vSwitchLB: Stratified Load Balancing for vSwitch Efficiency in Data Centers

Xin Yin*†, Enge Song†, Ye Yang†, Yi Wang†, Bowen Yang†, Jianyuan Lu†,
Xing Li*†, Biao Lyu*†, Rong Wen†, Shibo He*, Yuanchao Shu*□, Shunmin Zhu‡†□
*Zhejiang University †Alibaba Cloud ‡Tsinghua University
alibaba_cloud_network@alibaba-inc.com

ABSTRACT

The virtual switch (vSwitch) serves as a fundamental element in cloud network, critical for high-performance and strongly isolated inter-VM forwarding in local and external networks. Similar to other multicore systems, a vSwitch with multiple cores also faces the issue of core load imbalance. As a major cloud provider, we pinpoint four cases of core load imbalance within the vSwitch in our cloud, stemming from unequal traffic distribution across virtual queues and RSS buckets, as well as from traffic patterns like heavy hitters and micro-bursts. To tackle the different load imbalance cases, we present vSwitchLB, a vSwitch load balance framework. Specifically, we introduce a load imbalance detection module, accompanied by dedicated techniques designed to address each specific type of imbalance. Our preliminary evaluation shows that vSwitchLB can accurately classify different load imbalances encountered in the vSwitch on our cloud and then prevent any single core of vSwitch from being flooded and overwhelmed.

CCS CONCEPTS

• **Networks** → **Data center networks**; Network performance analysis; **Bridges and switches**.

KEYWORDS

Load-balancing, vSwitch, Data Center

ACM Reference Format:
Xin Yin, Enge Song, Ye Yang, Bowen Yang, Jianyuan Lu, Xing Li, Biao Lyu, Rong Wen, Shibo He, Yuanchao Shu, Shunmin Zhu. 2024. vSwitchLB: Stratified Load Balancing for vSwitch Efficiency in Data Centers. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024), August 3–4, 2024, Sydney, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663408.3663422

1 INTRODUCTION

Modern cloud networks are highly multiplexed environments that operate a multitude of workloads on shared infrastructure. Among these, virtual switch *vSwitch* is the crucial component [7, 30] to satisfy a diverse range of traffic demands that span the compute-intensive requirements of high-performance computing (HPC) [20], the high volume and rapid transmission requirements of big data and storage applications [13, 22, 29], and the low-latency requirements of real-time data processing tasks [3]. These varying workloads are heavily reliant on the vSwitch's capability to facilitate efficient and reliable network communication. Nevertheless, the presence of load imbalance within multicore vSwitch, poses significant challenges to meeting the Service Level Objectives (SLOs) [8] across various services [25].

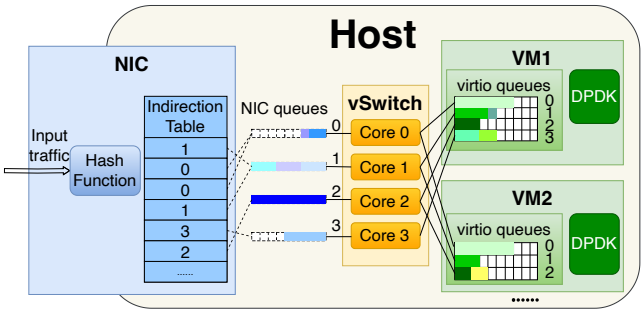


Figure 1: The architecture of the vSwitch datapath.

In multicore machines like virtual machines (VMs) or within a single server, load balancing proposals [5][18][27][17] focus on devising algorithms capable of either directing traffic flows across multiple cores or dynamically scaling the number of cores to adapt to fluctuating traffic loads. Metron [18], which employs a 'traffic-class' approach, proposes a system in which an overloaded core triggers the redistribution of half its traffic classes to a queue of a different core. However, it is class based and unable to load balance traffic which cannot be split. RSS++ [5], using a flow-based methodology, dynamically modifies the Receive-Side Scaling (RSS) indirection table by monitoring real-time load, thus shifting buckets from overloaded cores to underutilized ones. Despite its improvements over the static flow distribution of traditional RSS, RSS++ [5] may encounter difficulties with individual buckets swamped by multiple large-volume or 'elephant' flows.

In addition, the industry community has introduced a variety of technological solutions. Open vSwitch [9] with a Data Plane Development Kit data path [10] (OVS-DPDK) offers queue-based assignment algorithms: 'roundrobin', 'cycle', and 'group', which execute a singular sorting of queue loads that changes numerous queue-core mappings [11]. The extensive reorganization carries the potential to decrease throughput and induce significant fluctuations. The DLB [14] accelerator within the Intel Xeon Scalable Processor [15] offers a packet-based solution, dynamically delegating network traffic at the packet level across CPU cores, effectively

□Co-corresponding authors.

APNet 2024, August 3–4, 2024, Sydney, Australia

ACM ISBN 979-8-4007-1758-1/24/08
https://doi.org/10.1145/3663408.3663422

Table 1: Dynamic Proposals on Multi-core Load Balancing.

Scheme	Scenario	Granularity of Traffic Scheduling	Resolving Load Imbalance caused by elephant flows	Additional cores for central scheduling
Shenango [24] and Shinjuku [16]	Intra Server	Request	No	Yes
Metron [18]	Intra Server	Traffic-class	No	No
RSS++ [5]	Intra Server	Flow	No	No
Dyssect [6]	Intra Server	Flow	No	No
vSwitchLB	Virtual Switch	Queue, Flow, Flowlet	Yes	No

preventing overload scenarios. Nonetheless, DLB's [14] binary operational mode, which is either fully active or inactive, on activation, disperses all forms of traffic, which may lead to a potential decrease in throughput and an increase in overhead.

Current methods excel in their specific contexts, but struggle with finer-grained traffic imbalances. Queue-based solutions [11] fail to handle high loads in individual buckets, as moving such queues can overload cores. Flow-based approaches [5][18] falter with core imbalances from elephant flows. While per-packet dispatching [14] seems straightforward, within high-performance demanding contexts, the advantages it provides are insufficient to offset the increase in overhead and the subsequent decline in throughput. A comprehensive solution is needed to mitigate workload imbalances at various levels effectively and efficiently, minimizing overhead, and maintaining consistent throughput.

To address the above issues, we introduce vSwitchLB, a load balancing system specifically engineered for vSwitch environments. vSwitchLB is composed of two principal components: the detection and migration modules. The detection module is developed based on a comprehensive analysis of the inherent and distinguishing features of load imbalance cases observed within our cloud. Specifically, we classify cases of load imbalance into four distinct types, each originating from unique causes and manifesting through diverse traffic patterns. These causes range from unequal traffic distribution across virtual queues and Receive-Side Scaling (RSS) buckets to traffic patterns characterized by heavy hitters and micro-bursts. The migration module is crafted with targeted strategies, including queue-, flow-, and packet-level balancing techniques, devised to address the specific types of imbalances. By accurately detecting the type and applying the corresponding method, vSwitchLB can effectively address the issue of load imbalance.

2 MOTIVATION

In cloud network production clusters, despite the adoption of high-bandwidth access links, the technological evolution of other host resources remains stagnant [1], particularly in respect to CPU speeds and cache capacities. As articulated by hostCC [2], the performance bottlenecks within the host have emerged as the 'shortest planks' that retard the advancement of high-performance network infrastructure. In response to this challenge, We strive to augment host network performance by mitigating core load imbalances in vSwitches, an endeavor achieved through the sophisticated orchestration of interactions between the NIC and CPUs.

Following, §2.1 outlines the vSwitch datapath architecture, with §2.2 delving into NIC-CPU data path structures to identify reasons behind vSwitch load imbalances. Challenges in achieving core load balancing within data center vSwitches are discussed in §2.3.

2.1 vSwitch Datapath Architecture

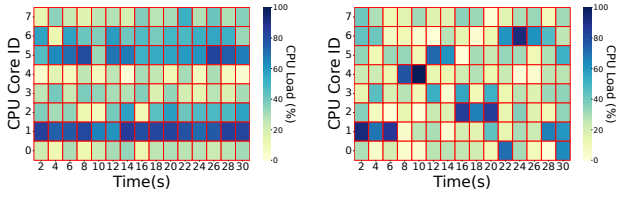
Network functions virtualization (NFV) is revolutionizing the deployment methods for network services, and within this infrastructure, the virtual switch emerges as a vital component. In contrast to traditional physical switches, the virtual switch functions as a software layer hosted on general-purpose commodity servers. It ensures the facilitation of packet processing and switching functions, vital for maintaining both interoperability and critical management aspects such as security and flow control. Furthermore, the vSwitch distinguishes itself by offering flexible functionalities and rapid deployment capabilities, which are not intrinsic to physical switches, thereby effectively meeting the scalability demands of NFV networks in data centers.

The architecture of the vSwitch datapath is depicted in figure 1. The vSwitch connects on one side to the physical network interface cards (NICs) on commodity servers, and on the other side to the virtual ports of virtual machines (VMs), providing a set of routing and switching protocol stacks to facilitate communication between virtual machines and between virtual machines and the external network.

Typically, packets that are received from the physical NICs are allocated to different queues based on rules supported by the hardware, such as Flow Director [26] and RSS. The multi-queue capabilities of NICs allow for queues to be affinitively bound to distinct processing cores, which assists in the distribution of packet processing across cores.

An example shown in Figure 1 illustrates the aforementioned process: the physical NIC on the left distributes incoming packets to various queues in accordance with RSS rules. RSS employs a hash function to calculates a hash value from specific packet fields the five-tuple of the incoming packet. Due to the indirection table's size (e.g. 512 entries), the hash is reduced (e.g. using the lowest N bits or modulo 512) to index the table. In the indirection table, flows that map to the same entry belong to the same bucket. Based on the result of the indirection table lookup, packets are placed in the appropriate queue—for instance, NIC queue 2—for the corresponding CPU core. NIC queue 2 has been pre-bound with affinity to core 2 of the vSwitch, guaranteeing that the packet channeled through NIC queue 2 is processed by core 2 of the vSwitch.

As depicted on the right side of Figure 1, virtual machines communicate with the vSwitch using the Virtio interface through virtqueues. Packets sent from the virtqueues of a VM are allocated across multiple processing cores in accordance with the affinity binding established between the virtqueue and the cores of the vSwitch. To elaborate, packets originating from virtqueue 0 of VM1 are directed to core 0 of the vSwitch for handling, and packets from virtqueue 2 of VM2 are directed to core 2 of the vSwitch for processing.



(a) Spatial (b) Dynamic

Figure 2: Workload Imbalance Across CPU Cores of vSwitch.

2.2 vSwitch Load Imbalance

Popular cloud applications such as web search, databases and network functions often utilize multiple CPU cores to process multiple requests concurrently in order to fulfill high performance demands. The subsequent issue, critical and widely discussed, arises with the workload distribution among cores in multicore machines. As outlined in Table 1, systems such as Shenango [24] and Shinjuku [16] employ additional cores to orchestrate request scheduling centrally. Yet, applying additional core to execute centric scheduling mechanisms for per-flow or even per-packet traffic management greatly strains the scheduling cores' computational power and heavily burdens the system's bus bandwidth. In scenarios with unpredictable traffic patterns, static allocation algorithms often fail to distribute loads effectively. Existing dynamic scheduling algorithms also fall short in addressing all variations of load imbalance, such as those induced by elephant flows [21, 28]. It is our assertion that reliance on multicore workload scheduling algorithms alone is insufficient in the context of vSwitchs. Traffic scheduling must be specifically designed and tuned in accordance with the particularities of the network environment.

Figure 2(a) and Figure 2(b) visualizes the CPU usage rate of vSwitches in data centers collected from the real world, showing the phenomenon of imbalanced load in both spatial and temporal dimensions. Combining the introduction of the virtual switch datapath in the previous subsection with the imbalanced load data, we have gained a deeper insight into the root causes of load imbalance. Next, we will reveal the essence of load imbalance in vSwitches from the perspective of finer-grained traffic.

Traffic Imbalance in Virtual Queues Leading to load imbalance (Type I). To enhance packet processing performance, the vSwitch binds virtual queues to specific CPU cores, a practice known as Affinity Binding, thus preventing frequent context switches and cache misses. However, when certain queues experience excessive load, this may lead to load imbalances, as depicted in Figure 3. Since this binding relationship tends to be relatively stable over a short period, it generally results in an imbalanced effect as depicted in Figure 2(a).

Traffic Imbalance in RSS buckets Leading to load imbalance (Type II). By recording the CPU cycles occupied for processing traffic per queue, we have observed that some load imbalance can be attributed to high loads on single queue, typically those on the NIC side, leading to core overload. The traffic in a single NIC queue originates from flows corresponding to multiple RSS buckets (entries in RSS's indirection table). RSS-based flow hashing ensures packets from the same flow are directed to identical RSS buckets, enabling parallel, in-order processing across cores. As depicted in

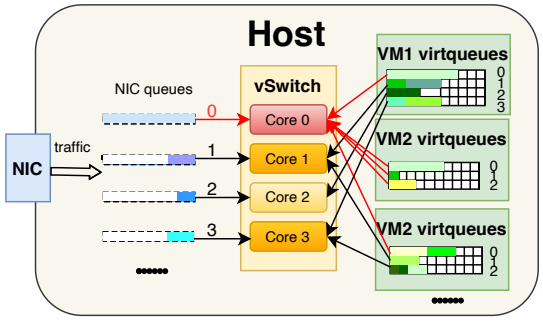


Figure 3: Traffic Imbalance in Virtual Queues Leading to load imbalance (Type I).

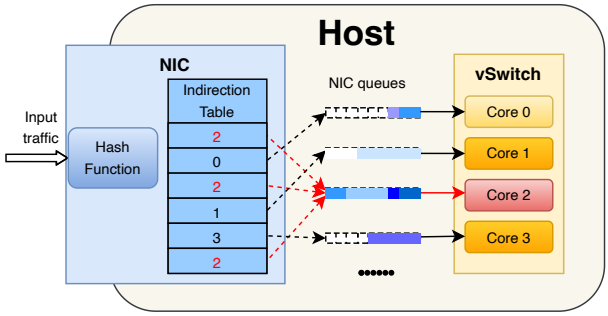


Figure 4: Traffic Imbalance in RSS buckets Leading to load imbalance (Type II).

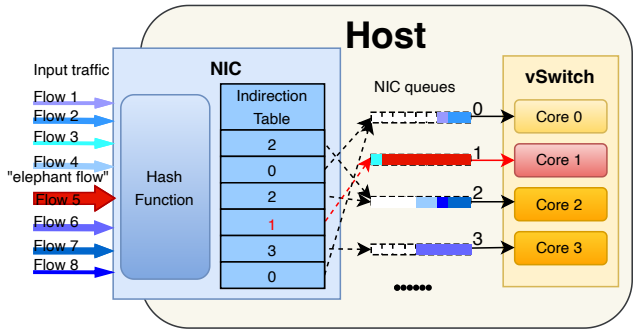


Figure 5: Heavy hitter Leading to load imbalance (Type III).

Figure 4, an imbalance can occur due to an overloading of a single queue when the RSS hash maps an excessive amount of traffic to those buckets. Such a scenario of load imbalance in a vSwitch is also illustrated in Figure 2(a).

Heavy hitter Leading to load imbalance (Type III). The traffic entering data centers is unpredictable, and we have observed some unusual flows leading to load imbalance in vSwitches. One such instance is the so-called "elephant flow", where a single heavy hitter from its inception at the NIC cannot be split, with all packets of the elephant flow being processed by the same core as shown in Figure 5. The affinity binding design, originally intended to ensure that packets from the same flow are processed sequentially by the same core to enhance network efficiency and avoid cache misses, unfortunately becomes a limitation, resulting in CPU core overload as illustrated in Figure 2(a).

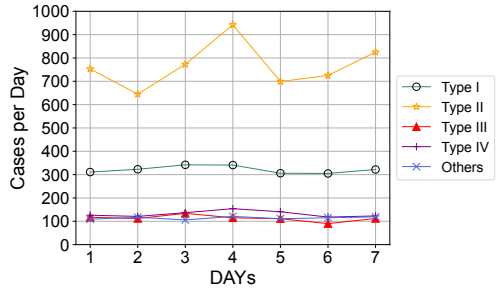


Figure 6: Occurrences of Different Types of Load Imbalance in 800,000 vSwitches Over 7 Days in Alibaba Cloud.

Table 2: Distribution of Different Types of Load Imbalance in 800,000 vSwitches Over 7 Days in Alibaba Cloud.

Different Types	Cases	Proportion
Type I	2247	22.2%
Type II	5362	53.0%
Type III	791	7.8%
Type IV	917	9.1%
Others	798	7.9%

Micro-burst Leading to load imbalance (Type IV). Another scenario involves anomalous client traffic, leading to the rare occurrence of load imbalance in vSwitches, as illustrated in Figure 2(b). Overloaded traffic exhibits imbalances not only in spatial distribution (across different CPU cores) but also exhibits rapid, dynamic changes over time. To elaborate, the traffic fluctuates swiftly, causing multiple cores to overload in succession within a continuous time frame; for instance, core 1 may experience a spike in load at one moment, followed by a reduction in load on core 1 with a subsequent spike on core 4.

Distribution of Different Types of Load Imbalance. Figure 6 presents the data on vSwitch load imbalance observed in data centers housing 800,000 servers, collected over a period of seven days. The statistical information in Table 2 details the total number of occurrences of each type of load imbalance and the proportion of each type. Type I to Type IV represent the quartet of load balancing classifications previously discussed. The category labeled 'Others' encompasses failures not identified within the system's load capturing parameters.

In our setup, data collection is facilitated by the vSwitch. The *constant_tsc* feature of the CPU allows for precise CPU utilization measurements by quantifying the ratio of active to total time. Continuous monitoring of CPU utilization allows us to detect load imbalances, which we define as a scenario where the maximum core utilization rate exceeds 80%, and the range of utilization rates across cores is greater than 40% over a period of two minutes. Upon detecting such imbalances, a finer granularity of data collection is initiated to categorize different types of load imbalance, focusing on CPU cycles consumed by each virtual and physical network interface queue and packet counts per bucket within the queue. A detailed discussion follows in §3.1.

2.3 vSwitch Load-balancing Challenge

Within the context of cloud networking environments, it is imperative to account for an array of practical constraints. These impose challenges upon the endeavor of achieving core load balance within vSwitches.

Categorizing accurately and resolving various types of load imbalances. According to the literature and the latest technological surveys, there is no one-size-fits-all solution that is effective in all cases. Drawing on the analysis of the imbalanced workload of vSwitches outlined in the previous section, it is essential to accurately identify different scenarios of load imbalance and rapidly implement effective measures to mitigate the imbalance.

Ensuring User-Transparent Remediation. During the process of alleviating load imbalances within a vSwitch, changes may occur in the flow-to-core allocation relationships, such as reassigning queues or flows to different processing cores. Regardless of the method employed to mitigate load imbalances, we aim for the entire process to be smooth and seamless, meaning it should not impact the NIC throughput, nor cause excessive jitter and request latency. Ultimately, it is crucial to ensure that users are unaware of the process, thus guaranteeing quality of experience (QoE).

Designing Resource-Efficient, Lightweight Solutions. In data center servers, the majority of resources, such as CPU, memory, storage, and network bandwidth, are typically allocated to support virtual machines that are sold to customers. Within vSwitches, CPU resources are highly valuable. Consequently, the resources available for solutions addressing load imbalances in vSwitches are limited. It is crucial that the designed solution bears this in mind, striving to be as lightweight as possible to avoid excessive consumption of hardware resources.

3 SYSTEM DESIGN

In this section, we will introduce vSwitchLB, a vSwitch load balancing system composed of detection and mitigation modules.

3.1 Detection Strategy

The detection module enables real-time identification of the type of load imbalance. With considerations for resource efficiency and lightweight operation, the detection module consists of two parts: coarse-grained sampling and fine-grained category.

Coarse-Grained Sampling. The primary objective of Coarse-Grained Sampling is to detect any occurrence of load imbalance among the cores of a vSwitch. Upon meeting the imbalance criteria, this triggers the subsequent fine-grained category phase. The main indicators of imbalance are related to the core utilization. Let $U = u_1, u_2, ..., u_n$ represent the set of utilization for n cores. Taking a set of CPU utilization samples every two seconds, we calculate the maximum value of the elements of the matrix U_{max} and the range $R = U_{max} - U_{min}$ to quickly determine the imbalance, evaluating whether they reach predetermined thresholds U_s and R_s. Let T_s denote a continuous sampling period of T_s minutes. An imbalance is detected within the period T_s if there are M instances where $U_{max} > U_s$ and $R(U) > R_s$, indicating persistent overutilization of certain vSwitch cores and an uneven overall load distribution.

Fine-Grained Category. Upon the confirmation of a load imbalance within the cores of the vSwitch, a more fine-grained measurement is initiated to collect detailed data. In order to categorize vSwitch load imbalances into four types, the fine-grained category primarily evaluates two categories of metrics: load distribution and

dynamic characteristics, corresponding to the spatial and temporal features of load imbalance in vSwitches previously identified.

Specifically, for dynamic characteristics, the CPU utilization at each moment is considered as a set, and the dynamic time warping (DTW) distance is calculated between each set of data U over a period of time. Through dynamic characteristics, one can rapidly discern if the variable falls under the category of 'unusual single-flow' marked by swiftly fluctuating anomalous client traffic (Type IV load imbalance). If it does not meet the threshold D_s of highly dynamic variation, the examination shifts towards the load distribution features.

Load distribution primarily measures two indicators: the CPU cycles consumed by a queue and the number of packets for each bucket within the queue. Flows that share an identical entry in the shared RSS indirection table are placed within the same bucket. The CPU cycles expended by each queue within the core may be ascertained through the invocation of OVS [9] command-line utilities. Each core tracks the number of packets received by each bucket. To do so, each core maintains a table of the same size as the number of RSS's [5] indirection table entrys and uses the low order bits of the hash as an index into the table and increments a simple counter. The CPU cycles consumed by queue i are denoted as Q_i, and the number of packets for bucket j of queue i is denoted as B_{ij}.

Type III load imbalance is identified when the packet count of an individual bucket exceeds 80% of the aggregate packet count across all buckets within the core, fulfilling the condition $B_{ij} > 0.8 \times \sum_{q=1}^{n} \sum_{b=1}^{m_q} B_{qb}$ (m_q indicates the number of buckets in queue q). This pattern typically suggests the presence of an elephant flow or multiple significant flows concentrated in a single bucket within a queue, creating a hot spot that can lead to performance degradation. Type II load imbalance manifests when a singular queue persistently accounts for over 80% of the total CPU cycle utilisation of the core, which is characterised by the condition $Q_i > 0.8 \times \sum_{k=1}^{n} Q_k$, yet no singular bucket within the queue disproportionately exceeds the packet count when compared with other buckets in the same queue. In this case, the overall CPU cycle consumption is heavily skewed towards one queue, indicating a possible bottleneck at the queue level. Type I load imbalance, occurs when multiple active queues are associated with the same core, and no single queue nor any bucket within those queues dominates the consumption of CPU cycles or the packet count.

3.2 Stratified Load-balancing Mechanism

The mitigation module employs a multi-tier design with methods that operate at multiple levels of granularity for various types of load imbalance scenarios. Moreover, the chosen methods guarantee a smooth and imperceptible experience for users by minimizing the volume of traffic migrating across cores.

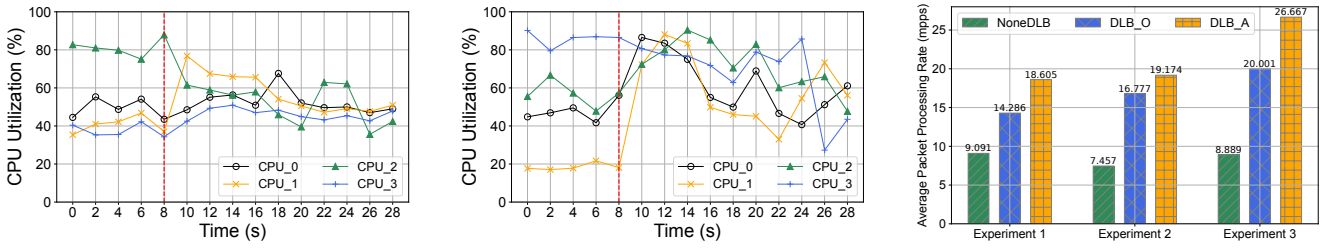
Dynamic Queue Reassignment. Dynamic queue assignment is employed to resolve Type I load imbalances. Taking into account the current core load conditions, we employ a queue policy to balance the load by reassigning the affinity bindings between queues and cores. During each migration operation, the core with the highest CPU utilization (U_{max}) transfers queues to the core with the lowest CPU utilization (U_{min}). Initially, the algorithm converts the CPU cycles consumed by all queues Q_i on the overloaded core into core utilization rates proportionallyC_i.$C_i = \frac{Q_i}{\sum_{k=1}^{n} Q_k} \times U_{max}$ Then they are sorted in descending order to create an array indicating the capacity of the queues' consumption. Applying a greedy algorithm, we compute all possible combinations from the array that meet specific conditions: the sum of combinations must be greater than a preset threshold MIN to avoid migrating low-traffic queues or empty queues, and less than (U_{max}-U_{min}) to ensure a reduction in imbalance. Additionally, the combination should comprise the fewest possible elements (to minimize migration traffic). If the resulting queue index is empty, then Method Two, dynamic flow rebalance, is employed.

Dynamic Flow Rebalance. Similar to the queue policy, the flow policy addresses Type II load imbalances by migrating flows from overloaded to underutilized cores. Flows mapped to the most heavily loaded core (with CPU utilization U_{max}) are migrated to the least loaded core (CPU utilization U_{min}). The counters of buckets B_{ij} in the overloaded core are proportionally converted into core utilization rates P_{ij}. $P_{ij} = \frac{B_{ij}}{\sum_{b=1}^{m_q} B_{qb}} \times U_{max}$. After sorting the buckets' counters in descending order, we obtain an array indicating CPU consumption capacity. The same algorithm used in the queue migration applies here — using a greedy algorithm to solve the array's full combination, where the sum of the combinations must be greater than the preset threshold MIN and less than (U_{max}-U_{min}), with the fewest possible combination elements to minimize migration traffic. Drawing on inspiration from RSS++, once the flows to be migrated are identified, we rewrite the indices of the corresponding entries in the indirect table, causing part of the flow from the overloaded core to move to the least loaded core. If no flows are identified for migration, Method Three, the DLB [14] Elimination, is invoked.

Similar to RSS++ [5], ensuring sequential packet delivery during dynamic flow rebalance involves migrating an RSS bucket between cores, with the handover finalized post verification of packet processing by the initial core and monitoring packet counts during table updates to validate the transition.

Dynamic Load Balancer (DLB) Elimination. This method addresses types III and IV load imbalances caused by abnormal traffic, utilizing the fourth-generation Intel Xeon Scalable Processor's hardware accelerator, DLB [14]. Combined with DPDK [10], DLB [14] dynamically distributes network traffic at the packet level to multiple CPU cores for processing while maintaining the order of packets after CPU processing. Through the distribution of traffic in flowlets, DLB [14] fundamentally addresses issues of load imbalance. Nonetheless, DLB [14] functions in a binary mode – which is either entirely active or completely inactive. Upon activation of the API, all data streams on the core, encompassing both elephant flows and abnormal traffic, are dispersed, leading to reduced throughput and increased overhead. Consequently, the activation of this method is reserved for infrequent and anomalous circumstances.

DLB's [14] atomic and order queues ensure packets are delivered sequentially. Atomic flows are single-core processed, maintaining order, while ordered stages balance loads and preserve original packet sequence across parallel workers.



(a) Mitigating Type-I load imbalance through dynamic queue reassignment

(b) Mitigating Type II load imbalance with dynamic flow balance

(c) Balancing heavy hitter to multicores with dynamic load balancer

Figure 7: Load balancing with vSwitchLB.

4 PRELIMINARY EVALUATION

Experimental Settings. We deploy two servers, connected through a Ethernet link, with one configured to act as the sender and the other as the receiver. The virtual machines running on these servers communicate with each other through OVS [9]. Additionally, this configuration supports the vHost Multiqueue [12] feature, which enhances the network performance of VMs by allowing multiple send and receive queues for VM network interfaces.

Our preliminary evaluation focused on three scenarios of vSwitch load imbalance: Type I, II, and III. The experimental results are illustrated in the Figure 7.

We resolved Type I and Type II imbalances by invoking dynamic queue reassignment and dynamic flow rebalancing respectively. Dynamic queue reassignment is contingent on the OVS-DPDK [9][10], which enables the allocation of RX queues via PMD commands [11]. In Figure 7(a), at the 8 second mark, the queue from the overloaded core 2 is quickly migrated to core 1, effectively mitigating the load imbalance. For the purpose of dynamic flow rebalancing, akin to RSS++, our system utilizes FastClick [4], an advanced iteration of the Click Modular Router [19]. Similarly, in Figure 7(b), rewriting entries in the indirection table to redistribute flows originally on Core 3 among other cores achieves load balancing.

We conducted DLB elimination experiments with the Intel Xeon processor 8480+. We generated three elephant flow patterns of varying sizes, distributions, and rates, corresponding to 'Experiment 1', 'Experiment 2', and 'Experiment 3' in the figure 7(c). The response of the vSwitch cores to the Type III load imbalance scenario was assessed by evaluating packet processing rates and load distribution across cores with DLB disabled, DLB order queue, and DLB atomic queue enabled, corresponding to NoneDLB, DLB_O, and DLB_A respectively. Without DLB, packet processing speed is constrained by the capability of a single core since packets from the same flow are processed by one core, leading to heavy hitters on the processing core and load imbalance across cores. Conversely, DLB order queue and DLB atomic queue enabled almost evenly balanced load distribution across 4 cores, enhancing processing rates. Moreover, DLB atomic queue exhibited overall superior performance compared to DLB order queue in our experiments.

5 RELATED WORK

In multicore traffic scheduling, static assignment methods leveraging hardware support, such as RSS, employ hashing based on packet headers to distribute traffic to distinct cores but risk imbalances due to flow-to-core randomness. NIC-supported dispatching, exemplified by Intel's Ethernet Flow Director [26] and Mellanox's ASAP2 [23], utilize specialized "match-action" APIs to direct traffic to core-associated queues. However, these methods may inhibit dynamic balancing and increase deployment complexity due to fixed data division across cores.

Given that static allocation approaches neglect the real-time workloads of processing cores, various proposals are based on centralized scheduling concepts, such as Shenango [24] and Shinjuku [16], which satisfy tail latency SLOs [8] and increase CPU efficiency through rapid core scheduling and preemption. However, assigning specific cores for meticulous task management results in a trade-off in throughput and a deficiency in scalability relative to modern high-speed link bandwidths.

Recent works propose dynamic traffic reassignment for core load balancing, exemplified by Metron's [18] traffic-class based method that redistributes traffic upon core overloads, though it struggles with nondivisible traffic classes. RSS++ [5] enhances this by adaptively adjusting the RSS table for even load distribution, improving CPU utilization and adjusting core allocation based on traffic. Nonetheless, it cannot handle scenarios with large-volume flows that overload single cores. Dyssect [6] mitigates such issues by coordinating core interactions with a central unit, optimizing load balancing and traffic prioritization. However, allocating offload cores is impractical for data center vSwitches due to significant central controller resource overheads.

6 CONCLUSION

This paper presents a load balancing system dedicated to vSwitches. Through an exploration of the vSwitch datapath architecture and examination of imbalanced workload data in vSwitches, we characterize a range of inherent load imbalance patterns and identify four typical scenarios of load imbalances. Within the realm of current technologies, a universal remedy addressing the array of imbalance issues present within vSwitches remains elusive. To bridge this gap, we have developed vSwitchLB, consisting of both detection and mitigation components. The detection component is engineered for the real-time discernment of load imbalance types. The mitigation component is structured with a stratified approach, deploying methods at varying levels of traffic, tailored to the unique requirements of each identified load imbalance scenario.

ACKNOWLEDGMENTS

The work was partially supported by National Key R&D Program of China Zhejiang (2023R5202).

REFERENCES

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. 2022. Understanding host interconnect congestion. In *ACM HotNets 22*. 198–204.

[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *ACM SIGCOMM 23*. 275–287.

[3] APACHE. 2024. APACHE KAFKA. https://kafka.apache.org/

[4] T BARBETTE. 2018. DPDK extensions for OpenBox. https://github.com/tbarbette/fastclick/

[5] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *ACM CoNEXT 19*. 318–333.

[6] Fabrício B. Carvalho, Ronaldo A. Ferreira, Ítalo Cunha, Marcos A. M. Vieira, and Murali K. Ramanathan. 2022. Dyssect: Dynamic Scaling of Stateful Network Functions. In *IEEE INFOCOM 22*. 1529–1538.

[7] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. 2018. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX NSDI 18*. 373–387.

[8] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[9] Linux Foundation. 2018. Open vSwitch. https://www.openvswitch.org

[10] Linux Foundation. 2019. Data Plane Development Kit. https://www.dpdk.org

[11] Linux Foundation. 2019. PMD Automatic Load Balance. https://docs.openvswitch.org/en/latest/topics/dpdk/pmd/

[12] Linux Foundation. 2019. vHost Multiqueue. https://docs.openvswitch.org/en/latest/howto/dpdk/

[13] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When cloud storage meets RDMA. In *USENIX NSDI 21*. 519–533.

[14] Intel. 2023. Dynamic Load Balancer. https://www.intel.com/content/www/tw/zh/download/686372/intel-dynamic-load-balancer.html

[15] Intel. 2023. The forth Intel Xeon Scalable Processor. https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable.html

[16] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *USENIX NSDI 19*. 345–360.

[17] Georgios P Katsikas, Tom Barbette, Dejan Kostić, JR Gerald Q Maguire, and Rebecca Steinert. 2021. Metron: High-performance NFV service chaining even in the presence of blackboxes. *ACM Transactions on Computer Systems (TOCS)* 38, 1-2 (2021), 1–45.

[18] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *USENIX NSDI 18*. 171–186.

[19] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[20] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. 2011. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *USENIX NSDI 11*.

[21] Jianyuan Lu, Tian Pan, Shan He, Mao Miao, Guangzhe Zhou, Yining Qi, Shize Zhang, Enge Song, Xiaoqing Sun, Huaiyi Zhao, Biao Lyu, and Shunmin Zhu. 2024. CloudSentry: Two-Stage Heavy Hitter Detection for Cloud-Scale Gateway Overload Protection. *IEEE Transactions on Parallel and Distributed Systems* 35, 4 (2024), 616–633.

[22] Chengfei Lv, Chaoyue Niu, Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, et al. 2022. Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning. In *USENIX OSDI 22*. 249–265.

[23] Mellanox. 2017. Mellanox ASAP2: Accelerated Switching and Packet Processing. https://www.mellanox.com/related-docs/products/SB_asap2.pdf

[24] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI 19*. 361–378.

[25] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. 2021. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *ACM SIGCOMM 21*. 194–206.

[26] Clayne B. Robison. 2017. How to Set Up Intel Ethernet Flow Director. https://software.intel.com/enus/articles/setting-up-intel-ethernet-flow-director

[27] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *ACM APNet 19*. 71–77.

[28] Enge Song, Nianbing Yu, Tian Pan, Qiang Fu, Liang Xu, Xionglie Wei, Yisong Qiao, Jianyuan Lu, Yijian Dong, Mingxu Xie, et al. 2022. MIMIC: SmartNIC-aided Flow Backpressure for CPU Overloading Protection in Multi-Tenant Clouds. In *IEEE ICNP 22*. 1–11.

[29] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *USENIX OSDI 22*. 267–284.

[30] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, et al. 2023. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In *ACM SIGCOMM 23*. 769–782.