Memory Efficient Loss Recovery for Hardware-based Transport in Datacenter

Yuanwei Lu
USTC & Microsoft Research

Guo Chen
Microsoft Research

Zhenyuan Ruan
USTC & Microsoft Research

Wencong Xiao
BUAA & Microsoft Research

Bojie Li
USTC & Microsoft Research

Jiansong Zhang
Microsoft Research

Yongqiang Xiong
Microsoft Research

Peng Cheng
Microsoft Research

Enhong Chen
USTC

ABSTRACT

Limited by the small on-chip memory, hardware-based transport typically implements go-back-N loss recovery mechanism, which costs very few memory but is well-known to perform inferior even under small packet loss ratio. We present MELO, an efficient selective retransmission mechanism for hardware-based transport, which consumes only a constant small memory regardless of the number of concurrent connections. Specifically, MELO employs an architectural separation between data and meta data storage and uses a shared bits pool allocation mechanism to reduce meta data on-chip memory footprint. By only adding in average 23B extra on-chip states for each connection, MELO achieves up to 14.02x throughput while reduces 99% tail FCT by 3.11x compared with go-back-N under certain loss ratio.

CCS CONCEPTS

• **Networks → Data center networks**;

KEYWORDS

Datacenter networks, Loss recovery, Hardware memory

APNet'17, August 03-04, 2017, Hong Kong, China

ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3106993

ACM Reference format:

Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2017. Memory Efficient Loss Recovery for Hardware-based Transport in Datacenter. In *Proceedings of APNet'17, Hong Kong, China, August 03-04, 2017,* 7 pages.

https://doi.org/10.1145/3106989.3106993

1 INTRODUCTION

Hardware-based transport protocol is deployed in datacenters at scale recently. On one hand, hardware-based transport largely frees CPU meanwhile providing ultra-low latency and high throughput (*e.g.*, RDMA [3]), thus has been replacing its software-based counterpart TCP in datacenters [8, 16]. On the other hand, hardware-based transport has been widely used for direct communication among multiple hardware computing endpoints (*e.g.*, LTL for FPGAs [6]), thus enables cloud-scale heterogeneous computation [6, 15].

To provide reliable transmission, lost packets must be retransmitted in transport. Limited by the small on-chip memory, existing hardware-based transport protocols implement go-back-N as their loss recovery scheme [2, 3, 6, 16]. Go-back-N costs very few memory, but is well-known to perform inferior facing even small packet loss ratio. For example, 1% loss rate leads to almost zero throughput in RDMA [16] (§ 2). However, even with PFC [7, 11] enabled, packet loss is still *inevitable* at scale. The loss ratio can often go as high as several percents [9] due to network failures (*e.g.* switch bug) [9, 16] or PFC mis-configurations [10], and usually takes hours or days to detect and mitigate [9]. In such cases, transport performance is dramatically degraded.

Naturally, a selective retransmission [13] loss recovery can greatly improve the transport performance in face of packet

loss. However, unlike software, the limited on-chip resources in hardware brings several challenges to implement selective retransmission. First, because applications may have some assumptions on in-order data delivery, a re-ordering buffer is required to store loss-induced out-of-order data before delivering them to applications. The size of the out-of-order buffer for each connection can be up to several megabytes, which is too much for hardware on-chip memory. [1] Second, selective retransmission requires extra meta data structure at both data receiver and sender to track out-of-order arrived packets. However, a typical meta data structure such as bitmap [13] can also be as big as several Kbs (assume 1KB MTU) for each connection, which again costs too much on-chip resource. [2]

To address above challenges, we propose MELO, a **M**emory **E**fficient selective **LO**ss recovery mechanism for hardware-based transport. MELO enables selective retransmission loss recovery for hardware-based transport with the cost of only ~20B extra memory for each connection, leveraging the following two techniques:

(1) *Architectural separation of data and meta data storage.* MELO buffers packet data in off-chip memory (host memory or on board DRAM), which has enough space (*e.g.*, GBs) to tolerate large out-of-order degree. Meanwhile, MELO maintains meta data in on-chip memory, which is fast enough for frequent query and update (*e.g.*, upon every packet's arrival). With well-designed fully-pipelined logic, MELO can process every packet only using on-chip information, with no need to wait for accessing data in off-chip memory.

(2) *Shared meta data structure.* Observing that the total number of out-of-order packets in all concurrent connections equals $ingress/egress\_bandwidth \times RTT$, MELO adopts a shared meta data structure for all connections and dynamically allocates memory for each connection to track out-of-order packets. MELO can track out-of-order packets for any number of concurrent connections, with the cost of only a *constant* memory.

We have implemented MELO in NS3 simulation with DCQCN. Our experiments show that MELO achieves 14.02x application throughput while reduces the 99% tail flow completion time (FCT) by 3.11x compared with go-back-N when facing certain loss ratio.

[1] Considering a 100Gbps RDMA network with 200us [8] RTT, a connection requires about 2.5MB to store all the out-of-order packets, while the on-chip memory in total is only several MBs.

[2] In current hardware-based transport, each connection usually consumes only hundred of bytes on-chip memory [14]. It would be very undesirable to double the on-chip memory consumption of each connection just of the loss recovery meta data.

2 BACKGROUND

2.1 Hardware-based transport

Hardware based transport gains its popularity as it provides low and stable latency as well as high throughput. RoCEv2 [3] and LTL [6] are two hardware-based transport deployed in commodity datacenters [6, 8]. RoCEv2 is implemented in NIC hardware and provides RDMA semantics, namely READ, WRITE and SEND/RECEIVE to upper-layer applications. LTL is implemented in reconfigurable hardware, *i.e.* FPGA, and is used to connect one local FPGA with a remote FPGA via network. Both RoCEv2 and LTL provide reliable transmission to upper applications. Thus lost packets should be retransmitted.

As a piece of hardware, the size of on-chip memory is very limited, thus there is usually no receive buffer for a connection. Arrived data are placed directly into application buffer (*e.g.* RDMA) or sent to application logic (*e.g.* LTL). Existing hardware-based transport such as RDMA and LTL mandate the order of delivering data to applications. For example, in RDMA, transaction ordering requires certain inter-message ordering [1]. And in LTL, each packet should be delivered to application in sequence increasing order [6]. Thus a re-ordering buffer is required to re-order the out-of-order arrived data before delivering them to applications.

2.2 The necessity of selective retransmission

Both RDMA and LTL assume a lossless network provided by PFC [11]. PFC is a hop by hop in-network flow control mechanism. With PFC, when a downstream switch detects that an input queue is exceeding its threshold, it will send a PAUSE frame to stop the upstream switch which connects to this input port. PFC can effectively remove congestion caused packet drops. On top of this lossless network, RDMA and LTL use go-back-N as their loss recovery mechanism. Unfortunately, in large scale datacenters, failures and PFC mis-configuration caused packet losses are inevitable. For example, C. Guo *et al.* report a persistent 0.2% silent random drop which lasts for 2 days [9]. To make things worse, sometimes the loss ratio of some network switch can be up to 2%. Though rare, once happens, the application performance will be significantly impacted. To quantify the loss impact on go-back-N, we measure Mellanox CX-3 Pro NIC RDMA WRITE throughput under various loss ratios on our testbed. As shown in Fig. 1, application throughput downgrades fast when loss ratio exceeds 0.01%. When loss ratio reaches 0.1%, throughput is close to zero. Selective retransmission [13] can greatly solve the problem, and in this paper, we design

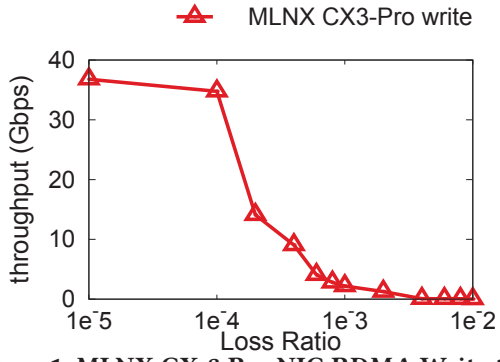


Figure 1: MLNX CX-3 Pro NIC RDMA Write tput.

mechanisms to implement selective retransmission in hardware.

2.3 The necessity of memory efficiency

Unlike software-based transport, hardware-based transport is implemented completely in hardware. As a piece of hardware, the computing resource is very limited as well as on-chip memory (*e.g.*, usually only a few MB). As a result, hardware usually puts its local states to off-chip memory and stores only a small portion of all states in hardware. Then hardware swaps data between on-chip and off-chip memory whenever a certain connection failed to find a match in on-chip memory. However, swapping has a cost, *i.e.* latency and PCIe bandwidth. Thus swapping very frequently would significant reduce the transport performance [12]. As a result, to design the loss recovery mechanism for hardware-based transport, on needs to be very careful not inducing too much on-chip memory footprint, *i.e.* be memory efficient.

3 DESIGN

In this section, we first present the overview of MELO, then we walk through the details of each design component.

3.1 MELO Overview

MELO implements the selective retransmission similar as in SACK TCP [13], *i.e.*, only lost packets will be retransmitted. The system overview of a MELO data receiver is shown in Fig.2. MELO implements the re-ordering buffer in off-chip memory while keeps the meta data, *i.e.* bitmap, in on-chip memory.

Packet loss is detected by a gap in packet sequence number (PSN). The data of an out-of-order packet is stored into the corresponding connection's re-ordering buffer. The data of in-order arrived packets is directly delivered to applications. Meanwhile, the bitmap of the connection will be updated.

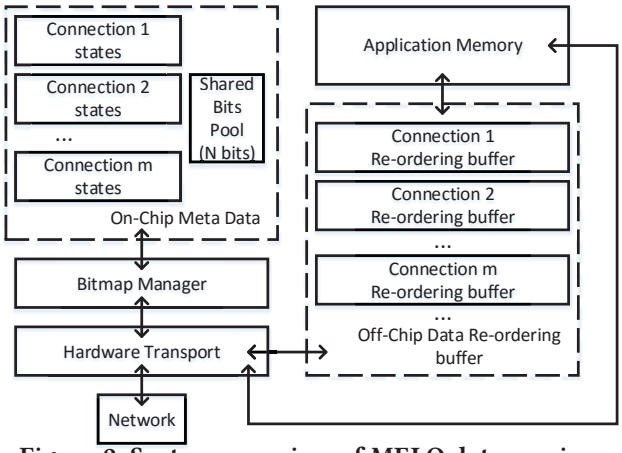


Figure 2: System overview of MELO data receiver.

Once a new hole is formed in the bitmap, a Negative ACK (NACK) will be sent to data sender, which carries most recent three holes in the bitmap. Based on the information in NACK, sender restores the bitmap at the receiver and retransmits those lost packets. If a hole in the bitmap is filled, a consecutive data will be copied from off-chip memory to application. The bitmap manager component is responsible for allocating bits from a shared bits pool to connections. Once the allocated bits of a connection is not enough to cover the incoming packet, a new portion of bits will be allocated from the bits pool. On the contrary, bits will be returned to the bits pool once they are freed by the connection.

Now we will zoom into each design component for a detailed description of MELO.

3.2 Design Details

3.2.1 *Data and meta data storage separation*

MELO stores out-of-order data off-chip due to the limitation of on-chip memory size. For meta data, MELO chooses to store them in on-chip memory. If meta data is stored off-chip, then for each retransmission packet, MELO needs to fetch the corresponding meta data from off-chip memory in order to check whether a hole in the bitmap is filled. The fetching process induces at least $1\mu s$ latency, during which processing pipeline has to stall. Thus MELO stores meta data on-chip to prevent this from happening.

3.2.2 *Off-chip re-ordering buffer*

As discussed before, MELO implements a per-connection data re-ordering buffer in off-chip memory. As off-chip memory is large in size, usually several GBs, a BDP large of buffer is allocated to each connection. Each connection records the base address of its re-ordering buffer in on-chip memory, called *base_addr*. Then the data of an out-of-order packet is

placed into the buffer using PSN as the offset. Along with the packet data, MELO also stores the application buffer address to copy to if necessary (*e.g.* for RDMA, application buffer address is required for later data copy), so that when a block of consecutive data starting from *RCV.NXT* is received, the on-chip logic can signal the CPU (*e.g.* in RDMA) or the data copy logic in hardware (*e.g.* in LTL) to copy the data to the correct application buffer or deliver to application logic.

3.2.3 Bitmap manager

MELO uses a bitmap structure to track out-of-order arrived packets. The bitmap data structure is put in on-chip memory to reduce the latency and ease the pipeline implementation. A naive design of the meta data would require a bitmap for each connection, each tracks a BDP worth of packets. As a result, each connection would require about 2.5Kb for the bitmap. [3]. This is about 1.26x of per-connection states of RoCEv2. [4] If we fix the on-chip die size and maintain a per-connection bitmap, only 44% of the connections can be stored on-chip compared with current RoCEv2 NIC design. This will lead to much more frequent memory swapping between host and NIC, in turn significantly downgrades the performance [12]. In order to reduce the memory footprint of the bitmap, we observe that the total transmission rate of all connections is bounded by ingress/egress bandwidth, which is 40Gbps/100Gbps. Thus not all connections bitmap will be full at the same time. Actually, the total bits required for all connections are
$ingress/egress\_bandwidth \times RTT$. Based on this observation, MELO uses a shared bits pool instead of per-connection bitmap, thus only adds a constant bits pool overhead regardless of the amount of concurrent connections. A bitmap manager module is employed to allocate the shared bits pool to multiple connections as they demand.

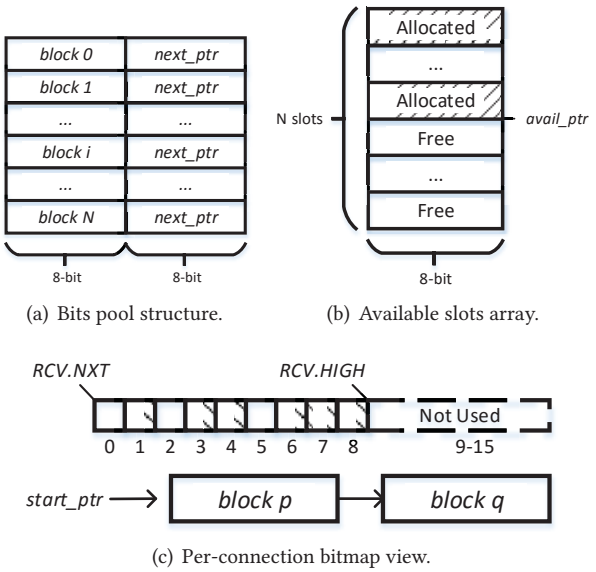
Bits allocation. When designing the bits allocation algorithm, one should use as little on-chip extra management resources as possible and also make an efficient utilization of the bits pool. The first thing an allocation algorithm needs to consider is whether a connection's bitmap is continuous in the bits pool or consists of disjoint blocks of bits. For the former choice, as each connection is very likely to have different bitmap size, it will lead to external fragmentation [5], *i.e.* the total free bits in the bits pool can meet the need of an allocation request but a continuous block cannot be allocated for the request. In order to allocate all the bits in the bits pool efficiently, a compaction mechanism should be developed to compact free bits into larger continous blocks. This however would require non-trivial logic and memory resources as a compaction algorithm has to traverse the bits pool to find free bits and copy data all the way around. As such, MELO chooses the latter approach: multiple disjoint blocks of bits are grouped together to form a connection bitmap.

More specifically, MELO divides the bits pool into multiple equal sized blocks. A free slots array is employed to track free bits in the pool, as shown in Fig. 3(b). Each entry in the array records one block in the bits pool. The array acts as a *stack* data structure: bits blocks are allocated from the stack top and also returned to the stack top, which is pointed to by *avail_ptr*. Any entry with index larger than or equal to the pointer is available for allocation. For the choice of bits block size, there is a clear tradeoff between internal fragmentation and management overhead. Larger size leads to higher internal fragmentation but lower management overhead and vice versa. MELO chooses 1B as the block size to balance internal fragmentation and the overhead of management. If the size of the bits pool is 1024b, MELO sets entry size of the available array to 8-bit. Then 1Kb extra management memory is required to store the available array.

Bits block concatenation. A connection's bitmap may consists of several bits blocks. MELO concatenates bits blocks via a linked list structure. As shown in Fig. 3(a), along with each bits block in the bits pool, there is a pointer points (*next_ptr*) to the next entry for a particular bitmap. As shown in Fig. 3(c), the receiver maintains a *start_ptr* to indicate the first bits block of the bitmap.

One potential drawback of the linked list structure is that it is not efficient for random access to a particular bit, *i.e.*, one needs to traverse the lists from the front to find the corresponding bits block. Fortunately, in our case, the access pattern to the bitmap can be not random at all. The incoming packets are either retransmission packets or new data packets. For new data packets, it needs to access the bitmap from the end of the linked list. For retransmitted packets, usually, it will access the bitmap from the beginning to fill the first hole in bitmap. Access to the middle of the bitmap will happen only when a previous retransmitted packet is lost, then later retransmission packets may need to access the middle of the bitmap to fill holes. However, when a retransmitted packet is lost, the connection will eventually timeout, as a result, all packets starting from the front of the bimap will be retransmitted again. Thus, it's safe for MELO to drop those packets that require access to the middle of the bitmap. As a result, the linked list will be accessed only from the front or the end of the linked list. Two pointers, namely *RCV.NXT* and *RCV.HIGH*, are used to mark the front and end bit, as shown in Fig. 3(c). And a *start_ptr* and *end_ptr*

[3] We assume a 100Gbps network with 200us RTT and 1KB MTU.
[4] In Mellanox CX3 Pro NIC, per-connection transport states are stored in a QPC (Queue Pair Context) data structure (*mlx4_qp_context*), which is 248 bytes.



(a) Bits pool structure. (b) Available slots array.

(c) Per-connection bitmap view.

Figure 3: Structures for bits pool allocation.

is used to indicate the corresponding bits block *RCV.NXT* and *RCV.HIGH* locates in respectively.

To summarize, to maintain a 1024b bits pool, MELO requires 1024b more data for *next_ptr* and another 1024b for the available array. Thus, MELO requires 3Kb in total for bits pool and its management, which remains constant no matter how many connections there are.

3.2.4 Selective acknowledgment generation

Whenever a new hole in the bitmap forms, which can be detected by PSN gap between incoming packets and *RCV.HIGH*, the MELO receiver generates a NACK immediately to the data sender. A NACK carries information about at most three holes in the bitmap. Similar as in TCP SACK [13], the most recently information is reflected to data sender, *i.e.*, holes closest to *RCV.HIGH*. A MELO NACK contains three holes at most to provide some redundancy. To further eliminate the linear search required to find the latest holes, MELO keeps track of the most three recent holes for each on-chip connection. And the holes are updated per incoming packet.

Using information carried in NACK packets, data sender is able to reconstruct the bitmap at the receiver. A MELO sender also maintains a bitmap for each connection using the shared bits pool mechanism. Then data sender does selective retransmission based on the bitmap. Clearly, the size of the sender bitmap is no larger than the receiver bitmap, thus MELO sets the sender bits pool the same size as the receiver one.

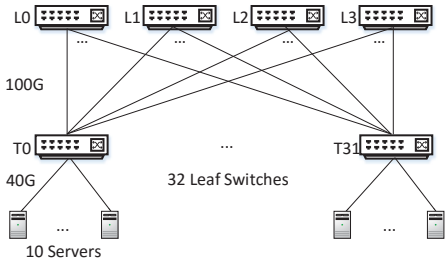


Figure 4: Simulation topology.

In summary, if we implement MELO with RoCEv2, for a network with 1MB BDP (1KB MTU), MELO requires 12B (4B each, 2B for start and 2B for length) to track bitmap holes, 6B (3B each) to track *RCV.NXT* and *RCV.HIGH*, 2B for *start_ptr* and *end_ptr*. And 6Kb for bits pool at sender and receiver. If we consider ~300 concurrent connections [12], in average, MELO requires an extra of 23B for each connection, which is 9.1% compared with already existing states (248B).

3.2.5 Discussion

Not enough bits. Sometimes though rare, network RTT can be extremely large due to PFC pauses or heavy congestion [8]. In this case, our bits pool may not be enough to cover all out-of-order packets. Other cases like a dropped retransmission packet may also put pressure on bits pool. Facing such a condition, MELO will simply drop the packets which failed to be allocated with bits block. As a result, MELO performs a go-back-N loss recovery for those packets. As this kind of condition is rare, MELO still gains good performance for common cases.

Swapping to/from off-chip memory. As we discussed before, the on-chip memory is limited thus it usually acts as a cache to off-chip memory and only store states of a small set of all connections. Upon a cache miss, the hardware chooses a connection to be swapped with the incoming connection whose states are stored off-chip. With bits pool, a to be swapped out on-chip connection should first return all the bits blocks allocated to the available array. Then the to be swapped in off-chip connection will be allocated bits blocks from the bits pool to cover its bitmap.

4 EVALUATION

Now we use packet-level NS3 simulation to demonstrate the power and characteristics of MELO. We have integrated MELO into DCQCN in NS3. [5] Via simulation, we first show the benefits of MELO, *i.e.*, it can improve flow throughput as well as tail latency compared with the original go-back-N mechanism in DCQCN. Then we use targeted experiments to show MELO's bits pool management can scale with the

[5]We thank the authors of DCQCN for providing DCQCN NS3 source code.

number of connections under different packet loss ratio. And we also show that CPU overhead required by MELO to do re-order is low.

Topology: In the following experiment, we use the leaf-spine topology shown in Fig. 4. It contains 4 spine switches and 32 leaf switches. Each leaf switch is connected with 10 servers. The inter-switch link is 100Gbps and server to leaf link is 40Gbps, thus forming a full-bisection network. We set the base RTT to 16μs, MTU to be 1KB and bits pool size to be 1Kb, if not explicitly stated.

4.1 Benefits of MELO

4.1.1 *Improved throughput*

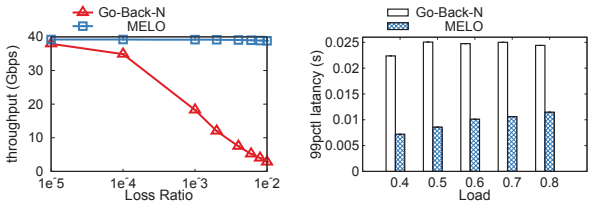
Setup: We choose two servers from Fig. 4 under the same leaf switch T_0. One server sends traffic at full speed to the other server. We manually configure certain loss rate on switch T_0, then measure the throughput of the connection. We compare DCQCN with go-back-N and DCQCN with MELO.

Results: As shown in Fig. 5(a), the throughput of go-back-N drops quickly as loss ratio increases, which matches the results we found in our testbed and DCQCN paper [16]. In contrast, as MELO implements the selective retransmission, only lost packets will be retransmitted, thus the recovery overhead is negligible. Specifically, when drop ratio is very low, *e.g.* 0.001%, MELO runs at nearly full speed and about 3.37% better than go-back-N. Under 0.1% loss, go-back-N only runs at 45.6% of the bandwidth, while MELO achieves 2.14x throughput as go-back-N and runs at 99.9% of the bandwidth. When loss ratio increases to 1%, MELO can utilize 99.0% of the bandwidth, and achieves throughput 14.02x as go-back-N.

4.1.2 *Reduced tail latency*

Setup: In this experiment, half of the servers act as senders and each sends RDMA traffic to one of the other half servers across different leaf switches. We generate traffic with flow size sampled from the web search workload distribution [4]. We further assume the flows arrives according to a Poisson process. By controlling the inter-flow arrival time, we generate traffic with different loads. We configure spine switch L_0 with 1% random drop. We then measure the flow completion time (FCT) of all flows and use the 99% tail FCT as the metric.

Results: As shown in Fig. 5(b), MELO significantly reduce 99% tail FCT compared with go-back-N, *i.e.*, 2.11x~3.11x reduction across various load, mainly benefited from two aspects: 1) It is quicker and more accurate for MELO to detect



(a) Throughput. (b) 99% tail FCT.

Figure 5: Benefits of MELO.

all the lost packets using SACK, thus MELO can retransmit the lost packets to the receiver more quickly compared to go-back-N; 2) Unlike MELO, go-back-N incurs a lot of bandwidth overhead by retransmitting redundant packets (which actually arrived at the receiver), which increases the congestion and impairs performance.

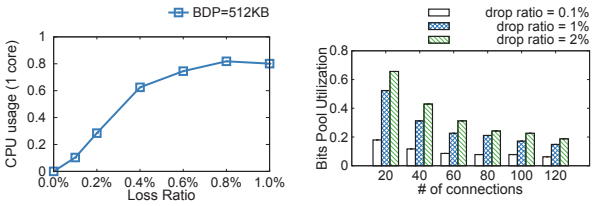
4.2 MELO Deepdive

4.2.1 *CPU usage*

If MELO is implemented with RDMA, the re-ordering buffer should be implemented in host memory and CPU is required to copy data from re-ordering buffer to application buffer. In this case, MELO trades a little CPU for better loss recovery. Now we use an experiment to quantify the CPU usage for re-ordering.

Setup: In this experiment, we configure the network bandwidth delay product (BDP) to be 512KB (RTT ~100μs, which is reported as the 99% latency of a commodity RDMA datacenter by [8]). We simulate the traffic between two servers sending at full speed under the same leaf and configure the switch to be lossy. Then we count the CPU cycles to copy out-of-order data based on the collected packet-level trace. We use one logic core of an 8-core Intel Xeon E5-2630 v3 2.4GHz CPU. The CPU usage is calculated as the fraction of one second to finish copying the trace produced within 1s.

Results: From Fig.6(a) shows that MELO normally consumes very low CPU when loss rate is moderate (*e.g.*, ~30% single CPU core usage on 0.2% loss rate). When the loss ratio is higher, more packets are out-of-order and need to be copied from reorder buffer to application. However, the worst case is bounded to be copying every packet at 40Gbps bandwidth. This causes 82.2% single CPU core usage, which is very low compared with the total CPU capacity in modern datacenter servers (*e.g.*, 32 cores or more). Moreover, the CPU usage can be largely optimized in the future, by using hardware to proactively copy data from reorder buffer to application, instead of using CPU.



(a) CPU usage. (b) Bits pool usage.

Figure 6: Micro-benchmarks.

4.2.2 *Bits pool usage*

Now we use simulation to show that MELO's bits pool can cover the most stringent case loss caused out-of-order, *i.e.* large BDP and multiple connections.

Setup: We set the network RTT to 200μs, which is the 99.9% RTT for a commodity RDMA datacenter as reported in [8]. Then one server initiates multiple connections to another server under a different leaf switch. We configure destination leaf switch to be lossy. Bits pool usage is measured under various loss ratio and amount of connections.

Results: As shown in Fig. 6(b), with the same number of concurrent connections, the lower the loss ratio, the lower the bits pool usage. This is because lower loss ratio leads to less out-of-order, in turn lower bits pool usage. We also observe another interesting result that as the number of concurrent connections increases, the amount of bits required decreases. The reason is that since all flows sharing the bandwidth, each flow has a smaller BDP as the number of concurrent connections grows. As such, a lost packet will lead to less out-of-order packets in a flow, so the total number of out-of-order packets is smaller when there are more concurrent flows. Specifically, with 20 concurrent connections under 2% loss ratio, the bits pool usage is 65.6%. When there is only one connection, the bits pool usage is at its peak, namely 96.9%, which stays the same across all loss ratio.

5 CONCLUSION

This paper presents MELO, a memory efficient loss recovery mechanism for hardware-based transport. MELO implements selective retransmission in hardware, thus it is efficient for loss recovery. MELO only adds a little extra on-chip memory, thus it is memory efficient. Behind MELO is an architectural separation of out-of-order data and meta data storage, *i.e.* data is stored in off-chip memory while meta data is stored on-chip. Moreover, a shared bits allocation mechanism is employed to minimize the on-chip memory footprint of meta data. MELO can improve both throughput and tail latency of applications. To our knowledge, MELO is the first effort to design a memory efficient loss recovery mechanism for hardware-based transport.

REFERENCES

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

[2] 2010. *Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A16: RDMA over converged ethernet (RoCE).* InfiniBand Trade Association.

[3] 2012. *Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE).* InfiniBand Trade Association.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. ACM, New York, NY, USA, 63–74. DOI: https://doi.org/10.1145/1851182.1851192

[5] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2014. *Operating systems: Three easy pieces*. Vol. 151. Arpaci-Dusseau Books Wisconsin.

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

[7] Cisco. 2015. Priority Flow Control: Build Reliable Layer 2 Infrastructure. (2015). http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809_ns783_Networking_Solutions_White_Paper.html.

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

[9] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, and others. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 139–152.

[10] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2016. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 92–98.

[11] ieee. 2010. 802.1Qbb - Priority-based Flow Control. (2010). http://www.ieee802.org/1/pages/802.1bb.html.

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

[13] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. 1996. *TCP selective acknowledgment options*. Technical Report.

[14] Mellanox. 2012. Mellanox EN Driver for Linux. (2012). http://www.mellanox.com/page/products_dyn?product_family=27&mtag=linux_driver.

[15] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, and others. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *international symposium on computer architecture* 42, 3 (2014), 13–24.

[16] 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 Computer Communication Review*, Vol. 45. ACM, 523–536.