Toward Effective and Fair RDMA Resource Sharing

Haonan Qiu, Xiaoliang Wang, Tianchen Jin, Zhuzhong Qian
Baoliu Ye, Bin Tang, Wenzhong Li, Sanglu Lu
National Key laboratory for Novel Software Technology, Nanjing University

ABSTRACT

Remote Direct Memory Access (RDMA) technique allows the messaging service that directly access the memory on remote machines, which provides low CPU overhead, low latency, and high throughput network transmission. On the other hand, however, due to the limited cache space in RDMA NIC (RNIC), it is still challenging to achieve effective and fair resource sharing across different applications. To address this problem, we present a scalable RDMA as a service to manage resource and deliver fair scheduling to applications' requests. We study the thread contention and preemptive schedule issues at end-hosts, and report the corresponding performance degradation through experiments. Then, we introduce `Avatar`, a model to manage memory and Queue Pairs (QPs) resource for a large number of connections, which eliminates the lock contention and provides fair data scheduling for applications with different priorities. Finally, we implement `Avatar` and demonstrate that `Avatar` can support a thousand of connections, improve the fairness and reduce the requests completion time up to 50% in comparison with the native RDMA.

CCS CONCEPTS

• **Networks** → **Data center networks**;

KEYWORDS

RDMA, Fairness, Scalability

ACM Reference Format:
Haonan Qiu, Xiaoliang Wang, Tianchen Jin, Zhuzhong Qian Baoliu Ye, Bin Tang, Wenzhong Li, Sanglu Lu. 2018. Toward Effective and Fair RDMA Resource Sharing. In *APNet '18: 2nd Asia-Pacific Workshop on Networking, August 2–3, 2018, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3232565.3232636

APNet '18, August 2–3, 2018, Beijing, China
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6395-2/18/08…$15.00
https://doi.org/10.1145/3232565.3232636

1 INTRODUCTION

RDMA allows applications to directly forward requests to RDMA network interface cards (RNICs). Since data can be operated by the RNIC, RDMA minimizes the operating system involvement. With regard to the low data transmission latency, high throughput, RDMA has been widely used by the high performance computing (HPC) community through the InfiniBand (IB) network. Recently, with the decreasing price of RDMA hardware and the compatible design of RDMA over Converged Ethernet (RoCE) standard [3, 4], datacenters have been substantially deploying RDMA in order to alleviate the communication bottleneck for distributed services [9, 10, 18, 20].

Although the RDMA protocol supports multiple transport modes and operations, it is still challenging to achieve high performance when migrating existing applications to RDMA-capable platforms. With regard to the various demands of applications, applying the native RDMA may not be a wise move [5, 11–13]. Given a specific application, the best-fit RDMA-aware system design requires careful parameter tuning, which requires having low-level knowledge of the NIC architecture, operation selections, etc [8, 9, 11, 12, 16, 17]. The developers will encounter the following barriers when building RDMA-aware services.

The ineffective resource sharing. It has been reported that due to the limited cache space on the RNIC, frequent cache misses occurring when a large number of active connections are established [9, 11, 12]. To address this issue, a general approach is to share the RDMA resources, e.g., Queue Pairs (QPs), Completion Queues, among multiple application threads. It is worth noting that previous resource sharing methods are ineffective. At the sender side, one send queue is shared by multiple application threads [9, 12, 16], and these threads need to contend for the lock that protects the send queue. When the number of threads increases, the performance (such as throughput) declines. At the receiver side, application threads need to frequently post free Work Queue Elements to each Queue Pairs, which wastes the memory space. The Shared Receive Queue (SRQ) partially solves this problem. However, SRQs are application isolated which can be shared by multiple applications.

The lack of fair scheduling for applications. RNICs provide multiple hardware priority queues for diverse traffic demand. However, due to the lack of fine-grained preemptive scheduling, the large data segment (up to 1GB) in low priority queue may block the newly arrived data segment of high priority for a long time, which causes degraded performance for latency-sensitive applications. Besides, the Head of Line blocking issue can cause long flow completion time. Therefore, given the multi-queue RNIC, operators demand for fine-grained fair scheduling in the shared environment.

This paper presents `Avatar`, a model that enables effective resource sharing among multiple applications, and provides fair traffic scheduling. We leverage the asynchronous data sending to achieve a lock-free design for QPs sharing, and introduce effective SRQs management to reduce the consumption of memory space and CPU polling for data receiving. Based on the proposed resources sharing design, we further dynamically adjust the en-queue data size of applications to allow fine-grained fair scheduling on RNICs. We implement and evaluate `Avatar` in a small cluster. The experimental results show that `Avatar` can support a thousand concurrent connections and improve the scheduling fairness on RNICs by reducing the request completion time up to 50% in comparison with the native RDMA.

2 BACKGROUND AND MOTIVATION

2.1 Background

The foundation of RDMA operation is the ability of a consumer to queue for a set of instructions that hardware executes. This facility is referred to as a Work Queue (WQ). Work queues are always created in pairs, called Queue Pairs (QP), one for sending operations and one for receiving operations. A Work Request (WR) places an instruction named Work Queue Element (WQE) in the work queue. RNIC executes the WQE without involving the kernel. When the transmission is done, a Completion Queue Element (CQE) is placed in the Completion Queue (CQ) associated with the QP.

Two kinds of transmission semantics can be set for WQEs. One is SEND/RECV verb called channel semantic, and the other is READ/WRITE verb called memory semantic. In this paper, we mainly focus on the channel semantic which is similar to the traditional TCP/IP operation.

For the design of host-side networking to support QoS, each RNIC maintains hardware queues, called Traffic Class (TC). RNIC will map the flow with specified Type of Service (ToS) to one hardware TC. Two kinds of QoS service are supported in RNIC: Strict Priority (strict), Minimal Bandwidth Guarantee [2]. When setting a group of TCs to be 'strict', they are served according to the assigned priorities, which are prior to other non-strict TCs. This property is useful for traffic demand that needs low latency but is not high volume to starve other service running on the same server. For non-strict TCs, Enhanced Transmission Selection (ETS) standard provides the minimal bandwidth guarantee for the traffic. After serving the load in TCs with Strict Priority , the remaining bandwidth is split based on ETS configuration among other non-strict TCs.

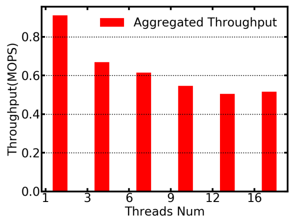


Figure 1: Lock contention on sharing resource causes degraded performance

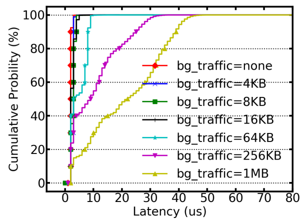


Figure 2: HOL blocking causes long flow completion time

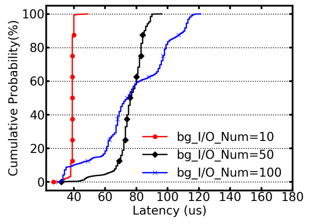
2.2 Limitation of Existing Approaches

Scalability Bottleneck. Several studies have considered the resource sharing to mitigate the impact of limited RNIC cache [6, 9, 11, 15, 16]. FaRM [9] and LITE [16] applied lock mechanism on the shared resources (like QPs). FaSST [11] has demonstrated that with the increase of the number of threads, the scalability bottleneck lies in the mutex_lock. We also ran a micro-benchmark to show the degraded performance due to lock contention (more details in Section 4). As shown in Figure 1, when the number of threads sharing the same QP was 3, the throughput dropped to 66% of the result without sharing. And when the number increased to 9, we observed only 56% throughput left. Therefore such synchronous QP-sharing was not scalable with the increase of connections. We need to design a new model to eliminate the bottleneck.

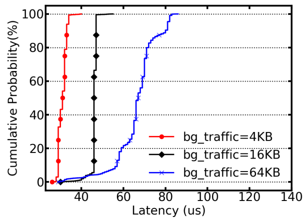
Lack of the fair traffic scheduling on RNIC. Given the resources sharing environment among multiple applications, the operator needs to guarantee the following demands:

- Traffics with the same priority should be served equally, which are not affected by the en-queue sequence of WQEs and the data-segment size.
- The high priority traffic achieves rapid data transmission service, which is not blocked by the traffic with low priorities.

To further exploit the behavior of current RNIC when multiple applications share the device, we considered the following scenarios: large flow and small flow share hardware queue of the same TC; large flow and small flow are assigned to queues of different TCs. We first studied the head-of-line blocking (HoL) by considering traffic' I/O requests coming to one TC. We used two QPs to continuously deliver traffic of same priority to one destination: one for foreground traffic of 1KB data units, one for background traffic of various data sizes. Figure 2 shows the average latency of 1000 times experiment. We observed a significant increase in latency of foreground traffic



(a) Different Number of Background Traffic I/O Requests



(b) Different Background Traffic Sizes

Figure 3: Impact of traffic size and I/O request number on traffic completion time

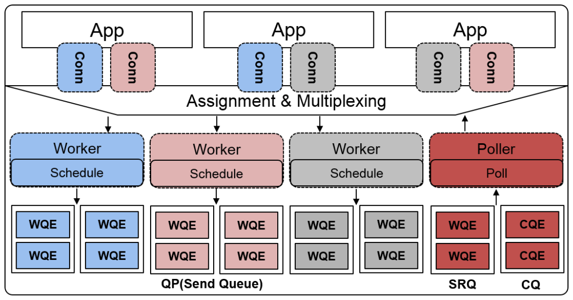


Figure 4: System model

when the data block size of background traffic was larger than 64KB. Unfairness may also happen due to the unbalanced enqueue sequence of WQEs, which causes one traffic delivers more data than the other one. Considering two traffic with the same priority, we varied the number of I/O requests of one traffic. As shown in Figure 3(a), when the number of one traffic I/O requests increased, the completion time of the other traffic increased quickly. Generally, the experiments indicated that current RNIC could not provide fair service for traffic sharing the same TC.

RNICs provide multiple hardware priority queues. The strict priority queues (Strict) get prior service over the non-strict priority queues (configured by ETS). To understand the impact of traffic of low priority on the traffic of high priority, we set up another experiment: we fixed the size of high priority traffic to 16KB and used the low priority traffic as background traffic whose size varied from 4KB, 16KB to 64KB. As shown in Figure 3(b), increasing the low priority traffic size could hurt the transmission latency of high priority traffic. The reason comes to the fact that the current RNICs lack of fine-grained preemptive scheduling. If the high priority queue is empty, the WQE in the low priority queue or ETS TC will be served. During this time, the newly arriving work request with high priority still need to wait until the current WQE transmission finishes, which may cause a long delay of the high priority traffic. The above observations motivate our work to provide fair traffic scheduling at end-host.

3 SYSTEM MODEL

We propose a model called `Avatar` to manage low-level sharing resources like QP, CQ, SRQ and guarantee fair service for high-level services. As illustrated in Figure 4, `Avatar` handles applications' traffic in three aspects: connection establishment, traffic scheduling and traffic receiving. For connection establishment, applications establish connections through the multiplexing layer of `Avatar`. *Connections to the same remote node share the same resources (QPs and the dedicated thread named* Worker*)*. In the shared environment, the traffic is not identified by the QP. Instead, `Avatar` maintains a unique identifier for each connection. The identifier is placed in WRs' `imm_data` domain [1]. After connection is set up, each peer of the connection holds each other's identifier, based on which `Avatar` is able to forward the arrival traffic to the right applications.

The multiplexing layer of `Avatar` also maintains an egress queue for managing WQEs and an ingress queue for managing CQEs, through which we can achieve lock-free resource sharing. When the WR is generated, rather than directly accessing to the underlying QPs, we send the WR to the local buffer registered by the connection establishment component of `Avatar`. Since connections to the same destination share one QP, the corresponding *Worker* will fetch the WRs of connections, and post the WQEs into the associated QP. Thus applications do not need to contend for acquiring the lock of QP to post WRs.

For traffic receiving, since the CQE contains the connection identifier, *in our design all remote QPs are connected to only one local Shared Receive Queue (SRQ), which maximizes the resource sharing*. One CQ is associated with the SRQ and one dedicated thread named *Poller* will poll the CQ and distribute the traffic to the corresponding application threads. More details please refer to [19].

3.1 Applications' Requests Recognition

In synchronous QP-sharing model, application threads need to acquire the lock guard for the QP when posting WRs. We revise the synchronous QP-sharing model to an asynchronous model. We apply the egress queues of the multiplexer layer of `Avatar` to decouple the applications' WRs and QPs forwarding. The *Worker* thread takes WRs from egress queues and posts them to the QP. The problem is that how does the *Worker* quickly find the egress queues with WRs, especially when the total number of connections are large. To address this problem, we adopt the `epoll` function in Linux kernel.

When a new connection is generated in the application, `Avatar` generates an egress queue and an event file descriptor

[1]Currently, we only focus on SEND/RECV operation, user can apply the WRITE-with-Immediate operation for immediate data.

for the connection. The new file descriptor is registered into the event list of the associated *Worker*. After placing WRs into the egress queue, the *Worker* receives a notification of the event file descriptor. Finally the *Worker* maps the event file descriptor to the corresponding egress queue and fetches WRs. The `epoll` can listen up to 65536 event file descriptors. And the time complexity of its events report is O(1).

3.2 Fair Traffic Scheduling

To adaptively adjust the sizes of traffic and make good use of hardware priority queues, we introduce an auxiliary layer, i.e., the schedule in Figure 4. According to the experimental results given in Section 2.2, the lack of fine-grained preemption scheduling for hardware queues of the commodity RNIC can lead to serious HOL blocking and increase the WQE response delay of high priority traffic. *To mitigate the impact of the large data block,* `Avatar` *will split each data block into segments of relatively small size.* By doing so, when the high priority traffic' requests arrive while NIC is processing the low priority traffic' requests, the requests of high priority will be served soon without experience a long delay [1]. For traffic of the same priority, `Avatar` controls the queuing order for traffic. In this case, due to the revised data block size and fair queuing order, traffic with the same priority will be fairly served.

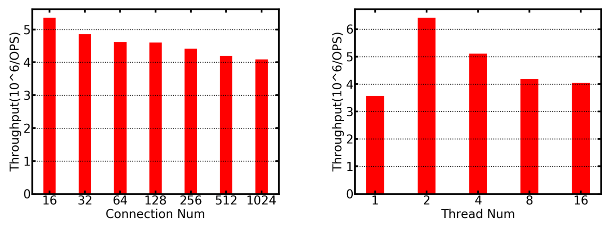
One may wonder that the number of WQEs is limited, and the MTT table size of the NIC is small which can be the resource bottleneck. In our design, the hugepage of Linux (2MB) is used and the *Worker* only posts a fixed number of WQEs to the corresponding QP for each transmission request. For the complexity of implementation, given the whole data blocks, the split data blocks are represented by sequential offsets. Once a split data block is posted to QP, the offset will be added by the length of this data block. When the whole data block is finished, the WQE will be updated to the content of next WQE in the egress queues of the multiplexing layer. Therefore, the state can be easily maintained.

3.3 Data Receiving

As mentioned above, the receiver identifies the arriving data based on the unique identifier of each connection filled in the `imm_data` domain of WRs. When the arriving data is received by the local SRQ, a CQE is posted to the CQ. The *Poller* polls the CQ to get CQEs and check the connection's unique identifier in the field of `imm_data`. Then the *Poller* pushes the received CQEs to the corresponding ingress queues of the multiplexing layer and checks if all the split data segments belonging to the same block are gathered. If it is true, the *Poller* will notify the corresponding application that the data have been successfully received.

Table 1: Evaluation Hardware Specifications

Component	Specification
CPU	Xeon E5-2650v2 (24 cores, 2.1Ghz)
DRAM	4 * 16GB DDR3 DIMMs, 1.6Ghz
RNIC	Mellanox EDR ConnectX-5,100Gbps



(a) Connection Scalability (b) Threads Scalability

Figure 5: Scalability of Avatar

3.4 Polling Thread Working Mode

Finally, we briefly introduce the method to save the CPU resource in our model. All remote QPs are connected to only one local Shared Receive Queue (SRQ). All threads in `Avatar` including *Worker*, *Poller* and the reloading SRQ thread are designed to work in blocked-interrupted-running mode. If there is no task (such as sending WRs to QPs, polling CQEs from CQ, reloading receive WRs to SRQ), threads will be blocked and be waked up by interruption of new events. `Avatar` uses the polling for new tasks until there is no active task for a fixed interval. Those threads will be blocked and wait for interruption events to avoid wasting CPU resources. Since in heavy load environment, *Worker* will keep processing WQEs of connections and *Poller* will keep busy polling the CQ to get CQEs, low latency is guaranteed here.

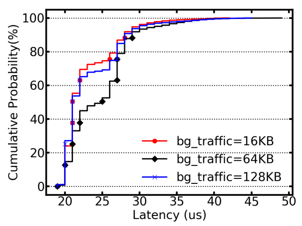
4 PRELIMINARY EVALUATION

We performed our preliminary experiments on two machines. Table 1 shows the hardware specifications of our evaluation.

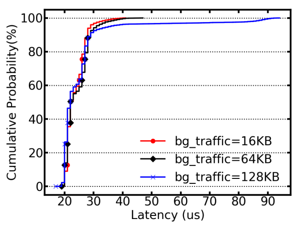
4.1 Scalability

`Avatar` shares the QP for applications that communicate with the same remote node. The scalability of `Avatar` can be tested for two aspects: the throughput of `Avatar` for serving multiple connections sharing the same QP and the throughput of `Avatar` for serving multiple applications' threads.

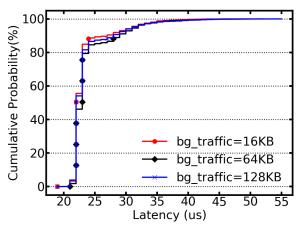
To evaluate the throughput of `Avatar` for serving multiple connections sharing the same QP, we generated N (=16 to 1024) connections operated by 4 threads. Each thread managed N/4 connections and sent 4KB data onto each connection. A circle is defined when one thread has sent 4KB data onto all N/4 connections. We collected 1000 cycles of 4 threads' throughput as the throughput of `Avatar`. The result is shown



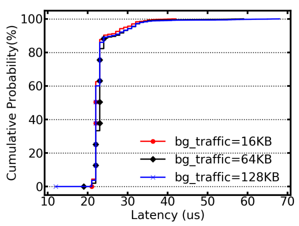
(a) WFQ w/ PRI



(b) WFQ w/o PRI

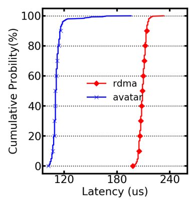


(c) RR w/ PRI

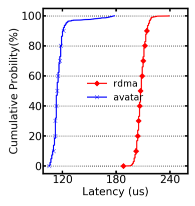


(d) RR w/o PRI

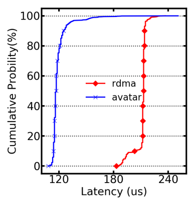
Figure 6: Effect of traffic-split and queuing mechanism



(a) bg_traffic=16KB

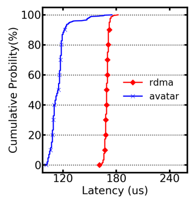


(b) bg_traffic=64KB

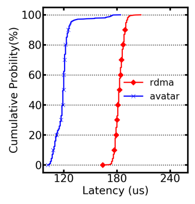


(c) bg_traffic=128KB

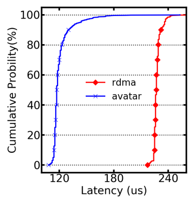
Figure 7: Traffic completion time w/o PRI



(a) bg_traffic=16KB



(b) bg_traffic=64KB



(c) bg_traffic=128KB

Figure 8: Traffic completion time w/ PRI

in Figure 5(a). `Avatar` could support thousands of connections sharing the same QP. It indicated that multiplexing one QP among thousands of connections could reduce the RNIC resource usage but maintain the performance.

Then we evaluated the throughput of `Avatar` for serving multiple applications (emulated by threads). Each thread sent 4KB data onto its remote node. Then 1000 * 4KB data were sent in one cycle. As shown in Figure 5(b), the throughput of `Avatar` is higher than that gives in Figure 1. It indicated that there was no performance degradation caused by the lock contention for the shared QP.

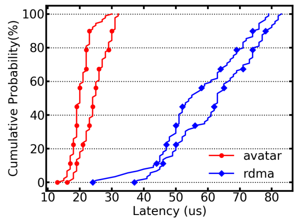
4.2 Fair Traffic Scheduling

To evaluate whether our model could meet the demand mentioned in section2.2, we tested the completion time of a single WQE and the completion time of the whole demand with different background traffic.

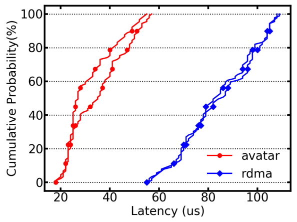
First, we tested the completion time of a single WQE. The WQE of the foreground traffic pointed to a data block of 16KB and the background traffic varied in the region of {16KB, 64KB, 128KB}. The priority setting of the foreground and background traffic was {Strict, ETS} (w/ PRI) or {Strict, Strict} (w/o PRI) respectively. We considered two scheduling approaches, WFQ (Weighted Fairness Queue) and RR (round-robin), at the scheduling layer of `Avatar`. Figure 6(a), 6(b) show the average completion time of a single WQE of the foreground traffic where the scheduling approach is WFQ. We could see that the completion time of foreground traffic was affected by the background traffic when using the setting of w/ PRI. By referring to the results of Round-Robin approach, as shown in Figure 6(c), 6(d), we noticed that the single WQE completion time of foreground traffic was not affected by the background traffic, and the setting of w/ PRI reduced the tail latency of the foreground traffic. Therefore, to meet the fairness demands, we could split the traffic and en-queue the data segments through RR approach in `Avatar`.

To measure the performance of the whole traffic completion time, we ran experiments by using the native busy-polling RDMA and `Avatar` respectively and compared their results. The foreground traffic transmitted 1 MB data in one circle. The average results of 1000 circles were gathered. Figure 7 and Figure 8 show the results by using the priority settings of w/ PRI and w/O PRI respectively. The whole traffic completion time of `Avatar` was improved by about 40% $\sim$ 50% in comparison with the native busy-polling RDMA in both settings. In `Avatar`, multiple WQEs referring to a group of data blocks were posted into QP continuously rather than a large block of data. So the NIC could process the arriving WQEs in pipeline, which improved the performance.

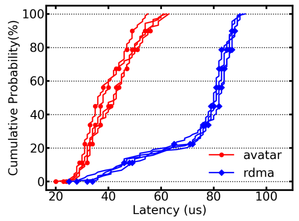
Furthermore, we increased the number of connections to evaluate the performance of fair scheduling. We considered the single WQE completion time by using both native busy-polling RDMA and `Avatar`. In the experiments, we generated the same number of traffic in the foreground and background and tested them in the setting of w/o PRI and w/ PRI respectively. The WQE size of foreground traffic was 16KB and the background traffic was 128KB. Figure 9(a),9(b) show the result when the number of total traffic was 4, the single WQE completion time of `Avatar` was improved by 50% than that of native busy-polling RDMA in both priority settings. The similar result could be observed in the experiment of 8 traffic, as shown in Figure 9(c) and 9(d). The experiments



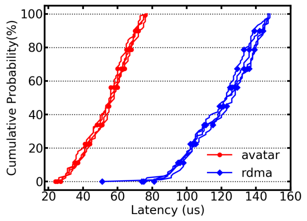
(a) 4 Traffics w/ PRI



(b) 4 Traffics w/o PRI

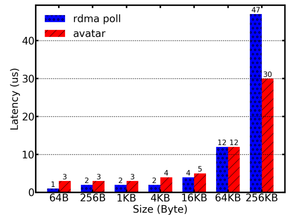


(c) 8 Traffics w/ PRI

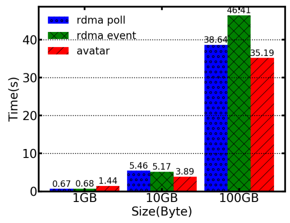


(d) 8 Traffics w/o PRI

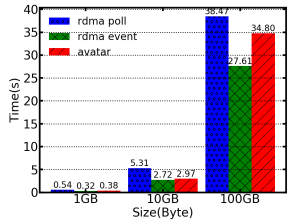
Figure 9: Traffic completion time under multiple connections



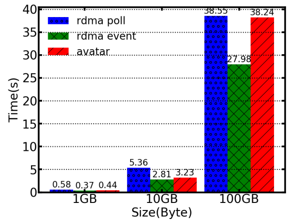
(a) Latency



(b) Real Time



(c) User Time



(d) Total Time

Figure 10: Latency and CPU cost

above demonstrated that `Avatar` could improve network service of applications and provide better QoS for traffic sharing low-level resources.

4.3 Latency & CPU Utilization

For evaluating the latency of `Avatar`, we set up one thread and one connection both in native busy-polling RDMA and `Avatar`. Then the thread sent M (=64B to 256KB) bytes data onto the connection. We collected the result of 1000 circles. As shown in Figure10(a), the extra latency introduced by `Avatar` was about 1-3 us when the data size was smaller than 16KB. The latency of `Avatar` and native busy-polling RDMA were equal under the 64KB data. When the data size was 256KB, `Avatar` costed 37% less latency than native busy-polling RDMA.

For evaluating the CPU cost of `Avatar` and native RDMA (both in busy-polling and event-triggered (called event RDMA)), we fixed the data size M to 1MB and let the thread transfer 1GB, 10GB and 100GB traffic respectively. Figure 10 shows all quotas of CPU usage for native RDMA and `Avatar`. They were measured by the *time* tool in Linux. Figure 10(b) shows that `Avatar` costed more real time (the whole life time of the program) when the data size was 1GB due to the resource initialization and destroy, but less real time than native RDMA when the data size was over 10GB. Figure 10(c) shows that `Avatar` costed less userspace time (the time of program running in userspace) than the busy-polling RDMA but more than the event RDMA. With regard to the kernel time (the time of program running in kernel), all three methods costed almost the same time for transferring 1GB and 10GB traffic. But for 100GB, `Avatar` costed 3.4s while the event RDMA and busy-polling RDMA costed 0.37s and 0.08s respectively. It happened due to the usage of *epoll* function in `Avatar`. In total, `Avatar` costed no more CPU time (userspace+kernel) than the busy-polling RDMA.

5 RELATED WORK

Recently, several approaches have been proposed to improving the scalability of RDMA-based systems. FaRM [9] applied the large page (2GB) and QP sharing to overcome the constraint of limited cache space in RNIC. FaSST [11] applied UD (unreliable datagram) for resource sharing to improve the network I/O performance. LITE [16] allowed applications to safely share resources in the Linux kernel. For reliable communication (RC), FaRM and LITE applied the synchronous mode of resource sharing where the lock can be the performance bottle.

Fair scheduling of traffic is important in datacenter. TITAN [14] provided a fair packet scheduling on commodity NICs which inspired part of our work. Karuna [7] scheduled mix-flows in datacenter. It handled the deadline flows with as little bandwidth as possible, while improved the completion time of non-deadline flows.

6 CONCLUSION

We presented `Avatar`, a model between RDMA applications and RNIC driver to manage RDMA resources and provide fair traffic scheduling. `Avatar` solves two key issues of native RDMA when used in datacenter environments: ineffective resource sharing and lack of fair traffic scheduling on RNICs. `Avatar` demonstrates that effective resource sharing can preserve native RDMA's performance, while fair scheduling improves the applications' performance. Overall, `Avatar` puts forward a new model to take advantage of RDMA.

ACKNOWLEDGMENT

This work was partially supported by National Key R&D Program of China 2017YFB1001801; NSFC Grant No. 61370028; The Collaborative Innovation Center of Novel Software Technology and Industrialization;

REFERENCES

[1] 2011. bql: Byte Queue Limits. https://lwn.net/Articles/469652/.

[2] 2018. Mellanox OFED Linux User Mannual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v4_3.pdf.

[3] IEEE. 802.11Qau. 2011. *Priority based flow control*.

[4] InfiniBand Trade Association. 2014. *Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE)*.

[5] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment* 9, 7, 528–539.

[6] B. Chandrasekaran, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. MIBA: A Micro-Benchmark Suite for Evaluating InfiniBand Architecture Implementations. In *Computer Performance Evaluations, Modelling Techniques and Tools*. 29–46.

[7] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. 2016. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *ACM SIGCOMM Conference*. 174–187.

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *European Conference on Computer Systems (EuroSys)*. ACM.

[9] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *USENIX NSDI*. USENIX.

[10] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *ACM SIGCOMM*.

[11] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX OSDI*.

[12] Anuj Kalia Michael Kaminsky and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *USENIX ATC*. 437.

[13] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *USENIX Conference on Annual Technical Conference (ATC)*. 291–305.

[14] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. 2017. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, Santa Clara, CA, 431–444. https://www.usenix.org/conference/atc17/technical-sessions/presentation/stephens

[15] S Sur, A Vishnu, Hyun Wook Jin, and D. K Panda. 2005. Can memory-less network adapters benefit next-generation infiniband systems?. In *Symposium on High Performance Interconnects*. 45–50.

[16] Shin Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*. 306–324.

[17] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-Hint: An Effective and Reliable Cache Management for RDMA-Accelerated Key-Value Stores. In *ACM Symposium on Cloud Computing (SoCC)*.

[18] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GRAM: scaling graph computation to the trillions. In *ACM Symposium on Cloud Computing (SoCC)*. ACM, 408–421.

[19] Haonan Qiu Xiaoliang Wang, Zhi Wang. 2016. RDMAvisor: Toward Deploying Scalable and Simple RDMA as a Service in Datacenters. http://cs.nju.edu.cn/wangxiaoliang/rdmavisor.

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