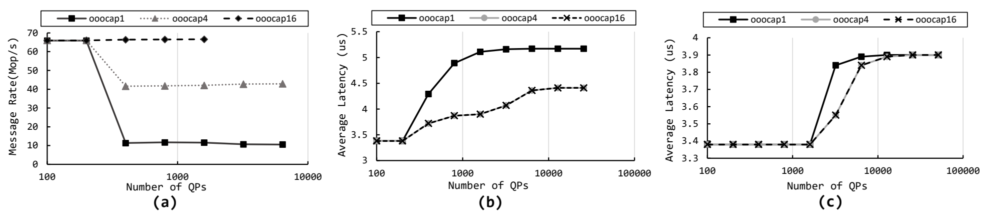

**Fig. 8** Out-of-order Capacity Test. (a) Message rate under different out-of-order capacity. (b) Latency under different out-of-order capacity, without latency-hiding QPC scheduling strategy. (c) Average latency under different out-of-order capacity, with latency-hiding QPC scheduling strategy. ooocap: out-of-order capacity.

## 5.3 csRNA Configurations

**Out-of-order Capacity.** To study how csRNA parameter configuration affects its performance, we perform a more detailed test of its use of on-chip memory resources. We use the same message rate test method as in Section 5.2 to evaluate csRNA performance under different out-of-order capacities. The results, presented in Fig. 8(a), demonstrate that the message rate of RNIC increases as the out-of-order capacity grows, while still maintaining a substantial number of connections. When the out-of-order capacity is 16, csRNA can achieve peak performance during connection scaling, which consumes only about 0.5KB of on-chip memory (calculated from the formula 2). This is because, in this case, 16 non-blocking QPC requests are sufficient to enable the QPC management module to run efficiently in pipeline, thus preventing the RNIC's hardware processing resources from being wasted due to the QPC cache missing. At this point, throughput is mainly constrained by the packet processing speed of RNIC hardware.

In addition, we investigated the effect of out-of-order capacity on latency performance. Fig. 8(b) shows the delay performance results of RNIC under different out-of-order capacities. The results show that when the number of connections is slightly larger than the QPC cache capacity, the average processing latency decreases with the increase of the out-of-order capacity. As the number of connections continues to increase (beyond 10k), the latency under each out-of-order capacity gradually converges. This is because when the number of connections is slightly greater than the QPC cache capacity, the non-blocking QPC management module can prioritize the prepared QPCs, thereby reducing the average processing latency. However, when the number of connections continues to increase, almost all QPCs in the RNIC are missing, and each message processing needs to increase the access time of two QPCs from host memory (one on the sender and one on the receiver). The optimization effect of non-blocking QPC management mechanisms on latency is no longer effective at this point. Fig. 8(c) shows the latency performance under a latency-hiding QPC scheduling strategy. The results reveals the similar trend as in Fig. 8(b), but with a lower latency than the results in Fig. 8(b). This is because the latency-hiding QPC scheduling strategy can hide the QPC access delay of the sender, thereby reducing the average processing latency of RNIC. The experimental results show that RNIC can achieve the optimal performance under 16 out-of-order capacity.

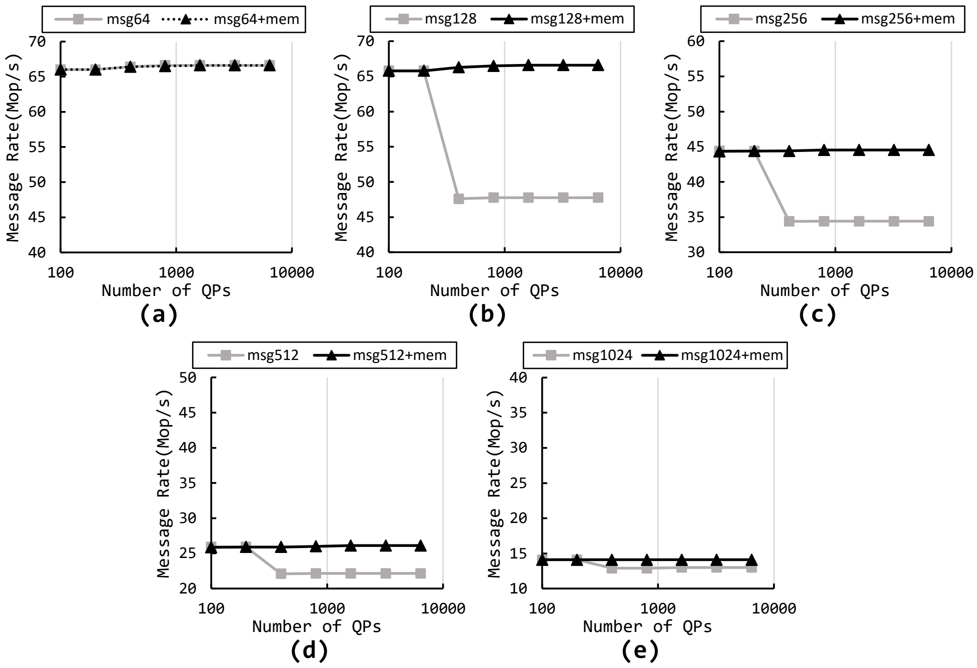

**Fig. 9** QPC Placement Test. (a)-(e) Message rate under different message size. msg: message size. mem: onboard memory.

**QPC Placement.** In this section, we examine the impact of QPC placement on RNIC performance. In Section 4.6, we analyze the PCIe bandwidth occupancy problem caused by the QPC cache missing and propose a method to eliminate this effect using on-board memory. Fig. 9(a)-(e) shows the message rate comparison between QPC in host memory and on-board memory for different message sizes. The results show that when the message size is small (below 64 bytes), QPC placement has little effect on performance. This is because RNIC's processing bottleneck at this point is the packet processing power of its hardware, not PCIe bandwidth. As message size increases, RNIC's processing bottleneck shifts to PCIe bandwidth. Storing all QPCs in on-board memory ensures that QPC access does not affect PCIe bandwidth requirements for messages. As message sizes continue to increase, the impact of QPC placement on RNIC message rates decreases (Fig. 9(c)-(e)). This is because QPC access is message granularity. As the message size gradually increases, the frequency of accessing the QPC gradually decreases. We did not test larger messages. In the current RNIC implementation, the amount of data transferred at a time is only at the KB level to avoid queues starving [28]. Experimental results confirm that placing QPC in onboard memory can effectively optimize RNIC performance when the number of connections increases.

## 6 Related Works

**RNIC design.** StRoM[3] presents an RNIC that supports offloading and focuses on the architecture of a basic RNIC and offloading stream tasks. However, StRoM has scalability limitations as it keeps all connection information in on-chip memory. StaR[6] tries to address the connection scalability problem by moving connection information to the remote side, but this may raise security issues when the remote node controls the local's state. 1RMA[27] is a recent system that aims to improve RDMA scalability in multi-tenant data centers by redesigning remote memory access and eliminating connection establishment processes. Unlike these works, our work seeks to enhance RNIC connection scalability without changing standard RDMA semantics. SRNIC[28] proposes a network-scalable RNIC architecture by introducing selective repeat in RDMA, and it stores all QPC in RNIC's on-chip memory. Our work tackles the performance challenges of integrating QPC cache into the RNIC.

**Connection scalability problem.** Existing research attempts to alleviate the RDMA scalability problem in software, leaving the scalability problem in RNIC unresolved. Some work [12, 17] try to ease the scalability problem in RNIC by limiting the number of connections in the software. For example, Flock [11] allocates up to 256 QP connections for their communication, which limits node scales in its connection topology to no more than 256 nodes. Other works prefer UD to avoid RDMA scalability problems. eRPC[18] uses two-sided operation over UD to avoid RDMA's scalability problem. In this way, developers must forgo the one-sided operation provided by the hardware, and they must implement reliable connections themselves, which causes additional overhead. In our work, we first answer the question of why RNIC performance degrades when the number of connections increases, and propose a non-blocking RNIC design that effectively improves RNIC connection scalability. Optimization in csRNA is orthogonal to the software optimization strategy mentioned above.

## 7 Conclusion

The connection scalability problem of RDMA due to limited on-chip memory in RNIC is a well-known challenge. Previous open-sourced works [3, 5, 16] on RNIC design do not address this issue. We propose csRNA, which provides a scalable RNIC design example. csRNA is a connection-scalable RNIC architecture that preserves peak performance when connections increase. The key idea is to introduce non-blocking QPC management and latency-hidden QPC scheduling strategy in RNIC that avoid latency incremental when QPC cache misses. We implement and evaluate csRNA in gem5 [24] and the results show that with less on-chip memory usage, csRNA can maintain peak performance (66.4Mops, 3.89us) when the connection scales to over 50K.

## 8 Declarations

### 8.1 Ethical Approval

Not applicable.

### 8.2 Competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

### 8.3 Authors' contributions

Ning Kang, Guangming Tan, Guojun Yuan, and Zhan Wang wrote the main part of the manuscript. Ning Kang, Fan Yang, Zhenlong Ma, and Xiaoxiao Ma are responsible for the design and evaluation. All authors reviewed the manuscript.

### 8.4 Funding

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).

### 8.5 Availability of data and materials

All data generated or analysed during this study are included in this published article. And the relevant program can be found in [9].

## 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] Pan, Pulin, et al. "Towards stateless rnic for data center networks." Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019. 2019.

[7] NVIDIA. 2022. Mellanox Adapters Programmer's Reference Manual (PRM). Website. https://network.nvidia.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf.

[8] Infiniband. 2022. Infiniband Architecture Specification, Volume 1. Website. https://cw.infinibandta.org/document/dl/8567.

[9] NCSG Group. 2022. csRNA. https://github.com/ncsg-group/csRNA.

[10] 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.

[11] 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.

[12] 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.

[13] Zamanian, Erfan, et al. "The end of a myth: Distributed transactions can scale." arXiv preprint arXiv:1607.00655 (2016).

[14] Wang, Xizheng, et al. "StaR: Breaking the Scalability Limit for RDMA." 2021 IEEE 29th International Conference on Network Protocols (ICNP). IEEE, 2021.

[15] Kalia, Anuj, Michael Kaminsky, and David G. Andersen. "Design guidelines for high performance RDMA systems." 2016 USENIX Annual Technical Conference (USENIX ATC 16). 2016.

[16] Mittal, Radhika, et al. "Revisiting network support for RDMA." Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. 2018.

[17] Tsai, Shin-Yeh, and Yiying Zhang. "Lite kernel rdma support for datacenter applications." Proceedings of the 26th Symposium on Operating Systems Principles. 2017.

[18] Kalia, Anuj, Michael Kaminsky, and David Andersen. ”Datacenter RPCs can be General and Fast.” 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019.

[19] Gao, Yixiao, et al. ”When Cloud Storage Meets RDMA.” 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). 2021.

[20] Li, Yuliang, et al. ”HPCC: High precision congestion control.” Proceedings of the ACM Special Interest Group on Data Communication. 2019. 44-58.

[21] Mellanox Technologies. 2022. libibverbs. Website. https://github.com/linux-rdma/rdma-core/tree/master/libibverbs.

[22] Mellanox Technologies. 2023. mthca driver. Website. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/infiniband/hw/mthca.

[23] Mellanox Technologies. 2023. mlx5 driver. Website. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/infiniband/hw/mlx5.

[24] Lowe-Power, Jason, et al. ”The gem5 simulator: Version 20.0+.” arXiv preprint arXiv:2007.03152 (2020).

[25] Kang, Ning, et al. ”csRNA: Connection-Scalable RDMA NIC Architecture in Datacenter Environment.” 2022 IEEE 40th International Conference on Computer Design (ICCD). IEEE, 2022.

[26] 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.

[27] 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.

[28] Wang, Zilong, et al. ”SRNIC: A Scalable Architecture for RDMANICs.” 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). 2023.