Roadblocks of I/O Parallelization: Removing H/W Contentions by Static Role Assignment in VNFs

Masahiro Asada
Nagoya Institute of Technology
Nagoya-shi, Aichi, 466-8555, Japan
asada@matlab.nitech.ac.jp

Ryota Kawashima
Nagoya Institute of Technology
Nagoya-shi, Aichi, 466-8555, Japan
kawa1983@ieee.org

Hiroki Nakayama
BOSCO Technologies, Inc.
Minato-ku, Tokyo, 105-0003, Japan
nakayama@bosco-tech.com

Tsunemasa Hayashi
BOSCO Technologies, Inc.
Minato-ku, Tokyo, 105-0003, Japan
hayashi@bosco-tech.com

Hiroshi Matsuo
Nagoya Institute of Technology
Nagoya-shi, Aichi, 466-8555, Japan
matsuo@nitech.ac.jp

***Abstract*—Achieving 100 Gbps+ throughput with commodity servers is a challenging goal, even with state-of-the-art Data Plane Development Kit (DPDK). Fundamental performance of CPU/Memory is now the bottleneck and simple code optimization of Network Functions (NFs) cannot be the solution. Hardware accelerators including FPGA are getting attentions for performance boost; however, relying on specific features degrades manageability of NFV-nodes. Common Receive Side Scaling (RSS) provides a means of H/W-level parallelization, but per-flow throughput is not accelerated. Existing software-based approaches distribute processing load of NFs, but I/O is still serialized for each datapath. We tackled I/O parallelization and uncovered encounterd certainly misty contentions in our previous study. Specifically, per-thread CPU cycle consumptions proportionally grew as increasing parallelization level, although the overhead of conceivable mutual executions (e.g. CAS operations) was trivial. In this paper, we pursue the cause of the issue and upgrade our I/O parallelization scheme. Our careful investigation of NFV-node internals ranging from application to device driver layers indicates that hidden H/W-level contentions involving DMA heavily consume CPU cycles. We propose a contention avoidance design of thread role assignment and prove our design can flatten per-thread CPU cycle consumptions.**

***Index Terms*—Network function virtualization, Multicore processing, Software architecture, Software performance, Data Plane Development Kit (DPDK)**

I. Introduction

Network Functions Virtualisation (NFV) [1] is a driving force for automating network operations and management, thanks to the various virtualization technologies over commodity systems. However, such supreme flexibility sacrifices performance of packet processing due to the general-purpose design of software/hardware architectures, and Data Plane Development Kit (DPDK) [2] is not a decisive solution for high-end networks [3]. Now Ethernet is aiming at 800 Gbps and 1.6 Tbps wire-speed in a couple of years, meaning that the degradation caused by virtualization will have much severe impact on performance of high-end networks.

The severe situations can be described as follow. In a common usage of DPDK, a Poll Mode Driver (PMD) thread pinned to a dedicated CPU core has to finish its job within 5.12 ns ($\simeq$ 17 cycles at 3.2 GHz) for each packet to handle 100 Gbps traffic of 64-byte packets [4]. This elicits the fact that optimizing per-thread processing is hopeless without a dramatic performance boost of CPU/memory.

Using dedicated hardware accelerations seen in Smart NICs (e.g. [5] [6]) is promising for specific workloads, but network administrators confront practical issues like interoperability and CAPEX. P4 language [7] can be a cushion against the problems; however, highly automated network slicing [8] requires agile service composition and deployment ability that straightforward hardware offloading cannot provide.

Receive Side Scaling (RSS) [9] splits the handling of bundled flows onto separate threads, and is effective on performance in terms of the sum of each throughput. Li et al. [10] examined a parent/worker-based packet-level parallelization, but the gain was limited due to serialized packet I/O. Flow Director [11] can be a basis on parallelizing incoming packets at H/W-level [12], but the state management does not support non-TCP (e.g. QUIC) traffic and is overwhelming for stateless VNFs. Hence, a more efficient and generic packet-level parallelization scheme is necessary.

We examined such a parallelization scheme that can be incorporated into DPDK [13]. Specifically, multiple PMD threads share a single receive queue in a coordinated fashion based on Compare-and-Swap (CAS) operations. As a result, we observed proportional increase in per-thread CPU-cycle consumptions, even though the actual overhead of CAS-based arbitrations was trivial as shown in Section VI again. This implies that there are unexplicit hidden H/W-level contentions for sharing the same receive queue.

In this paper, we pursue the cause of the overhead and revise the design and implementation of our packet-level I/O parallelization scheme. To flatten the increase, we newly adopt a static role assignment approach that is derived from our careful investigation of NFV-node internals ranging from applications to device driver layers. The main contributions of this paper are as follows:

978-1-7281-9486-8/20/$31.00

- A packet-level I/O parallelization scheme is proposed.
- In-depth investigation unveils the hidden contentions.
- Our static role assignment curbs the contention overhead.

This paper is organized as follows: Related work is shown in Section II. We show system requirements of our method in Section III. Then, we explain design and implementation details in Section IV and V respectively. We present evaluation results in Section VI, and discuss remaining issues in Section VII. Finally, we conclude this study and give future work in Section VIII.

II. Related Work

A. Packet-level Parallelization

P. Li et al. [10] have examined a parallelization of VNF internal under the DPDK framework. Two Master-Slave approaches for packet-level parallelization and the traditional RSS-based approach for flow-level one were evaluated using various kinds of VNFs. They adopted a fully software-based packet distribution such that a master thread receives every incoming packet and slave threads perform the main body of the VNF. Even though fine-grained per-packet parallelization is achieved by their design, its performance was below the RSS-based approach because the master thread running on a single CPU core became the performance bottleneck.

H. Sadok et al. [12] have proposed another per-packet parallelization with avoiding the intensive bottleneck. Their parallelization targets TCP flows in a stateful manner, and Flow Director [11] is used to properly distribute incoming packets to worker threads. The evaluation results showed that the processing load of TCP flows was well distributed into the threads, but their approach is not applicable to multi-protocol traffic, especially seen in data center networks.

B. Processor Architecture Awareness

Approaches reducing CPU cache access overhead of servers to optimize packet processing are also proposed. G. Katsikas et al [14] [15] have comprehensively optimized network and servers on service chains by synthesizing NFs and dispatching packets to an appropriate core with tagging. They claim that removing inter-core communication to utilize faster L1/L2 cache is an important factor for performance improvement. For overhead reducing, we focus on the I/O mechanism while their approach aggregates network functions.

A. Farshin et al [4] have proposed I/O optimization by employing slice-aware management which extends Data Direct I/O [16]. Their work adjusts the address of packets being DMA-ed and closes the distance of cache and processing core to lower access latency. While they suppose the undocumented behavior of Intel's modern CPUs, our approach is more generic.

III. Requirement

In this section, we summarize practical system requirements of our packet-level I/O parallelization scheme aiming for accelerating softwarization of high-end network systems.

A. Scalable Performance of per-Flow Processing

Since RSS just separates incoming traffic into a small set of flows, maximum throughput of per-flow processing is not improved. Our parallelization method has to be well-architected to dive into the notion of dividing each flow (i.e. packet-level parallelization). In practice, packet I/O is the remaining part of such a parallelization and its serialized nature should be removed without losing traffic-independence. Packet-level parallelization can introduce the nature of out-of-order to each flow, and therefore, a lightweight and generic re-ordering mechanism is also required.

B. Software-based Parallelization

Hardware offloading is a well-known technique to accelerate performance of specific workload (e.g. host networking [17] [18]), but the varied environment of development and operation spoils the generality and flexiblity of software-based VNFs. For instance, VNFs based on de-facto standard technologies can be easily-ported or lively-migrated to another system environment. Hence, our approach should be embodied without depending on specific (vendor-dependent) hardware implementations.

IV. Proposal

We explain architectural overview of our proposal, a fully software-based and packet-level I/O parallelization scheme, in this section.

A. Problem Statement

As mentioned in Section I, our previous design and implementation of the parallelization was based on the simple access arbitration using CAS operations. The previous evaluation results confronted us incomprehensible performance characteristics such that unexpected increase of per-thread CPU cycle consumptions as well as saturated throughput, even though the fact that overhead of the arbitration was negligible. Figuring out the causes of the phenomena was the major remaining issue in that study.

And now, our careful investigation throughout NFV-node internals (from device driver to application layers) demonstrated that the following single-line of code in mlx5 driver[1] caused the linear growth of CPU cycle consumptions.

```
wqe->addr = rte_cpu_to_be_64(
  rte_pktmbuf_mtod(rep, uintptr_t));
```

Specifically, the code denotes a *supplement* of an empty packet buffer to a corresponding entry of the receive queue. The code itself does not contain explicit contention between PMD threads, which implies existence of hardware-level contention related to DMA. Therefore, we re-design and re-implement our parallelization mechanism to avoid such a contention.

[1] https://github.com/DPDK/dpdk/blob/v19.11/drivers/net/mlx5/mlx5_rxtx.c#L1371 [Accessed: 19-May-2020]

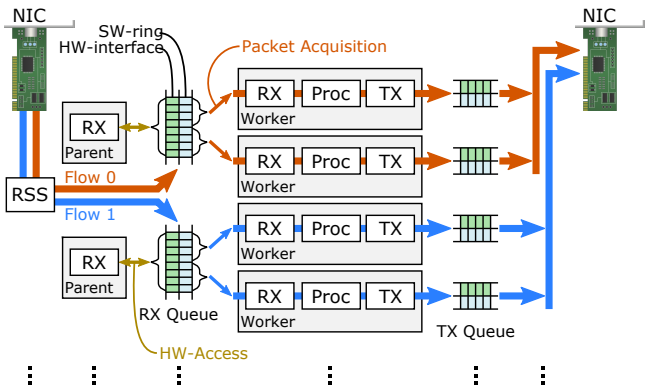


Fig. 1: Overall Architecture of the Proposed Parallelization

B. Architectural Design

We show the overall architecture of our proposal in Fig. 1. Packet reception process at the device driver (details are described in Section V) is newly separated into "Hardware Access" and "Packet Acquisition" to alleviate performance overhead derived from the aforementioned implicit contention. The former process is fully delegated to a parent (master) thread, meaning that it is exclusively responsible for managing the underlying NIC device. The other threads (workers or slaves) perform actual data-plane processing. The important point here is that they access the same receive queue to acquire their packets without touching any contention-prone object, and logical segmentation of the receive queue is also necessary.

Figure 2 illustrates the four design patterns of how threads and/or entries of the receive queue are divided for packet-level I/O parallelization. (a) Default is a way of unparalleled packet I/O processing (a quite-general situation in DPDK). (b) Arbitration is a straightforward parallelization with explicit arbitration (proposed in our previous work). (c) Only Range Division logically separates the queue and exclusively assigns the segments to each worker, which can remove arbitrations for acquiring packets from the queue. (d) The role of Parent is introduced to have the full responsibility of accessing the contention-prone objects in the queue. Note that "Arbitration and P/W" pattern was not examined because of complicated assignment rules between threads and queue segments.

NIC's queues are bound to memory with Memory Mapped I/O (MMIO) [19], making them work as hardware interface, hence concurrent access to queues can be done like memory access. However, access to such memory region triggers actual hardware processing (e.g. PCIe transactions) and consequently parallelization of MMIO access can cause H/W-level contentions, which is hidden from software on the host machine and difficult to be dealt with. That is why we unified the Hardware Access procedure to one thread while our previous methods separate it to multiple worker threads. Such contentions in the Packet Acquisition procedure can be prevented because it does not touch hardware.

Our packet distributing rule is influenced by neither flow

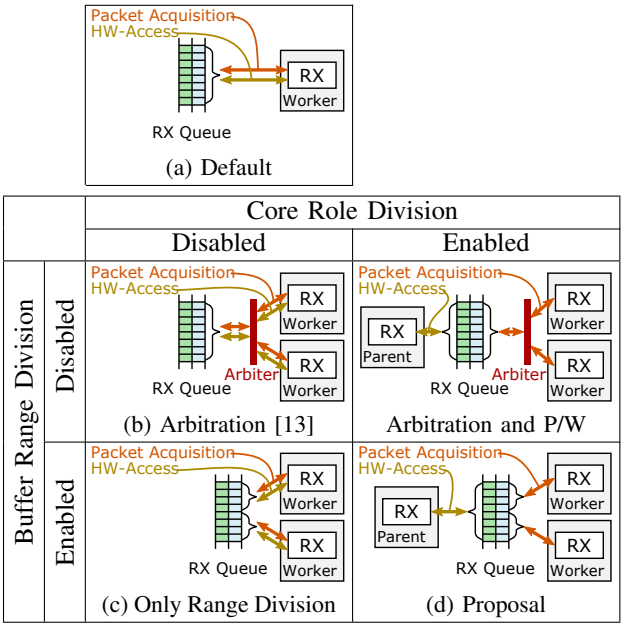


Fig. 2: Design Patterns of Division

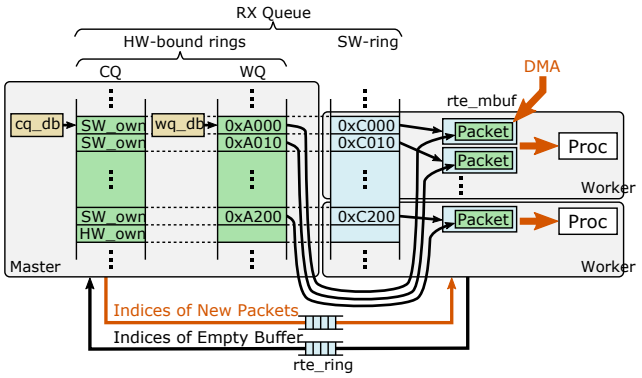


Fig. 3: Implementation of Packet Receiving

nor other data inside of packets because our rule depends on the buffer index range of queues, not packets themselves. Therefore, equal packet distribution and unlimited scale-out can be realized. Moreover, our proposal can combine RSS to increase the degree of parallelism. From the perspective of user applications (network functions), their main logic and packet I/O APIs to handle do not need to be modified.

V. Implementation

We explain the detail of our implementation in this section. We modified a packet receiving function of DPDK's PMD of Mellanox ConnectX-5 (mlx5) to run our method. Our parallelization mechanism is hidden from user applications.

The packet receiving mechanism which we implemented is shown in Fig. 3. The receive queue consists of two HW-bound rings (CQ and WQ) and one SW-ring keeping pointers to packet buffers. Software triggers hardware I/O via these HW-bound rings and their indices (cq_db and rq_db) to be

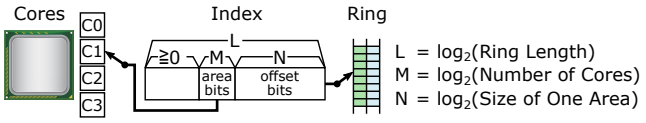


Fig. 4: Range Division Rule Adopted to Rings

notified packet arrivals and update hardware status. We call such processing "Hardware Access". On the other hand, SW-ring is accessed to forward DMA-ed packets to the user application and provide new buffers (Packet Acquisition). The two processing is run by different cores in our implementation while the conventional one is by the same core.

A. Core Role Division

We assign "Hardware Access" to a parent core (the left side of Fig. 3) and "Packet Acquisition" to remaining worker cores (the right side of Fig. 3). To avoid the overhead of simultaneous access to HW-bound rings (refer to Section VI), only one core executes hardware access. Owing to this separation, pure software operation can be extracted from the receiving process and common parallelization techniques can be adopted. And, CAS arbitration is removed in this method because all processings which need arbitration is done by a parent core.

B. Buffer Range Division

To realize equal packet distribution for improvement of per-flow throughput, we divide CQ, WQ, and SW-ring by batch size, and each area is assigned to a specific worker core. To decide the relation of entries of rings and cores with small costs, a buffer index is used as shown in Fig. 4. "Area bits" in the index identifies a worker core to which packets are forwarded, this is independent of flows. Range division and index transmission by a parent core are done after checking the arrival of new packets. Conditions in this paper are $L = 10, M = 0, 1, 2, N = 5$.

VI. Evaluation

In this section, we evaluate the effect of the problems described above by our new design and implementation. We used two physical servers with Mellanox ConnectX-5 cards and they are connected via two 100 GbE links. As shown in Fig. 5, single flow 100 GbE traffic with 64-byte UDP packets is generated by TRex [20] on a server, then it comes back through another server running a forwarding application. This application was impelented based on L2FWD to evaluate our method. The DuT server specifications are shown in Table I.

A. CPU Cycles in the Packet Forwarding

First, we evaluate the number of CPU cycles for packet forwarding at each core to clarify the effects of our parallelization method. Each graph of Fig. 6 shows the average per-packet CPU cycles of receive processing when adding worker cores up to four. 'Baseline' is the number of cycles got from default forwarding implementation. Only Range Division

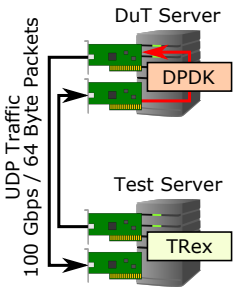
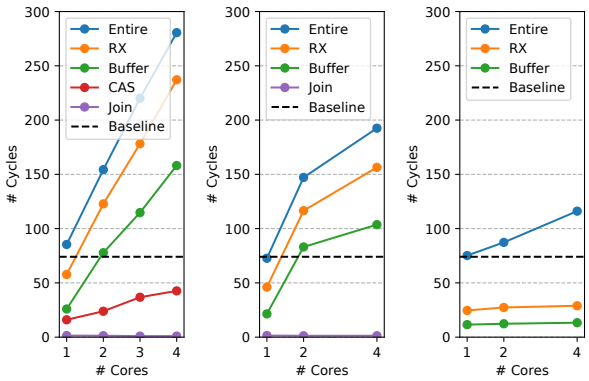


Fig. 5: Environment

TABLE I: DuT Server Specifications

CPU	Intel Core i9-7940X 3.10 GHz 14 cores (HT disabled)
CPU Cache	L1d: 32K, L1i: 32K, L2: 1M, L3: 19.7M
Memory	32 GB DDR4
NIC	Mellanox Technologies ConnectX-5 Ex 100 GbE Dual-Port
OS	CentOS 7.7
DPDK	v19.11
PCM [21]	201902 release



(a) Arbitration (b) Only Range Division (c) Proposal

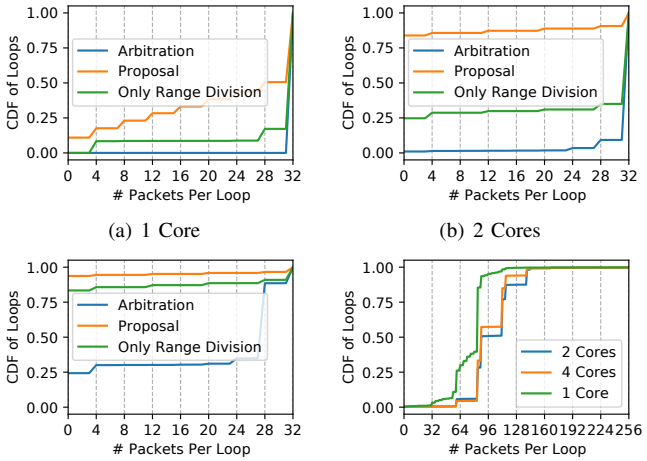
Fig. 6: CPU Cycles in the Parallel Forwarding by Each Core

and Proposal work only when the number of cores is the power of 2 because the range is calculated from bits of ring indices. 'CAS' and 'Join' operations in the graph represent the cost of arbitration for queue indices updating and thread synchronization, and 'Buffer' is the operation to each entry of ring which has no contention in program code. 'RX' is the number of total cycles for packet reception and 'Entire' is the sum of forwarding.

Figure 6(a) shows the result of a simple arbitration approach (previous work [13]). The growth of CAS cycles represents the increasing cost of arbitration between increasing cores. However, Buffer cycles also increase despite the independence from any other core. When applying buffer range division (refer to Section V-B) instead of CAS operation, processing cycles are decreased as shown in Fig. 6(b). Additionally, by applying the Parent/Worker role assignment model, the number of the total RX cycles is kept despite the number of cores as shown in Fig. 6(c). This means the hidden contentions are mostly removed from each core. The gradual rising of entire cycles is caused by the transmitting sequence.

B. The Number of Processed Packets

Next, we evaluate packet batching size to measure the load of cores and check the effect of I/O overhead reduction. VNFs with DPDK receive multiple packets as a batch per RX call `rx_burst()` in a processing loop, and the batch size becomes larger as core load increases and RX call frequency



(a) 1 Core (b) 2 Cores

(c) 4 Cores (d) Parent Thread in Proposal

Fig. 7: CDF of Packet Processing Loops

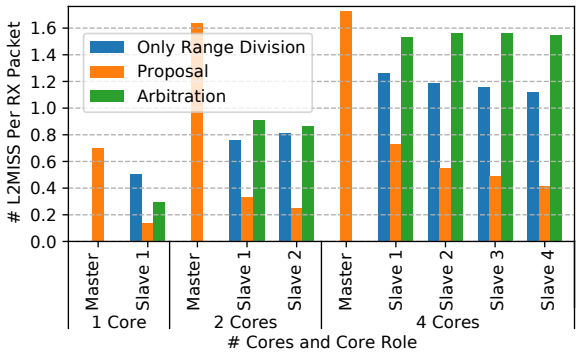


Fig. 8: L2 Cache Miss Per RX Packet

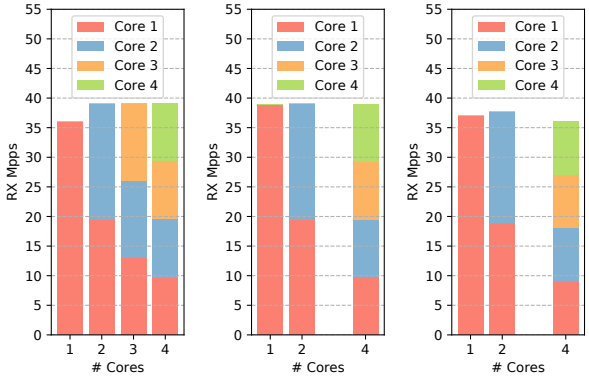
decreases. So, the forwarding core load can be measured by getting the distribution of packet batch size on every loop.

Graphs in Fig. 7 show the Cumulative Distribution Function (CDF) of processing loops handling a batch with the various number of packets. When running at one core, shown in Fig. 7(a), there is no loop in Arbitration except for with 32 packets, indicating an overload: processing speed completely lacks compared to the packet arriving speed. On the other hand, the number of loops in Proposal increases steadily from a small number of packets under less load. Most of the loops handle a multiple of four packets due to the NIC's behavior. As cores are increased up to four, shown in Fig. 7(b) and Fig. 7(c), core load becomes small enough in Proposal.

Figure 7(d) shows how the number of packets per loop was distributed in terms of the parent thread. The parent thread can handle up to 1024 packets (equal to the ring size) on every loop. The result shows that almost all loops handle less than 160 packets, indicating that the load of the parent thread was small enough.

C. Hardware Statistics

To check the change of CPU behavior, we evaluate I/O overhead as a perspective of L2 cache miss of cores. Because



(a) Arbitration (b) Only Range Division (c) Proposal

Fig. 9: Packet Receiving Throughput at Each Core

packets are placed on L2 cache firstly when being treated by cores, L2 misses is one of a major performance factor of high-speed network I/O. Figure 8 shows the number of cache misses per received packet grouped by methods and cores. Note that the master (parent) thread was used in our proposal only because the other methods do not separate the role of the threads.

Our proposal reduces cache misses compared with other methods on worker cores, resulting in less overhead at packet receiving. That is because buffer addresses are reused due to range division and hardware access is removed from worker cores. On the other hand, a huge number of misses on a master core occur as it includes all hardware access and handles all packets. However, these frequent misses on a master core do not become a bottleneck as shown in Section VI-B.

D. Throughput

Finally, to check the performance of our method, we measured forwarding throughput. Figure 9 shows packet receiving throughput at each core when adding cores from one to four.

Overhead contained in each method is clarified when running at one core (shown at the left of each graph). By Changing from Arbitration to Only Range Division, CAS operation is removed and throughput is improved from 36.04 Mpps to 38.98 Mpps. However, the throughput of Proposal is decreased to 37.12 Mpps because inter-core communication is added. When the number of cores is increased, the throughput of each core is decreased and the total is almost the same compared to one core running.

VII. Discussion

We discuss what happens in the datapath from NICs to their device drivers and the cause of performance limitation. As shown in Section VI-A and VI-B, we removed most of the overhead in I/O parallelization. And, there is no longer the bottleneck in the S/W-side because the number of packets per loop is far below the batch size limitation. So we focus on the H/W-side datapath.

TABLE II: Increase of H/W Counters Per Second ($\times 10^6$)

rx_packets_phy	rx_q0packets	rx_out_of_buffer	rx_discards_phy
148.22	36.16	0.6568	111.41

First, the increase per second of NIC's H/W parameters [22] during forwarding are fetched via rte_eth_xstats_* APIs by DPDK, shown in Table II. From the result, the value of rx_packets_phy means that 100 Gbps wire-rate traffic arrived at the NIC. On the other hand, rx_packets_phy means that only 36.16 Mpps are successfully received and the remaining are dropped. Packet drop caused by lack of buffer reflects rx_out_of_buffer and rx_discards_phy, the former relates to buffer provided from S/W, and the latter relates to that handled in the H/W-side. These results imply the bottleneck in the H/W-side processing, not the S/W side.

Next, we estimated maximal packet arrival speed which can be observed from the S/W-side, and found that it is after all 38.90 Mpps. We applied a "fake" function to execute receive processing at maximal speed. Each entry is neither accessed nor got at all, so S/W load is minimal in the condition, and this limitation cannot be exceeded by any S/W-side efforts.

From this investigation, internal I/O processing of NICs should be improved, especially, the load of per-packet processing must be reduced. For example, it can be effective that doing some kind of packet batching in the H/W-side [23]. On the other hand, the I/O performance can be improved when the H/W-side bottleneck is removed. Our study has demonstrated that the S/W-side improvement is possible by parallelization as long as MMIO works as an interface of datapaths.

VIII. Conclusion

While software-based I/O parallelization is required to boost network performance for practical NFV, the simple parallelization technique occurs hidden contentions which is difficult to be resolved. The road map of our work is as follows:

1) Clarifying the feasibility of a packet-level I/O parallelization (in our past work)
2) Removing roadblocks which can be solved in the S/W-side (in this paper)
3) Further investigation and optimization for throughput improvement (future work)

In this paper, we analyzed the current packet receiving mechanism and proposed novel packet-level I/O parallelization. And, we removed all roadblocks against I/O scaling in the S/W-side. On the other hand, remaining H/W-side factors blocking the performance improvement have to be identified and removed to achieve linear performance improvement.

Acknowledgment

This research and development work was supported by the MIC/SCOPE #182106107.

References

[1] "Network Functions Virtualisation," https://portal.etsi.org/nfv/nfv_white_paper.pdf [Accessed: 13-March-2020].

[2] "DPDK: Data Plane Development Kit," https://www.dpdk.org/ [Accessed: 14-March-2020].

[3] R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo, "Evaluation of Forwarding Efficiency in NFV-Nodes Toward Predictable Service Chain Performance," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 920–933, Dec 2017.

[4] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the Most out of Last Level Cache in Intel Processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, New York, NY, USA, 2019.

[5] "NFV Workloads — Networking Applications with Intel FPGA," https://www.intel.com/content/www/us/en/wireline/products/programmable/applications/nfv.html [Accessed: 14-March-2020].

[6] Netronome Systems, Inc., "SmartNIC Overview," https://www.netronome.com/products/smartnic/overview/ [Accessed: 14-March-2020].

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[8] GSM Association, "An Introduction to Network Slicing," 2017.

[9] Ted Hudek, "Introduction to Receive Side Scaling," 2017, https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling [Accessed: 14-March-2020].

[10] P. Li, X. Wu, Y. Ran, and Y. Luo, "Designing Virtual Network Functions for 100 GbE Network Using Multicore Processors," in *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, May 2017, pp. 49–59.

[11] C. B. Robison, "How to Set Up Intel Ethernet Flow Director," https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director [Accessed: 19-March-2020].

[12] H. Sadok, M. E. M. Campista, and L. H. M. K. Costa, "A Case for Spraying Packets in Software Middleboxes," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets '18, New York, NY, USA, 2018, pp. 127–133.

[13] M. Asada, R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo, "Datapath Parallelization for Improving the I/O Performance on NFV Nodes," in *IEICE Technical Report (NS2019-37)*, vol. 119, no. 92, 6 2019, pp. 11–16, (in Japanese).

[14] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. Maguire, Jr., "Metron: NFV Service Chains at the True Speed of the Underlying Hardware," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, Apr. 2018, pp. 171–186.

[15] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire, Jr., and D. Kostić, "SNF: synthesizing high performance NFV service chains," *PeerJ Computer Science*, vol. 2, p. e98, Nov. 2016.

[16] "Intel Data Direct I/O Technology Overview," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf [Accessed: 3-April-2020].

[17] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66.

[18] Netronome Systems, Inc., "Hardware Acceleration for Network Services," https://www.netronome.com/m/documents/WP_Hardware_Acceleration.pdf [Accessed: 19-May-2020].

[19] "How To Write Linux PCI Drivers — The Linux Kernel documentation," https://www.kernel.org/doc/html/latest/PCI/pci.html [Accessed: 19-May-2020].

[20] "TRex: Realistic traffic generator," https://trex-tgn.cisco.com/ [Accessed: 23-March-2020].

[21] "opcm/pcm: Processor Counter Monitor," https://github.com/opcm/pcm [Accessed: 21-April-2020].

[22] "Understanding mlx5 ethtool Counters — Mellanox Interconnect Community," [Accessed: 11-September-2020].

[23] T. Amir and Saeed, "RX and TX Bulking/Batching," in *Netdev 2.1, The Technical Conference on Linux Networking*, Montreal, Canada, 2017.