A SmartNIC-based Load Balancing and Auto Scaling Framework for Middlebox Edge Server

Zhen Ni, Cuidi Wei, Timothy Wood
Department of Computer Science
The George Washington University
Washington D.C., USA
{leonizhen, cuidi, timwood}@gwu.edu

Nakjung Choi
Network System and Security Lab
Bell Labs Core Research, Nokia
New Jersey, USA
nakjung.choi@nokia-bell-labs.com

***Abstract*—Edge Cloud servers running network functions for cellular carriers must provide both high performance and high resource efficiency. Recent research has focused on load balancing across multiple servers in a large data center, yet load balancing within a single host has been neglected. Multi-core servers make use of multi-queue NICs in order to distribute incoming packets to different CPUs; however, existing approaches can lead to an unfair distribution of work since different flows may have significant variations in size and processing time. While the NIC has the detailed information about incoming traffic, it lacks knowledge about the server's resources which prevents the hardware from fully supporting auto scaling and load balancing features. These problems are exacerbated for edge middlebox servers running virtual network functions, where processing latency is critical.**

SmartNICs offer a unique opportunity to resolve this problem with their increased programmability and flexibility. In this paper we present SmartLB - a programmable hardware framework to explore how middlebox servers can work together with SmartNICs to provide a cross-layer solution. Our evaluation shows that SmartLB can increase flow-level fairness, reduces tail latency, and support auto scaling feature for high performance network functions.

***Index Terms*—Edge Computing, SmartNIC, NFV, Load Balance, Auto Scale**

I. Introduction

Modern enterprises deploy middlebox services to improve security and performance in their networks [1]. These middlebox servers run multiple replicas of each network function (NF) to meet the demand. Cloud providers deploy the multi-queue network card to distribute incoming packets to those NFs. While different CPUs poll packets from different queues, the NIC still needs a strategy to decide the packet distribution across all these RX queues. However the current approach can lead to serious unfairness of workload assignments. This is because the real-world network contains elephant and mice flows, and different flows may have significant variations in processing time. Many of the popular load balancer approaches are designed for data centers, which focus on balancing the workloads across multiple computing servers but do not deal with the flow balance on one single host. Current middlebox servers have been struggling from the unbalance problem, with some cores always busy and other cores remaining idle [2].

Receive Side Scaling (RSS) has been widely used for years by state-of-art multi-core systems for flow distribution on the single host. In RSS, the NIC implements a hash function and a packet header's hash value is used to select a CPU. However, RSS may cause imbalance between different CPU cores since it tries to equalize the number of flows assigned to each core, not necessarily the number of packets. Thus a mix of elephant and mice flows can lead to significant differences in processing time, but RSS is not aware of the CPU states. Overloaded CPUs could still receive packets from RSS distribution, and cause high latency or packet drops [3] [4].

In addition, with the explosive growth of cloud, service providers deploy the auto scaling feature to better fit the network status. The service manager starts new instances of NFs during peak hours, and shuts down needless replicas while receiving light traffic to save the cost. However the current packet distribution techniques lack sufficient support for NFs to scale up and down. This is because common approaches, such as RSS, are in stateless mode and do not store the flow information. Any scaling up or down might make the packets from a flow be distributed to a different NF and lead to the loss of flow affinity.

Edge cloud is a type of distributed cloud computing that processes data at the periphery of the network, which is particularly useful for network operators running 5G and next generation wireless services. Instead of having a large scale central data center deal with all the tasks, edge cloud moves partial computing resources and data storage to the distributed stations that are closer to the source. The edge computing can avoid the overhead caused by transmitting all the data to the central cloud [5]. However compared to a normal cloud, the industry is making edge-customized servers with limited resources, e.g., Nokia AirFrame Open Edge Server [6]. This has brought significant challenges to edge providers with traditional hardware-based network functions and could lead to network congestion when facing heavy workload in the local area [7] [8].

The SmartNIC offers a great solution for the edge cloud to better utilize its limited computing resources. The SmartNIC, also called programmable NIC, is a new type of network interface card that provides programmability in the hardware layer, releasing some pressure from the CPU. Similar to CPUs and GPUs, the SmartNICs have multiple micro processing cores that can process packets before they get DMA'd into

host memory. The SmartNIC grants edge-based middlebox servers the possibility to deploy the in-network data processing dynamically. Tasks can be offloaded to the SmartNIC to improve the overall resource utilization and perform better against heavy workloads.

In this paper we seek a way to use SmartNICs on middlebox edge servers to optimize the performance and reduce latency. We present the design and evaluation of SmartLB, a SmartNIC-based approach that makes better decisions, while reducing load on the CPU. The SmartLB framework contains a load balancer and an auto scaler that are fully deployed on the SmartNIC. The SmartNIC itself does not know the host utilization information, so we build a communication channel that enables better cooperation. In summary, our contributions are:

- We offload the load balancing task to the SmartNIC to achieve high performance without consuming any host resources.
- We design a dynamically weighted algorithm to assign flows to ports to better balance the load between them.
- We build a communication channel between the host and SmartNIC to cooperatively scale up the number of network functions.
- We construct a stateful flow table that protects flow affinity when scaling up or down.

We deploy SmartLB on the Netronome Agilio CX 2x10GbE SmartNIC [9] and evaluate the framework using the OpenNetVM [10] NFV platform and the Snort [11] intrusion detection NF. We show that SmartLB framework can quickly respond to the signal sent from host, improve the flow distribution fairness by up to 20%, and reduce the tail latency by more than 15%.

II. Background and Related Work

Network Functions Virtualization: Traditional NFs need to run on dedicated hardware and bring difficulty for the engineers to do NF development and maintenance. Network Functions Virtualization (NFV) provides a new way to develop network functions in the virtualized environment, such as VMs and containers. Modern cloud providers and network operators use NFV technology to improve the service flexibility and save the maintenance cost against the explosive increase of user requests. [12] [13]

NFV Load Balancing: NFV allows multiple NFs to run on the single physical host at the same time. Many state-of-art NFV platforms deploy multiple RX and TX queues on the generic NIC to deal with the flow distribution in the multi-tenant environment. However the NIC still needs a strategy to fairly send the incoming flows to different CPUs. RSS is a common methodology that has been widely used by current systems. RSS uses a stateless hash function and a predefined indirection table to randomly pick a destination for the incoming traffic. However, the RSS approach could lead to CPU congestion because different flows might take different time to process and the generic NIC does not know the utilization information from NFs. It is also possible for RSS to break the flow affinity when facing the NF scaling, because the hash-based solution is stateless and does not store the flow information [4]. RSS++ introduces an optimized RSS-based solution to deal with load balancing [3]. RSS++ modifies the original RSS approach by adding new load balancing timer and bucket reassignment to move flows between cores. However, RSS++ is a software approach running on a dedicated core that consumes computing resource which is limited in the edge cloud. RSS++ also lacks detail about its flow migration, which might cause trouble that one single flow is truncated and distributed to multiple cores.

Data Center Load Balancing: Data centers use different load balancing techniques to distribute network traffic across multiple servers. There are two main approaches exist: static algorithms, which do not take into account the state of the different servers, and dynamic algorithms, which are usually more general and efficient, but require exchanges of information between the different computing units, at the risk of a loss of efficiency. Common load balancers, such as Google's Maglev, try to spread the work evenly to improve the efficiency and optimize the overall performance [14]. In Google's network environment, packets are distributed to the Maglev machines via Equal Cost Multipath (ECMP) and each Maglev machine matches the packets to their corresponding services and spreads them evenly to the service endpoints. Maglev is a stateful approach that stores the flow information in a per-connection state and can be used for future packets. Another type of load balancers, such as Beamer, use the stateless technique to achieve flow distribution [15]. Beamer can be implemented in both software and hardware with P4, and its software prototype is twice faster than Google's Maglev. In this paper we borrow the idea from data center load balancing techniques like Maglev and Beamer, to design a stateful load balance framework with dynamic algorithms that runs on the hardware.

SmartNIC: SmartNIC is the special type of network interface card that supports user-defined program or various data plane applications to be implemented on it. The SmartNIC usually has micro processing cores on the chip to perform data plane operations, and can be treated as Data Processing Unit (DPU). There are many different SmartNIC vendors on the market, which lead to a variety of hardware structures and software programming models. In this project, we are using the Agilio CX SmartNICs designed by Netronome [9]. This card is equipped with 60 programmable flow processing cores and 48 packet processing cores, and over 19MB of memory on the chip. The Netronome card supports the P4 programming language [16] for flow distribution, and also C language as a plugin to develop data plane program on the flows or packets.

There are many projects using the SmartNIC with NFV systems. P4NFV is a unified P4 switch abstraction framework that leverages a host-local SDN Agent to improve the overall resource utilization [17]. The P4NFV project provides an innovative idea to accelerate network processing with the SmartNIC. The Lemur paper introduces a new way of executing NF chains across heterogeneous hardware while meeting

the Service Level Objectives (SLOs) [18]. Lemur offloads NF chains back to the hardware accelerator but remains the flexible deployment and low latency. Researchers from Stanford propose a packet scheduling approach on their SmartNIC [19]. Their framework shares coherent memory with the host server, and instantly incorporates host load feedback into its scheduling decisions. E3 [20] explores what Microservices could be run on SmartNIC with achieving better energy efficiency. They compare the performance of Microservices on SmartNIC and on the host which shows the SmartNIC has lower energy consumption. Also, FlowBlaze [21] builds complex stateful packet processing functions in hardware and hides low level implementations. Pontarelli declares that FlowBlaze could yields 40Gb/s with lower power usage.

III. System Design

In this section we present SmartLB, a hardware-based load balancer that supports the NF auto scaling feature on the host without breaking flow affinity. Our contribution contains two main aspects:

- SmartNIC-based stateful load balancer to distribute incoming flows to different cores;
- Dynamic auto scaler to cooperate with the host NFs;

Problem Formulation: Our design targets Edge NFV systems where multiple replicas of a middlebox service are deployed on a single host. Each replica runs on its own dedicated core(s) and retrieves packets directly from the NIC, either using support for multi-queue NICs or SRIOV (Single Root I/O Virtualization) interfaces. Assigning ports and cores to each replica in this way is important for high performance NFV since it allows optimizations such as having packet data go directly into the correct CPU cache with techniques like Intel DDIO [22].

Typically NFV load balancing is achieved through RSS, which randomly assigns flows to queues or ports based on a hash of their 5-tuple. However, this approach poses two problems: first, RSS tries to randomize the number of *flows* sent to each NF which can cause severe unbalance if there is a skewed workload where some flows have significantly more *packets* than others; second, RSS is stateless, so if the NFV platform adjusts the number of replicas, it will break flow affinity and direct flows to replicas which may not have the necessary state to process them. Alternatively, a software-based load balancer can be deployed within the host's NFV framework, but this requires dedicating more CPU resources and eliminates the caching benefits of having NFs directly receive packets.

SmartLB aims to overcome all of these challenges. Figure 1 shows the structure of SmartLB. The framework consists of a stateful load balancer and a dynamic auto scaler, which are both deployed on to a proagrammable NIC. The load balancer uses a weighted algorithm to forward incoming flows to ports dynamically, and a state table to store the flow distribution result. The stateful design guarantees flow consistency during scaling. SmartLB provides communication channel so the NIC

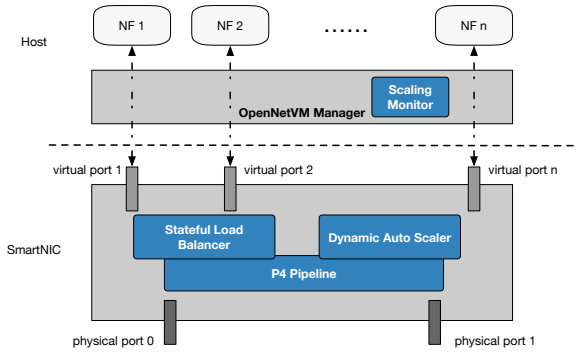


Fig. 1: System Design

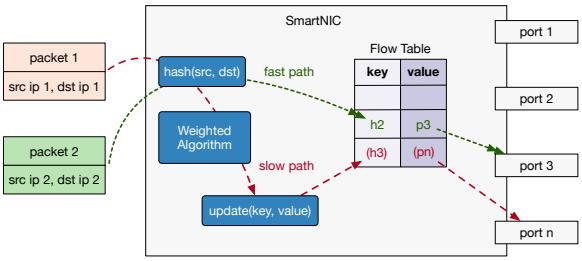


Fig. 2: Stateful Load Balancer

can cooperate with the host when scaling NF replicas up or down.

A. Stateful Load Balancer

The core of SmartLB's design is an algorithm to dynamically assign weights to ports in order to balance the incoming load across them. A port's weight represents the probability of the corresponding port to be assigned a new flow in the next period. SmartLB initializes the weight for each port to equally share the total value (256, in our implementation) when the system starts. For example if there are two NFs running, both ports get a initial weight of 128. Their weights are then used to determine the intervals [0,127] and [128,255] which means the two ports have a equal chance to receive incoming flows from the load balancer. When a packet comes into the pipeline and is determined to be from a new flow, we generate a random number between 0 and 255. The new flow is distributed to the virtual port whose corresponding weight interval covers the random number.

In order to dynamically update the ranges assigned to each NF, we also maintain counters for each virtual interface to keep track of the number of packets received on each interface. The weight, w_i, of each port is dynamically updated at the end of each measurement period using:

$$w_i = \frac{256 * (total - count_i)}{total * (n-1)}$$

where $total$ is the total number of packets received the SmartNIC, and n is the number of the total active ports. Thus the new weights are based on the inverse ratio of the packet counters, so a port which had heavy load in the last period will

be assigned a relatively small weight for the next period so that it can release some processing pressure. In the same way, those ports which receive less packets in the current period will be assigned larger weights and are more likely to receive new flows in the next round. We let SmartLB dynamically update the weights every second in our evaluation. Every time this weight update happens, we reset all the counters to make sure each round is a fresh iteration.

After a new flow is randomly assigned using the weights defined above, SmartLB must record the decision in a state table in the SmartNIC's memory space. In the table, the key is the hash of source and destination IP address and the value is a unique index that corresponds to each virtual port. With the state table lookup, the later packets within the same flow can be directly forwarded without going through the algorithm again. The state table ensures we protect the flow affinity, because the packets will always be forwarded to the same virtual port if there is a hit in the table lookup. We clear the table entry for each flow after the connection is closed to make sure there is always enough memory for active flows.

SmartLB is implemented with P4, which defines the pipeline, and C, which builds the stateful data structure and weight calculation algorithm. The P4 pipeline controls the flow of processing between when a packet arrives at the physical ports and gets DMA'd into the host's memory. We define P4 parsers to extract the header information for the hash functions. Parsers are also used to map those bits in the actual packet into the P4-defined representations, such as metadata. The AgilioCX SmartNICs that we use allow P4 to call custom actions written in C, allowing us to add our more complex stateful functionality that cannot easily be supported in P4 alone.

There are four different kinds of memory on the AgilioCX SmartNIC's chip:

- Local Memory.
- Cluster Local Scratch (CLS).
- Cluster Target Memory (CTM).
- Internal Memory (IMEM).
- External Memory (EMEM).

SmartLB implements the state table in EMEM, which has enough size to support 1.6 million tables entries and latency of 150-500 cycles. All the other data structures are implemented in the IMEM for lower latency.

Figure 2 shows how SmartLB processes the slow and fast paths. When a packet arrives at a physical port, the pipeline extracts IP headers, calculates the hash, and looks up the flow table. If the lookup returns a miss, such as packet 1 in the figure, the packet gets sent to the slow path. Its hash value is passed to the weighted algorithm. The algorithm decides a port to forward, and returns its index. Then the hash value and port index are updated in the flow table for the future packets of the same flow. If the lookup returns a hit, such as packet 2 in the figure, it directly reports the port index to the P4 ingress controller. The P4 ingress controller will read the value from the flow table and controls the hardware to forward packets to the correct port.

B. Dynamic Auto Scaling

An edge cloud data center will need to scale services up or down automatically based on time, utilization, or users' needs. From a practical perspective, the industry is taking a hierarchical approach for different time-scale automation, e.g., non-RT RIC, near-RT RIC in ORAN framework [23]. Similarly, we envision that each server controller can balance workload based on a local observation/decision in the range satisfying a policy given by high-layer decision-making entities. When replicas are added or removed from a host, we need to ensure the load balancer adapts appropriately without breaking per-flow consistency. Our use of a SmartNIC to offload the load balancer makes this more challenging since the NFs and load balancing software are running in two distinct platforms.

To resolve this challenge, we leverage the fact that the SmartNIC can observe all outgoing packets in order to create a communication channel from host to NIC. During the system initialization SmartLB enables the maximum number of SRIOV interfaces, but only a subset of these ports will be marked as "active". All the other unused ports will remain in sleep mode and receive no packets; since SRIOV are virtual interfaces this incurs no overhead to the system. A global variable "activated" is maintained to represent the number of activated virtual port. In the main P4 pipeline we design a scaling program that listens on all egress traffic. This program sits at stage 0, which has higher priority than all other functionalities like table lookup or weight update.

On the host side, the NFV management framework is responsible for determining when to scale up the number of replicas. Our implementation extends the OpenNetVM NF Manager [10] to define a utilization threshold for auto scaling, but our design is applicable to other NFV frameworks as well as more complex scaling algorithms. At run time, the NF manager monitors the utilization data of all CPU cores. If the utilization of a CPU meets a high/low threshold, the hypervisor decides to scale up/down one replica of the NF. After the NF is initialized (or ready to be deactivated), the NFV framework sends out a special packet as the "scaling signal" to the SmartNIC through the virtual port that should be enabled/disabled. We encode the signal in the Ethertype field in the Ethernet header frame. For example, we put 0x7777 in the Ethertype field for scaling up, and put 0x8888 for scaling down (these numbers are not used by common protocols).

The scaling program running on the SmartNIC checks the Ethertype value of all egress packets. If it observes the special flag, the port control function will be called to activate or deactivate the corresponding virtual port. The main P4 pipeline will also get notified, causing it to reset the weight of the target port. If scaling down, we set the weight to zero to make sure that no new flows will be assigned to this port in the future. The port will not get deactivated until the SmartLB observes the FIN of a TCP connection or the flows reach a timeout value. When scaling up, SmartLB rebalances the weights for all ports so new flows will be assigned to the newly activated port.

IV. Evaluation

SmartLB aims to better balance the incoming network flows, and perform higher throughput than state-of-art approaches. In this section we prepare several experiments to demonstrate the power and efficiency of SmartLB. We ask three main questions to our framework in the evaluation:

- How does the load balance algorithm work comparing to RSS?
- How does the SmartNIC perform against same software-based functions?
- Is the dynamic auto scaler "smart" enough to process the request from the host?

In our experimental setup, we physically connect three servers as source server, middlebox server, and destination server. The middlebox is an HP ProLiant GL160 G6 server with two Intel Xeon $X5650$ CPUs @ 2.67 GHz and one Netronome Agilio CX 2x10GbE SmartNIC. The server runs Ubuntu 16.04 with Linux kernel version 4.4.0. On this middlebox server, we run OpenNetVM platform and multiple copies of Snort intrusion detection NFs on the platform. Both the source and destination servers are equipped with the Intel @ $82599ES$ 10 Gigabit Ethernet Controllers to send and receive the traffic.

A. Weighted Algorithm Analysis

First, we evaluate the effectiveness of SmartLB's load balance algorithm under a skewed workload. We run an Nginx web server that stores several files with different sizes. On the source server, we use the WRK2 http client [24] to send requests to the Nginx web server through the middlebox server to download the files. We control the file size and limit the data transfer rate to generate two different types of flows. A light flow is 200KB/s and lasts one second, while a heavy flow is 73MB/s and lasts ten seconds. In this evaluation, we repeatedly create 3 heavy flows and vary the number of concurrent light flows for different test cases. On the host we run 4 copies of the snort NF and do inspection workloads on all packets. We deploy the algorithm in a software version and compare it with the traditional RSS approach by monitoring the throughput received by each snort application. We measure the evaluation result in four metrics: Max/Min throughput, Jain's Fairness Index (JFI), finished requests, and 99% latency comparison.

Figure 3a shows the throughput received by the heaviest snort divided by the throughput from the lightest snort. We gather this metric every second and report the average statistics over each experiment. When Max/Min is higher, it means there is more load imbalance between the most and least loaded snort applications. This ratio is used to illustrate the unbalance level of the framework. We constantly generate 3 heavy flows, and change the number of light flows, shown as the x-axis, to control the skewness level of the whole network traffic. As the number of light flows increases, the skewness level of the total traffic decreases because the light flows are easier to balance even with a random algorithm like RSS. The result shows that our weighted framework is able to achieve less RX rate difference between the heaviest and lightest port, which shows a stronger load balance ability against regular RSS approach. Our algorithm achieves 13% to 20% improvement for network traffic with different skewness levels.

Figure 3b shows the JFI results between the two mechanisms. JFI is a metric commonly used to show fairness in congestion control algorithms. The closer JFI is to 1, the more balanced the flows are on each snort. Our weighted load balancer is always closer to 1 compared to RSS. The result shows that our weighted algorithm can provide better fairness against RSS approach while forwarding the same network flows to ports.

Since a heavy request could consume a whole snort application, a bad load balancer might forward upcoming flows to a busy snort that causes traffic congestion and drops packets if the queue is full which would lead to throughput decrease. Figure 3c shows the number of finished heavy requests of both mechanisms in three minutes. We generate heavy workloads to the middlebox server in order to make the snort applications busy and test the ability of the load balancer to find a relatively idle port. The result proves that with our weighted algorithm the system is able to finish at least 10% more heavy requests during the period whereas RSS drops packets leading to lower throughput.

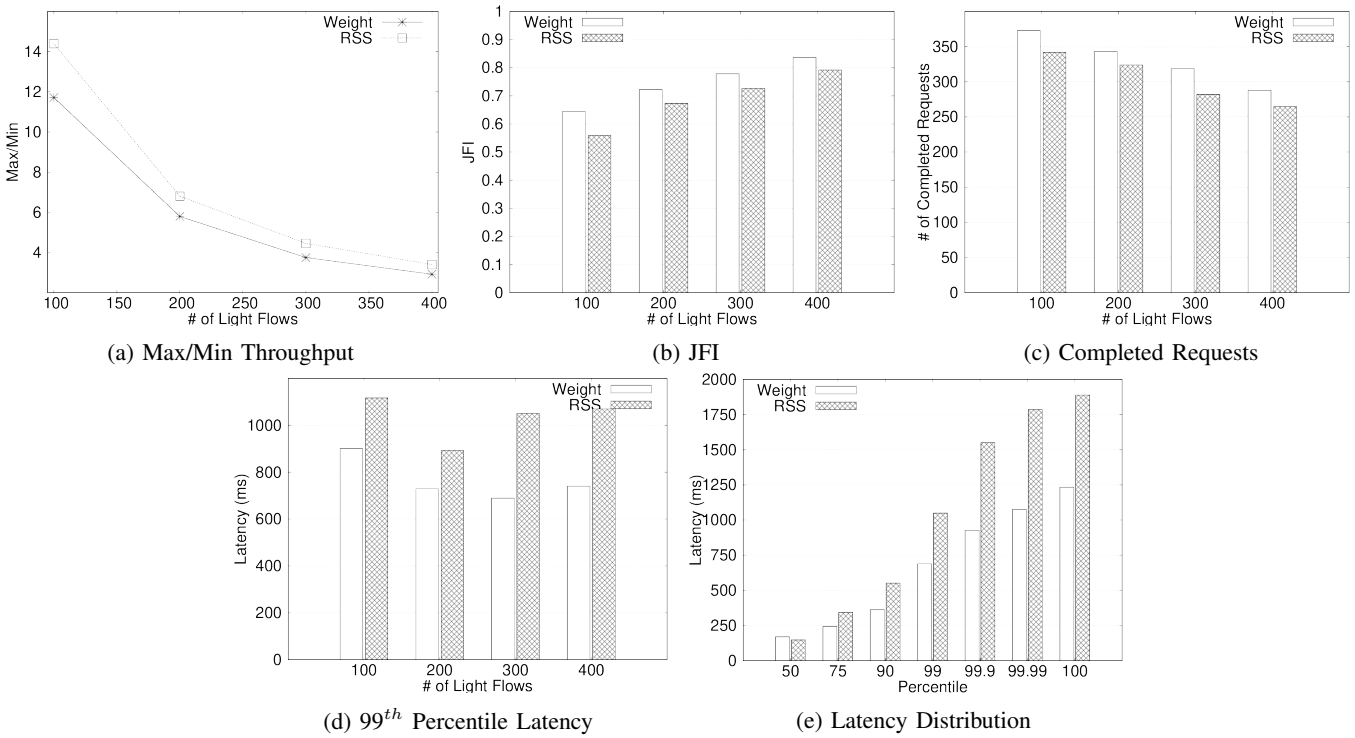
Next we measure the tail latency of the two mechanisms when processing different skewness levels of network flows. Figure 3d shows the 99% response time of the latency sensitive light flows. Our weighted algorithm has much less time cost comparing to the RSS, which means the weighted algorithm achieves quicker response than RSS. This is because SmartLB faster balances the skewed flows with better fairness to avoid the potential traffic congestion. Incoming flows are processed smoothly with less waiting time and leads to lower response time. We pick one of the workloads with 3 heavy flows and 300 light flows and demonstrate the distribution of the request latencies in figure 3e. The graph show that our weighted algorithm has overall less latency time than RSS.

B. SmartNIC vs Software Load Balancer Overhead

Next we evaluate the efficiency of offloading the load balancer to the SmartNIC versus a software approach on the host CPU. We run the same weighted load balancer algorithm on both platforms and compare the performance. For the traffic generation, we use Pktgen on the source server to send several captured pcap files with different sizes of packets to the middlebox server at the full rate of 10Gbits/sec. On the middlebox server, we prepare three different mechanisms to compare the throughput:

- Bridge NFs without any load balancer
- SmartNIC-based load balancer + bridge NFs
- Software load balancer NF + bridge NFs

We deploy the bridge NFs to simply receive incoming packets and send out the same port without any extra computation. Since the Bridge NFs are easily able to meet the 10Gbps rate, this ensures that we measure the performance of the load balancer itself and not the network functions. We use



(a) Max/Min Throughput (b) JFI (c) Completed Requests

(d) 99^{th} Percentile Latency (e) Latency Distribution

Fig. 3: Weighted Load Balancer vs RSS

the first setup as a baseline of this evaluation without any load balancer. In the second case, we run our load balancer framework on the SmartNIC and bridge NFs on the host. In the third case, we program the same load balancer functionality and run it as a network function on the OpenNetVM platform with 1 CPU core dedicated to the load balancer. In all the three mechanisms we have the bridge NFs mirror packets to the sender, and we collect the receive throughput from Pktgen on the source server.

Figure 4 shows the percent of the 10Gbps traffic rate which can be achieved by each of the three mechanisms. The baseline, can meet the 100% rate for all sizes of packets. Neither the SmartNIC or host-based load balancers are able to meet the full line rate for small pacekts because of the flow lookup required for stateful load balancing. The SmartNIC achieves about 40% higher throughput than the SW-based approach for small packets, and even more importantly, it frees up resources on the host since there is no need to dedicate a core to the software load balancer. For 256 byte and larger packets, the SmartNIC is able to meet the line rate, and we expect that as SmartNIC hardware improves, offloading load balancer functionality will become even more beneficial.

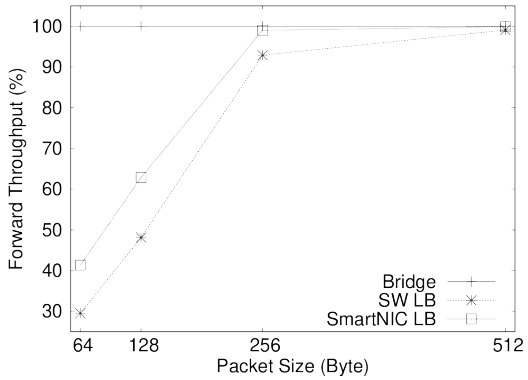
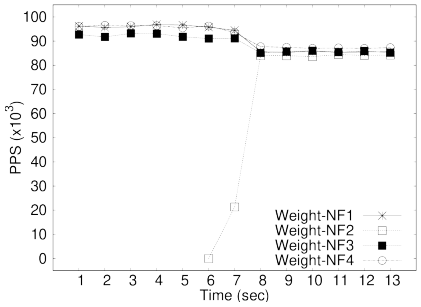


Fig. 4: SmartNIC vs Software

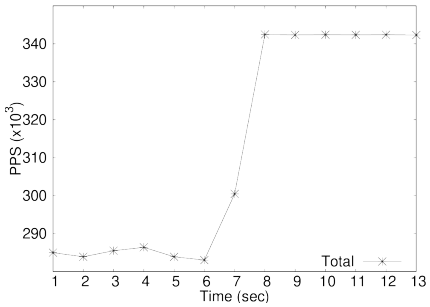
C. Auto Scaler Analysis

Finally, we evaluate the performance of the auto scaling feature. In this test we take the example of scaling applications up when facing heavy workloads. On the host, we run three replicas of the snort application at the beginning. We deploy four virtual ports on the SmartNIC but only the first three are activated for the snort NFs and the last one remains in sleep mode. We generate traffic from Pktgen on the source server and send to the middlebox. We assign packet analysis jobs to all the snort applications and increase the TX rate to make the CPU utilization meet the auto scale threshold.

Figure 5 shows the performance of the auto scaler. The x-axis represents time in second, and the y-axis is the number of packets per second received by each snort application. Figure 5a shows the throughput of all the NFs. The first three snort applications keep busy processing incoming packets, and the hypervisor makes the decision to scale up one more snort at the 6th second. The new snort gets initialized and triggers the scale up signal. The SmartNIC receives the signal, and activate the fourth virtual port for the new snort. The result shows that the whole process takes less than 2 seconds to re-balance the throughput for all snort applications. Figure 5b shows the total throughput received by the system. The auto scaler



(a) Snort Throughput



(b) Total Throughput with Auto Scaler

Fig. 5: Auto Scaler Analysis

can increase the receiving rate and improve the bottleneck from traffic congestion. The new snort NF can quickly reduce some pressure from other busy replicas and increase the total throughput of the middlebox.

V. Conclusion

In this paper we introduced SmartLB, a dynamic SmartNIC-based load balancing and auto scaling framework, that builds a reliable communication channel between hardware and host, and guarantees the flow affinity during the NF scale up or down. We focus on deploying and improving the load balance and auto scale feature for modern middlebox edge servers. We evaluate our framework to compare against the common solution RSS, and demonstrate the ability to achieve better distribution fairness, higher performance and lower latency.

Acknowledgements: This work was supported in part by Nokia Bell Labs and NSF Grants CNS-1814234, CNS-1823270, and CNS-1837382.

References

[1] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, (New York, NY, USA), p. 13–24, Association for Computing Machinery, 2012.

[2] Y. T. Woldeyohannes, A. Mohammadkhan, K. K. Ramakrishnan, and Y. Jiang, "Cluspr: Balancing multiple objectives at scale for nfv resource allocation," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1307–1321, 2018.

[3] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kostić, "Rss++: Load and state-aware receive side scaling," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, (New York, NY, USA), p. 318–333, Association for Computing Machinery, 2019.

[4] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun, "Elastic rss: Co-scheduling packets and cores using programmable nics," in *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, (New York, NY, USA), p. 71–77, Association for Computing Machinery, 2019.

[5] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[6] "Airframe open edge server." https://www.nokia.com/networks/products/airframe-open-edge-server. [ONLINE].

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[8] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 465–478, 2015.

[9] "Netronome - Smart NICs." https://www.netronome.com. [ONLINE].

[10] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A platform for high performance network service chains," in *HotMiddlebox*, ACM, 2016.

[11] "Snort." https://www.snort.org/. [ONLINE].

[12] "Vmware vcloud nfv." https://docs.vmware.com/en/VMware-vCloud-NFV/index.html. [ONLINE].

[13] Z. Lv and W. Xiu, "Interaction of edge-cloud computing based on sdn and nfv for next generation iot," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 5706–5712, 2020.

[14] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 523–535, 2016.

[15] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pp. 125–139, 2018.

[16] "P4 language consortium." https://p4.org/. [ONLINE].

[17] A. Mohammadkhan, S. Panda, S. G. Kulkarni, K. K. Ramakrishnan, and L. N. Bhuyan, "P4nfv: P4 enabled nfv systems with smartnics," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 1–7, 2019.

[18] J. Yen, J. Wang, S. Supittayapornpong, M. A. M. Vieira, R. Govindan, and B. Raghavan, "Meeting slos in cross-platform nfv," in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, (New York, NY, USA), p. 509–523, Association for Computing Machinery, 2020.

[19] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis, "Mind the gap: A case for informed request scheduling at the nic," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, (New York, NY, USA), p. 60–68, Association for Computing Machinery, 2019.

[20] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-efficient microservices on smartnic-accelerated servers," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 363–378, 2019.

[21] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, *et al.*, "Flowblaze: Stateful packet processing in hardware," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp. 531–548, 2019.

[22] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling SLOs in Network Function Virtualization," pp. 283–297, 2018.

[23] "O-ran alliance." https://www.o-ran.org. [ONLINE].

[24] G. Tene, "Wrk2." https://github.com/giltene/wrk2.