Revisiting Traffic Splitting for Software Switch in Datacenter

YEONHO YOO, Korea University, Seoul, Republic of Korea
GYEONGSIK YANG, Korea University, Seoul, Republic of Korea
CHANGYONG SHIN, Korea University, Seoul, Republic of Korea
HWIJU CHO, Korea University, Seoul, Republic of Korea
WONMI CHOI, Korea University, Seoul, Republic of Korea
ZHIXIONG NIU, Microsoft Research, Beijing, China
CHUCK YOO, Korea University, Seoul, Republic of Korea

Datacenter network topology contains multiple paths between server machines, with each path assigned a weight. Software switches perform traffic splitting, an essential networking operation in datacenters. Previous studies leveraged software switches to distribute network connections across paths, under the assumption that the software switches accurately divide connections according to path weights. However, our experiments reveal that current traffic splitting techniques exhibit significant inaccuracy and resource inefficiency. Consequently, real-world datacenter services (e.g., data mining and deep learning) experience communication completion times that are ~2.7× longer than the ideal. To address these problems, we propose VALO, a new traffic splitting technique for software switches, to accomplish two goals: high accuracy and resource-efficiency. For the goals, we introduce new concepts: score graph and VALO gravity. We implement VALO using the de-facto software switch, Open vSwitch, and evaluate it thoroughly. On average, VALO achieves 13.1× better accuracy and 25.4× better resource efficiency compared to existing techniques, with maximum improvements reaching up to 34.8× and 67.7×, respectively. As a result, VALO demonstrates 1.3×–2.5× faster average communication completion times for real-world datacenter services compared to existing techniques.

CCS Concepts: • **Networks** → **Data center networks**; **Cloud computing**; **Bridges and switches**; **Network performance modeling**.

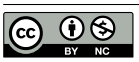
Additional Key Words and Phrases: Software switch, Traffic splitting, Datacenter, Cloud, Multipath routing

ACM Reference Format:
Yeonho Yoo, Gyeongsik Yang, Changyong Shin, Hwiju Cho, Wonmi Choi, Zhixiong Niu, and Chuck Yoo. 2025. Revisiting Traffic Splitting for Software Switch in Datacenter. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 2, Article 39 (June 2025), 26 pages. https://doi.org/10.1145/3727131

1 Introduction

The surge in demand for large-scale networking services, such as web search [19], data mining [63], and distributed training of deep learning models [84], has led to the utilization of datacenters (DCs) that provide multiple network paths between hosts, virtual machines (VMs), and containers. For example, Meta [57], Google [24], and Alibaba [66] implement multiple paths between VMs and containers to expedite their workloads, such as AI training. By leveraging multiple paths, these

Authors' Contact Information: Yeonho Yoo, Korea University, Seoul, Republic of Korea, yhyoo@os.korea.ac.kr; Gyeongsik Yang, Korea University, Seoul, Republic of Korea, g_yang@korea.ac.kr; Changyong Shin, Korea University, Seoul, Republic of Korea, cyshin@os.korea.ac.kr; Hwiju Cho, Korea University, Seoul, Republic of Korea, hjcho@os.korea.ac.kr; Wonmi Choi, Korea University, Seoul, Republic of Korea, ymcui@os.korea.ac.kr; Zhixiong Niu, Microsoft Research, Beijing, China, zhniu@microsoft.com; Chuck Yoo, Korea University, Seoul, Republic of Korea, chuckyoo@os.korea.ac.kr.



ACM 2476-1249/2025/6-ART39
https://doi.org/10.1145/3727131

services achieve high throughput, low latency, and high reliability in network communications [68, 79, 82]. Within the network topology of multiple paths, VMs and containers generate significantly different numbers of network connections (e.g., TCP or UDP) so that paths carry a wide range of network traffic depending on the number of connections.

So, across multiple paths, it is crucial to ensure the efficient and accurate distribution of network connections to provide performant networking services. In this context, "traffic splitting" is a technique used to distribute network connections across multiple paths based on assigned path weights. Path weights are the desired ratio of the number of connections assigned to specific paths relative to the total number of connections. These weights are determined by various factors in network topology, such as link capacity and congestion degree. For example, a path experiencing high congestion is assigned a lower weight to reduce the number of connections it handles. The goal of traffic splitting is to make the ratio of the number of connections per path close to the path weights [43, 83, 91]. Because network throughput and latency are largely determined by traffic splitting, the accuracy of traffic splitting plays a paramount role in the performance and reliability of cloud services [65, 67, 68].

In real-world systems, traffic splitting is frequently realized by software switches. For example, the DCs of Google [24] and Alibaba [49] run software switches at the physical servers that have hosts (VMs and containers), so that the packets from the containers or VMs go through the software switch. At the initial (ingress) point of the paths, the software switches determine a specific path to transmit the packets. OpenStack [70], a widely used VM orchestration software, also uses software switches for traffic splitting. Previous studies on software switches have proposed several techniques to determine path weights to work with traffic splitting to improve link utilization and enhance the quality of service [21, 35, 36, 45].

However, previous studies did not question the accuracy and overhead of traffic splitting from software switches. They assumed that 1) switches accurately distribute network connections across multiple paths according to path weights, and 2) switches do not introduce major bottlenecks during traffic splitting. Consequently, previous studies have focused primarily on determining path weights reflecting network status (e.g., the volume of network traffic, link aliveness, and switch queue congestion). However, our motivating experiments reveal unexpected challenges in traffic splitting: 1) considerable inaccuracies (§3.2) and 2) resource-inefficiencies (§3.3).

Our motivating experiments with real-world traces (CAIDA [4, 5] and ClassBench [54]) show that existing traffic splitting techniques demonstrate ~138% errors in traffic splitting compared to the expected behavior based on path weights. In addition, even for a single packet, a software switch can consume ~32K CPU cycles and induce ~19.8 μs latency for traffic splitting, which shows its gross inefficiency. These observations completely contradict the assumptions of previous studies. As a result, representative DC network workloads, web search, data mining, distributed training of deep learning models, and in-memory caching, can suffer ~2.7× longer flow completion time compared with an optimal traffic splitting scenario. Thus, it is quite clear that revisiting traffic splitting could benefit the performance of networking services in DCs.

To this end, we propose VALO, a new traffic splitting technique of software switches that satisfies 1) high accuracy and 2) resource-efficiency. To achieve these goals, we first analyze the mechanisms of existing traffic splitting techniques (§2.2) and pinpoint the challenges with our motivating experiments (§3). We then construct our own mathematical model, called "score graph." Given path weights, this score graph estimates the number of network connections assigned to each path, which we denote as "volume" (§4.1). We find that the ratio of measured connections per path significantly differs from the ratio of path weights, indicating the inaccuracy of existing traffic splitting techniques. But the ratio of the measured connections per path is close to the volume.

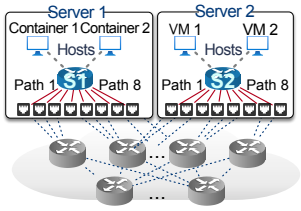


Fig. 1. Usage of software switches for traffic splitting.

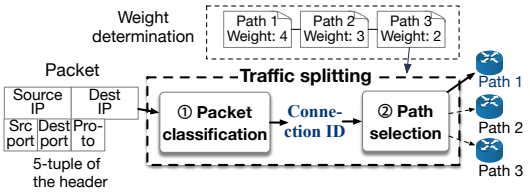


Fig. 2. Traffic splitting mechanism. The traffic splitting occurs at the switch where multiple paths are available, such as S1 in Fig. 1.

From the score graph, we formulate an accurate and resource-efficient technique called VALO. The key idea is to introduce a novel parameter called "VALO gravity" that makes the volume of each path align with the given path weight (§4.2). VALO gravity values are used to distribute network connections to multiple paths, ensuring adherence to given path weights.

We implement VALO using Open vSwitch (OVS) [64], the de-facto and most widely used software switch. With this implementation, we extensively evaluate VALO in terms of its accuracy and resource-efficiency using real-world Internet and DC traces. Furthermore, we conduct end-to-end evaluations of VALO with real-world DC workloads (e.g., web search, data mining, deep learning, and in-memory cache of Twitter). The major contributions of this study are as follows:

- Discover the critical challenges associated with the traffic splitting of software switches in DC networking: inaccuracy and resource-inefficiency.
- Design an accurate and efficient traffic splitting technique by score graph and VALO gravity.
- Achieve 13.1× (~34.8× at maximum) improved accuracy and 25.4× (~67.7× at maximum) enhanced resource-efficiency on average compared to existing traffic splitting techniques.
- Deliver 1.3×–2.5× superior end-to-end performance on average, and 1.4×–2.8× faster at the 99th-percentile tail, for DC workloads (e.g., deep learning and in-memory cache of Twitter).
- Demonstrate that VALO improves the flow completion time by 1.4× on average when it is applied to other multipath routing studies.

2 Background

This section explains the background of this study: the roles of software switches in DC (§2.1) and traffic splitting mechanisms (§2.2).

2.1 Software Switch

Usages. As communication frequently becomes a bottleneck for DC services (e.g., data mining and distributed deep learning [33, 34]), DCs and previous studies [29, 66, 76] equip each server with multiple network interfaces. For example, Meta and Alibaba equip each server with eight network interfaces [30, 66]. Each network interface is connected to a different link in the network topology, corresponding to distinct paths. Then, each server runs multiple hosts (e.g., containers and VMs) and a software switch performs traffic splitting to transmit packets to and from these hosts [58, 77]. In the example shown in Fig. 1, S1 in server 1 is a software switch that acts as an ingress switch. S1 determines one of the network interfaces and their corresponding paths (e.g., Path 8) to transmit packets between two hosts (Container 1 and VM 2). S2 is a software switch that works as egress to deliver packets to VM 2. When VM 2 sends packets to Container 1 in the opposite direction, S2 becomes the ingress switch that performs traffic splitting, and S1 becomes the egress switch. Note that the use of software switches for traffic splitting is prevalent, including Google [42], RedHat OpenStack [70], Alibaba [49], and LINE [3, 41].

Types. Various software switches have been proposed [89]. OVS [64], a popular open-source software switch, is the de facto standard. It is widely deployed in open-source DC solutions such as OpenStack [59] and Kubernetes [38]. Many papers also regard OVS as the software switch providing required networking functionalities for DCs (including traffic splitting) and have attempted to improve various aspects of it, such as scalability, feasibility, and security [22, 28, 69]. Considering its widespread use, this study uses OVS as the basis for building VALO.

VFP is another software switch that is currently used in Microsoft Azure by supporting various traffic management schemes, such as traffic limiting, packet filtering, and NAT [29]. Also, Hoverboard is a software switch used in Google's DCs to manage large-scale packet processing, including traffic splitting [24]. Apsara vSwitch forwards packets from VMs inside a host to an external network in Alibaba Cloud [49]. It allows customization of switch components (e.g., flow table) to achieve high performance tailored to each tenant. VALE is designed to operate at network link layer to provide high throughputs in packet delivery [71]. Snabb is a software switch for network function virtualization and bypasses the kernel networking stack to enhance networking throughput in packet delivery [61]. These software switches are widely used in DCs and necessitate accurate and efficient traffic splitting across multiple paths.

2.2 Traffic Splitting

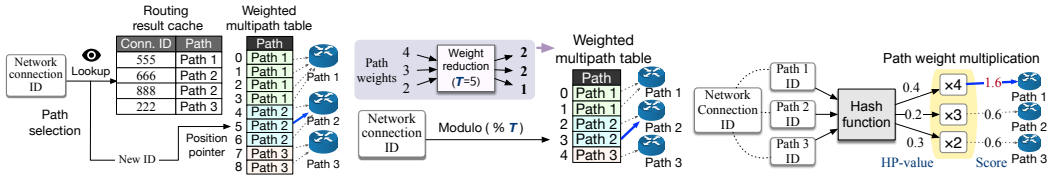
In this subsection, we explain the traffic splitting mechanism. Between source and destination hosts, DC network topology has multiple paths (set of links and switches) [45, 93]. For each network connection of multiple packets, traffic splitting selects which paths can transmit the network connection between two hosts and selects one of them.

Traffic splitting has three characteristics. First, it splits traffic into multiple paths based on path weights. The software switch is connected to multiple interfaces (as explained in §2.1), and each interface corresponds to a distinct path in the network topology. So, the software switch distributes network connections to its interfaces according to these weights. Second, all packets belonging to the same connection use the same path; otherwise, packets may arrive out of order, leading to poor throughput and reliability [91]. Third, traffic splitting is done per-packet basis because each switch receives and processes packets individually.

Fig. 2 shows the traffic splitting mechanism of a switch per packet. When every packet enters a switch, the switch first performs packet classification (① in Fig. 2) to identify the network connection to which the packet belongs. Then, it checks the paths (switch's attached interfaces) that can deliver the packet from the source to the destination hosts. Based on the identified network connection for the packet (connection ID), the switch performs path selection (②). We explain the packet classification and path selection in detail.

2.2.1 Packet classification. To get a network connection ID from an incoming packet, a hashing algorithm, such as CRC [1], is used. The 5-tuple values from the packet header (i.e., source IP address, destination IP address, source port number, destination port number, and transport protocol) are used as the key inputs for the hashing algorithm. The result of the hashing algorithm is a network connection identifier (ID), a unique value for each network connection. Also, pre-processing can be used before the hashing to decrease the length of key values. For example, by applying bitwise operations (e.g., XOR) to 5-tuple values, the key length can be decreased. However, it is known that this pre-processing step does not affect the performance of traffic splitting [91].

Hash functions are lightweight in identifying network connection IDs but can result in hash collision or polarization. This means that different network connections, each with distinct 5-tuple values, might produce identical IDs, which reduces the accuracy. But, this issue has been explored and addressed in other studies [43, 83, 91] by significantly reducing the hash collisions.



(a) Weighted round-robin (b) WCMP (c) Scoring

Fig. 3. Path selection techniques.

Our experiments show that hash collisions are controlled within 0.01% on real-world traces (details in §6). So hash collision is not the main concern in this study.

2.2.2 Path selection. With the network connection ID, traffic splitting selects a path for the packet. For the path selection (② in Fig. 2), existing studies and implementations are categorized into four techniques: 1) random, 2) weighted round-robin (WRR), 3) weighted cost multipathing (WCMP), and 4) scoring. Note that these four techniques are state-of-the-art and are used in both existing hardware and software switches [6, 7, 40, 93].

Random. Random technique [40] is one of the widely used techniques in many traffic splitting implementations, such as equal-cost multipath routing (ECMP) [37, 40], due to its simplicity. Specifically, random technique selects a path with a random distribution. So, a path is selected randomly. Generally, all packets of the connection go to the chosen path to avoid the out-of-order problem that reduces network throughputs of TCP [91]. Random technique typically uses either specific bit values from 5-tuple values as its random seed or a hashing to generate random values for choosing the path (e.g., path ID) [9]. The subsequent packets are assigned to the same path because the packets of the same network connection have identical 5-tuple values. This technique is relatively simple in path selection ($O(1)$ of time complexity). However, the random technique is unable to accurately maintain path weights, due to its inherent randomness.

WRR. WRR [6] is the representative technique in supporting path weights. Fig. 3a shows WRR technique that assigns network connections to each path sequentially while considering path weights. Specifically, given the network connection ID of a packet, WRR initially checks whether the connection ID has been processed before. It looks up a table called "routing result cache" that stores previously processed network connection IDs and their assigned paths. If it matches, WRR sends the packet to the stored path in the table.

If not, WRR determines the path for the network connection by the "weighted multipath table." This table has entries for every path, and the number of entries is proportional to weights. For example, in Fig. 3a, path weights for three paths (Path 1, 2, and 3) are 4:3:2, and the numbers of entries for paths are four, three, and two. The path weights of WRR should be integer values since the weights represent the number of entries, which is an integer. WRR also has "position pointer" that points out the path next to be assigned for the new network connection ID. So, WRR assigns the path that the position pointer indicates. The position pointer then advances to the next entry.

WRR needs to manage the routing result cache, weighted multipath table, and position pointer. It looks up tables, which results in $O(n)$ complexity when the number of table entries is n. In addition, whenever path weights change, WRR should flush the table entries, which generates additional latency. Furthermore, memory usage from the weighted multipath table increases with the size of the weight and available paths because WRR replicates table entries for the given path weights. When the sum of path weights gets large across multiple paths and servers in DC, this replication leads to significant memory consumption.

WCMP. WCMP [93] aims to reduce the high memory usage associated with WRR. The high memory consumption is because of creating multiple entries for each path in the weighted multipath table as explained above. In WRR, the number of entries for each path is equivalent to its path weight. So, to reduce the memory consumption of the table entries, WCMP sets the maximum number of table entries (T). Then, it modifies the given path weights to become equal to or less than T. This results in the total number of table entries being equal to or less than T. Similar to WRR, the path weights of WCMP are integer values.

Fig. 3b shows an example where T is 5 and initial weights are 4:3:2 for Path 1, 2, and 3. The weights are then proportionally reduced to 2:2:1 because T is 5. This modification of weights reduces the table size but distorts the original weights. Path 2 and 3 are given weights of 3:2, but the modification makes the weights into 2:1. This distortion can lead to inaccuracy.

Also, WCMP reduces the lookup overhead on the routing result cache of WRR by using a modulo operation. For each packet, network connection ID % T is calculated and simply used as an index on the weighted multipath table. So, WCMP finds the table entry of the weighted multipath table without looking up the routing result cache. In summary, WCMP reduces memory usage and lookup overhead, but modifying path weights and omitting position pointer can compromise accuracy.

Scoring. Scoring [7] tries to reduce the complexity of path selection. Fig. 3c shows scoring technique that uses an additional hash function for path selection [9]. The connection ID and path ID become the inputs for the hash function. For example, in Fig. 3c, the hash function receives a key (connection ID, path ID). Then, for each path, the hash function generates a random value, which we refer to as "HP-value" (0.4, 0.2, and 0.3 in Fig. 3c). Scoring then takes path weights into account by computing "scores" by multiplying each HP-value by its path weight (4, 3, and 2), yielding scores of 1.6, 0.6, and 0.6. The path with the highest score (Path 1 in Fig. 3c) gets selected. In summary, scoring chooses a path by a random value for a given connection ID but also considers path weights. Scoring requires no table structures and only needs a hashing algorithm and multiplications, thus exhibiting $O(1)$ complexity for a fixed number of paths. This is much less complex than WRR. Due to its reduced complexity, de facto software switches (e.g., OVS) utilize scoring [7].

2.2.3 Relationship between traffic splitting and weight determination. Traffic splitting determines the number of connections per path based on path weights. It works with weight determination (as shown in Fig. 2) that decides and updates path weights based on network congestion and failures. For example, Google [42] and Microsoft Azure [56] utilize SDN controllers to detect network congestion and determine the path weights for each switch according to the congestion. SDN controllers [2, 8] also provide weight determination functionalities by detecting network congestion and failures. Also, some studies [36, 45] update path weights by having software switches measure the round-trip time of packets (e.g., packet probing). As these switches can detect transmission delays, they update the weights without the need for external SDN controllers. These weight determination techniques provide the path weights used by traffic splitting.

Broadly, the use of traffic splitting and weight determination together to enable multiple paths for network communication is called multipath routing (or traffic load balancing). Many studies in this area assume that existing traffic splitting techniques operate accurately and efficiently, focusing primarily on optimizing weight updates [45, 85, 93]. However, in the next section, we reveal that traffic splitting itself poses significant challenges, making multipath routing far from ideal.

3 Motivating Experiments

This section presents the traffic splitting challenges with motivating experiments. We first explain our setup (§3.1) and then results (§3.2–§3.4). Lastly, we summarize the experiment results (§3.5).

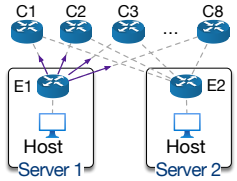
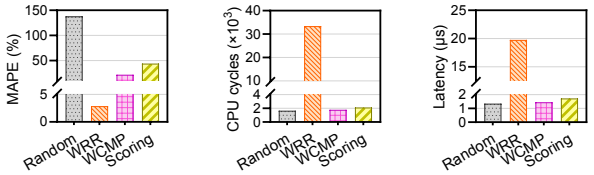


Fig. 4. Experiment topology.



(a) Accuracy (b) CPU usage (c) Latency

Fig. 5. Comparisons of traffic splitting.

3.1 Experiment Setup

Comparing techniques and metrics. Unless stated otherwise, we use the same experiment setup and methods throughout this paper. We compare four traffic splitting techniques with different path selection methods: random, WRR, WCMP, and scoring. The four techniques use the Jenkins hash function for the hashing of the packet classification step because it results in low hash collision rates (please refer to §6). We implement the random, WRR, and WCMP techniques into OVS [10]. Also, we utilize the scoring technique in OVS for experiments. We evaluate the following aspects:

- Accuracy: We present accuracy by measuring error rates from traffic splitting, which are discrepancies in the number of connections assigned to paths [43, 91]. Here, we measure the mean absolute percentage error (MAPE), a widely adopted metric for quantifying the discrepancy between observed and ideal performance [50, 53, 92]. For each trial, we calculate MAPE across all paths as Eq. (1). When a trial involves n paths, for the i-th path, we get the percentage error between 1) C_i, the number of connections that should be assigned based on path weight, and 2) $\hat{C}_i$, the actual number of connections assigned. The MAPE for each trial is the average percentage error of n paths. Note that MAPE can exceed 100% when $|C_i - \hat{C}_i|$ is greater than C_i, meaning the error is significantly large. We conduct 200 trials and present the average value from the trials.

$$MAPE = \frac{100}{n} \times \sum_{i=1}^{n} \frac{|C_i - \hat{C}_i|}{C_i} \tag{1}$$

- Resource-efficiency: We measure CPU cycle and latency of traffic splitting in software switches. Both metrics are measured per packet, and their average values from 200 trials are presented. The reason why we measure per packet is because traffic splitting occurs per packet (§2.2). Also, since the number of packets per connection is different, measuring per connection makes differences from packet counts rather than traffic splitting techniques.
- DC networking performance: As DC networking performance, we compare flow completion time (FCT) across four key DC workloads.

Topology and measurement methods. We build an experiment topology using Mininet [47] that emulates software switches, servers, and hosts using OVS and containers. The topology runs on a physical machine of Intel Xeon E5-2600 CPUs (24 cores) and 64 GB memory. The switches and hosts are linked via veth interface. Also, each link's bandwidth capacity is set to 1000 Mbps.

For accuracy and resource-efficiency experiments, we use a two-tier topology (Fig. 4) that is commonly used [11, 78]. We create two servers, each having one host (container). Each host is connected to an edge switch (E1 and E2 in Fig. 4), which is a software switch located on its server. We vary the number of paths between two servers from two to eight, which is a common deployment scenario for public cloud [30]. According to the number of paths, the number of core switches changes. The physical machine emulates all components of the topology, and each host and switch uses 2 CPU cores. Note that this configuration is used in the existing studies [73, 87].

We measure the performance at edge switches that perform traffic splitting. We instrument OVS in order to track 1) the number of connections assigned to each path and 2) the elapsed time that the switch takes to perform traffic splitting for each packet to measure accuracy and latency, respectively. Also, CPU usage is measured as the number of CPU cycles required for traffic splitting using `clock()` function. We report the average number of CPU cycles consumed per core.

For the DC networking experiments, we create 32 hosts to generate network connections for DC service workloads, as like existing studies [35, 45]. The CPU cores of the machine are evenly divided into the switches and hosts in order to make sure that CPU is not a bottleneck for the experiments [27, 47]. FCT values are measured from the 32 hosts by traffic generator [15].

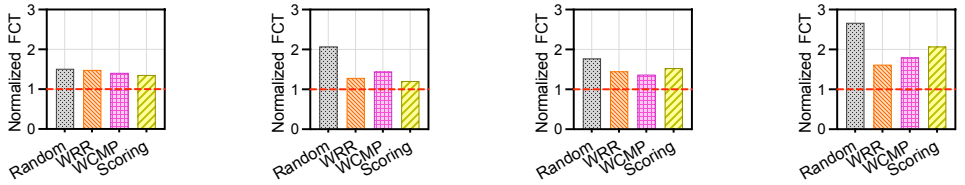
Experiment parameters. Path weights are parameters specified by external weight determination techniques, which consider the remaining bandwidth and congestion degree of each path [45, 46, 65]. We also observe that real-world DC traces show weights typically range from 1–100 [14, 63, 86], so in our experiments, we randomly select path weights from this range. Actually, we experiment with weight values beyond this range, and the results show a similar tendency. So, we omit them. Furthermore, we apply the selected weights identically to all traffic splitting techniques, ensuring a fair comparison. Similar to related studies [83, 93], bandwidth capacity of paths is set in proportion to weights. For instance, if path weights are 2:3, the bandwidth capacities of paths are set to 2:3.

We repeat experiment multiple trials to obtain reliable results. Specifically, the experiments for accuracy and resource efficiency are repeated for 200 trials. In each trial, we change path weights. DC networking performance experiments are conducted for 10 trials. On average, the accuracy and resource efficiency experiments take 11 s per trial, whereas the DC networking experiments 336 s per trial. So, the total running times are 37 min and 56 min, respectively. Note that the numbers of experiment trials are comparable or higher than previous studies that conducted 3–10 trials [45, 51, 75]. All graphs show average values.

Workloads. All experiments use real-world traces. For accuracy and resource-efficiency experiments, we use two traces: CAIDA and ClassBench. CAIDA provides anonymized traces from DCs and Internet [4, 5]. ClassBench generates 5-tuple traces based on IPv4 prefixes [54]. We randomly sample 20K network connections from each trace, so a total of 40K connections for experiments. Note that the two datasets are publicly available and are among the most widely used traces [55, 95]. We also increase the number of connections up to 120K in §5, which we believe provides a sufficient amount for measurement. For DC networking experiments, we generate network connections using four DC workload traces: web search [14], data mining [32], distributed training of deep learning models (deep learning) [84], and in-memory cache from Twitter service [86]. For deep learning, we use a traffic trace from distributed training with four GPU workers and four parameter servers. All network connections are generated using an open-source traffic generator tool [15].

3.2 Accuracy

Fig. 5a shows the results of accuracy experiments by MAPE (y-axis) per traffic splitting techniques (x-axis). Each bar presents an average value. The technique with the lowest error is WRR, at 2.9%. This is because WRR maintains the routing result cache and weighted multipath table for accuracy. On the other hand, random shows the highest error (worst accuracy, 138%), and the result implies that it cannot consider weights properly. Also, WCMP and scoring show comparably high errors, at 21.9% and 44.1% each, as they take path weights but distort the weights to reduce memory usage or computation complexity as explained in §2.2.2.



(a) Web search (b) Data mining (c) Deep learning (d) In-memory cache

Fig. 6. DC networking performance.

3.3 Resource-efficiency

Fig. 5b shows the average CPU usage measured by the total amount of CPU cycles required for traffic splitting per packet. Among four techniques, WRR consumes the most, followed by scoring, WCMP, and then random. Specifically, WRR exhibits CPU usage that is 20.1×, 18.6×, and 15.7× higher than random, WCMP, and scoring, respectively. Considering that software switches perform various functionalities (e.g., overlay networking, packet filtering, address translation, and in-band telemetry), WRR has a severe problem. Next, Fig. 5c presents the average latency for traffic splitting. Similar to CPU usage, WRR also requires the longest time for traffic splitting—19.8 μs. Because WRR keeps path weights in its structures and requires table lookups, it inevitably incurs high overheads. WRR takes 14.8×, 13.7×, and 11.5× longer than random, WCMP, and scoring.

3.4 Impact on DC Networking Performance

We measure FCT values of network connections for web search, data mining, deep learning, and in-memory cache from Twitter. To highlight the impact of the traffic splitting techniques, we define an ideal scenario where software switches perform perfectly accurate traffic splitting. Specifically, before running experiments, we assign network connections to paths based on pre-defined weights. This is possible because we can know from the DC traces the number of connections and their arrival time in advance. Then, during experiments, the switch references a lookup table that maps each connection ID to its assigned path. However, this ideal scenario is infeasible in real-world, where connections cannot be known in advance and link congestion and failures cannot be handled with this scenario [83, 91, 93].

We normalize the measured FCT values of traffic splitting techniques against the baseline and present the values in Fig. 6. So, the subfigures in Fig. 6 represent the results for four different DC workloads using four traffic splitting techniques. The increase in FCT is highest with random—2.1× on average, peaking at ~2.7× (in-memory cache). Scoring follows with an average increase of 1.5× and a peak of ~2.1× (in-memory cache). WCMP and WRR show increases of 1.5× and 1.5× on average, with peaks of ~1.8× and ~1.6×, respectively, for in-memory cache. The results clearly show that existing traffic splitting performance is quite suboptimal.

In addition, we find that workloads exhibit different tendencies over techniques. For example, in the web search workload (Fig. 6a), scoring is comparable to WRR, whereas in the in-memory cache (Fig. 6d), scoring is worse than WRR (by 28%). This is because each workload has a different amount of data to transfer: in-memory cache has a median data size of 200 KB, while web search has 80 MB. FCT is influenced by a few factors, such as traffic splitting, queueing delay, and data transmission delay [63, 81, 93]. For the web search workload, queueing delay and data transmission become dominant [80, 88]. So, traffic splitting techniques become comparable.

3.5 Summary

In summary, random technique is resource-efficient, but it cannot take weights into account, which makes it unsuitable for use in DCs. WRR shows the highest accuracy among the four, but it is significantly resource-inefficient due to its complex structures and operations. WCMP also shows better resource-efficiency than WRR, but it presents poor accuracy due to the distortion of weights. Scoring is resource-efficient but exhibits poor accuracy even though it derives scores in path selection by multiplying path weights by HP-values. Because all techniques have either accuracy or resource-efficiency issues, we present VALO by exploring the design space to improve the accuracy of the scoring technique to consider both accuracy and resource-efficiency.

4 VALO Design

We first analyze the existing scoring technique to identify reasons for inaccuracies (§4.1). Then, we introduce the VALO workflow, with its novel parameter, VALO gravity, aimed at enhancing accuracy. Also, we develop resource-efficient calculation method for VALO gravity (§4.2).

4.1 Scoring Analysis with Modeling

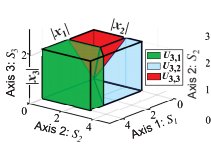
Score graph. Here, we introduce the concept of a "score graph," which counts the number of network connections on each path to identify reasons for inaccuracies in the existing scoring technique. When traffic splitting is performed over n paths (i.e., n number of next hops from the software switch), we define a score graph (I_n) to model and estimate the outstanding network connections as the set of "points." We use the following notations. For each path i ($1 \leq i \leq n$), a variable for path weight is represented as x_i. For j-th outstanding network connection, the score value for path i is denoted s_{ij}. Each axis i (e.g., axis 1, axis 2, and axis 3 in Fig. 7a) represents the values in S_i where $S_i = \bigcup_j s_{ij}$. Then, the point that represents the j-th network connection is defined as $(s_{1j}, s_{2j}, \cdots, s_{nj})$.

For example, the score graph of the j-th network connection over three paths (I_3) consists of the points represented by (s_{1j}, s_{2j}, s_{3j}) in Fig. 7a over axis 1, axis 2, and axis 3. Each axis represents the values in S_1, S_2, and S_3. Assume that k-th connection exists, and its HP-values are 0.4, 0.2, and 0.3 for three paths. When x_1, x_2, and x_3 are 4, 3, and 2, the s_{1k}, s_{2k}, and s_{3k} are 1.6, 0.6, and 0.6, respectively. Then, the connection is represented by a point (1.6, 0.6, 0.6) on I_3.

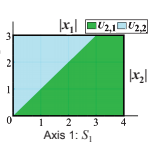
As another example, Fig. 7b shows the score graph for two paths (I_2). The points of the graph are defined as (s_{1j}, s_{2j}). The graph in Fig. 7b has two axes: axis 1 represents S_1 values of network connections and axis 2 for S_2 values. If a k-th network connection whose HP-values are 0.2 and 0.5 exists, and x_1 and x_2 are 4 and 3, s_{1k} and s_{2k} are 0.8 and 1.5. So, the connection is represented by the point (0.8, 1.5) on I_2.

In this way, I_n can represent scores of all outstanding network connections. To simplify the modeling, we assume that the range of HP-values on each axis is from 0 to 1. If the actual HP-value from hash functions exceeds 1 (e.g., 0–10), we scale it down using an appropriate ratio (e.g., by dividing the value by 10 for the case). Then, s_{ij} is calculated by multiplying x_i by HP-value. So, the range of S_i (each axis for path i) is from 0 to x_i.

Next, per network connection, scoring selects the path with the highest s_{ij} over all paths. In the two paths graph (Fig. 7b), suppose a point $(0.8, 1.5)$ in the blue-colored area (denoted as $U_{2,2}$ in Fig. 7b). From the point, s_{2j} (1.5) is higher than s_{1j} (0.8), so the network connection of the point is directed to path 2. Also, in the three paths graph (Fig. 7a), consider an arbitrary point (s_{1j}, s_{2j}, s_{3j}). The network connection of the point is directed to path 1 when s_{1j} is higher than s_{2j}, and also, s_{1j} is higher than s_{3j}. The point exists in the green-colored area (denoted as $U_{3,1}$ in Fig. 7a). By generalizing these examples, we define $U_{n,i}$ as the set of points on the graph I_n that are directed to

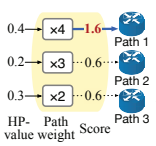


(a) Graph of three paths (I_3)

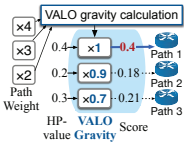


(b) Graph of two paths (I_2)

Fig. 7. Score graph examples.

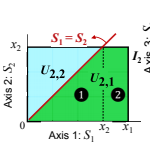


(a) Existing scoring

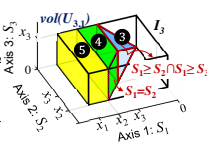


(b) VALO

Fig. 8. VALO workflow.



(a) Two paths example



(b) Three paths example

Fig. 9. $vol(U_{n,i})$ examples.

path i. The definition is as follows:

$$U_{n,i} = \bigcup_j \{s_{ij} \geq s_{1j} \bigcap s_{ij} \geq s_{2j} \bigcap \cdots \bigcap s_{ij} \geq s_{nj}\} = \bigcup_j \{\bigcap_{1 \leq m \leq n} \{s_{ij} \geq s_{mj}\}\} \tag{2}$$

The union of U_i on n paths is defined as the score graph (I_n).

$$I_n = \bigcup_{1 \leq i \leq n} U_{n,i} \tag{3}$$

Number of connections per path. Based on the defined $U_{n,i}$ and I_n, VALO estimates the number of points directed to each path. For path i, the number of points existing in $U_{n,i}$ is counted, as the points in $U_{n,i}$ are directed to path i. However, individually counting the points is challenging and severely inefficient due to the fact that 5-tuple values that include IP addresses and port numbers can theoretically allow for trillions of distinct network connections. So, instead of counting each point, we measure the "volume" of $U_{n,i}$, denoted as $vol(U_{n,i})$.

Values in S_i are uniformly distributed along each axis. This is valid for the following reasons. First, existing hash functions are designed to distribute hash values (HP-values) uniformly to avoid collisions between them [25]. So, because S_i is the multiplication of constants (weights × HP-values), it follows a uniform distribution. Note that this aligns with common practices when considering hash output [26]. We also test the representative hash functions and select the one whose output is most similar to the uniform distribution (see details in §6). So, instead of counting individual points, VALO calculates $vol(U_{n,i})$. Then, the number of network connections is proportional to $vol(U_{n,i})$.

Validation. We compare 1) path weights and 2) the ratio between $vol(U_{n,i})$. We experiment traffic splitting by running 20K network connections of the CAIDA dataset with the same experiment setup in §3.1. We test two cases: 1) for two paths of x_1 and x_2 are 4 and 3 (Fig. 7b) and 2) for three paths of x_1, x_2, and x_3 are 4, 3, and 2 (Fig. 7a).

In the case of two paths (I_2), the paths transmit 12634 and 7466 connections, which results in a ratio of 5:2.97. This quite differs from the assigned path weights, 4:3. We compute $vol(U_{2,1})$ and $vol(U_{2,2})$ through surface integral, obtaining a ratio of 5:3. The volume ratio is close to the actual ratio in which the network connections are divided. For the three paths (I_3), the experiment results show that the connections are divided into 5.05:2.81:1 ratio (11400, 6342, and 2258). This seriously deviates from the assigned path weights of 4:3:2. But the ratio of $vol(U_{3,1})$, $vol(U_{3,2})$, and $vol(U_{3,3})$ are 5.13:2.88:1 that closely aligns with the measured ratio of the connections. The above experiments show that although the scoring technique deviates the connection distribution from the path weight, the ratio of $vol(U_{n,i})$ conforms closely to the actual division ratio.

4.2 VALO Workflow and VALO Gravity

Based on our analysis, we introduce a new parameter—"VALO gravity"—to address the inaccuracy of scoring. VALO gravity makes $vol(U_{n,i})$ to align with the ratio of given path weights.

4.2.1 VALO workflow. Fig. 8 explains the VALO workflow and the role of VALO gravity in comparison with the scoring technique. When external network controllers or DC operators enter path

weight values to software switches, the existing scoring technique (Fig. 8a) ly uses them. In contrast, VALO calculates and utilizes VALO gravity values as new parameters to calculate scores (Fig. 8b). Whenever path weights are updated, VALO calculates the VALO gravity values for paths. VALO gravity values are determined to ensure that the ratio of volumes, $vol(U_{n,1}):vol(U_{n,2}):\cdots:vol(U_{n,n})$, is close to the ratio of path weights. Then, VALO calculates the score by multiplying the HP-value by the VALO gravity instead of the given weight. It then selects the path of the highest score.

4.2.2 Resource-efficient calculation of VALO gravity. We explain the calculation method of VALO gravity by $vol(U_{n,i})$ with weight variable, x_i. We can use surface and volume integrals to calculate $vol(U_{n,i})$. However, as the number of paths increases, the score graph increases the number of its axes and dimensions, and the calculation complexity becomes extremely high. So instead of the integral, we devise an efficient method to calculate $vol(U_{n,i})$.

***vol*$(U_{n,i})$ calculation.** Fig. 9a shows a score graph of two paths, I_2: axis 1 and axis 2 represent S_1 and S_2 values, respectively. Also, the range for axis 1 is 0–x_1, and for axis 2 is 0–x_2 as explained in §4.1. We assume that x_1 is greater than x_2, so the length of the side on axis 1 is longer in this example. The score graph is constructed by the union of $U_{2,1}$ and $U_{2,2}$. Specifically, $U_{2,1}$ represents $s_{1j} \geq s_{2j}$, and $U_{2,2}$ represents $s_{1j} \leq s_{2j}$. So, $U_{2,1}$ (marked green) and $U_{2,2}$ (marked blue) are divided by the line of $s_{1j} = s_{2j}$ (red line). Next, we calculate $vol(U_{2,i})$. The area of $U_{2,1}$ is subdivided into two: triangle (❶ in Fig. 9a) and rectangle (❷). So, $vol(U_{2,1})$ is calculated as Eq. (4a). Also, $vol(U_{2,2})$ (blue triangle) is calculated as Eq. (4b).

$$vol(U_{2,1}) = \frac{1}{2}x_2^2 + \left(x_1x_2 - x_2^2\right) \quad (4a) \qquad vol(U_{2,2}) = \frac{1}{2}x_2^2 \quad (4b)$$

Next, Fig. 9b, we consider I_3 with x_1, x_2, and x_3. I_3 is the union of $U_{3,1}$, $U_{3,2}$, and $U_{3,3}$. Axes 1, 2, and 3 represent S_1, S_2, and S_3 values, respectively. Also, the ranges for axes 1, 2, and 3 are 0–x_1, 0–x_2, and 0–x_3. Fig. 9b shows the $vol(U_{3,1})$ area by subdividing the area into three (❸–❺). The volumes of ❸–❺ are calculated in a manner similar to the I_2 example and result in Eq. (5a). Specifically, ❸ is one-third the volume of a cube with length x_3, ❹ is half the volume obtained by subtracting a cube with length x_3 from a cuboid with lengths x_2, x_2, x_3, and ❺ is the volume obtained by subtracting a cuboid with lengths x_2, x_2, x_3 from a cuboid with lengths x_1, x_2, x_3. Following the similar method, $vol(U_{3,2})$ and $vol(U_{3,3})$ are calculated as Eq. (5b) and Eq. (5c), respectively.

$$vol(U_{3,1}) = \frac{1}{3}x_3^3 + \frac{1}{2}\left(x_2^2x_3 - x_3^3\right) + \left(x_1x_2x_3 - x_2^2x_3\right) \quad (5a)$$

$$vol(U_{3,2}) = \frac{1}{3}x_3^3 + \frac{1}{2}\left(x_2^2x_3 - x_3^3\right) \quad (5b) \qquad vol(U_{3,3}) = \frac{1}{3}x_3^3 \quad (5c)$$

Through observation of the above examples, we notice regularities in $vol(U_{n,i})$. For example, when n paths exist, the first term of the equation is divided by $\frac{1}{n}$, the second term by $\frac{1}{n-1}$, and so on. So, we generalize $vol(U_{n,i})$ as follows:

$$vol(U_{n,i}) = \sum_{m=i}^{n} \frac{1}{m}\left(X_{n,m} - X_{n,m+1}\right), \quad X_{n,m} = \begin{cases} x_m^m\, x_{m+1} \ldots x_n & n > m \\ x_n^n & n = m \\ 0 & n < m \end{cases} \quad (6)$$

Proof of correctness. We prove the correctness of Eq. (6) by mathematical induction. When n=2 (base case),[1] $vol(U_{2,i})$ is $\sum_{m=i}^{2} \frac{1}{m}\left(X_{2,m} - X_{2,m+1}\right)$. Expanding this, $vol(U_{2,1})$ and $vol(U_{2,2})$ are identical to Eq. (4), confirming the correctness for $n = 2$. For the inductive hypothesis, assume that

[1]At least two paths are required for traffic splitting.

Eq. (6) holds for $n = k$ such that $vol(U_{k,i}) = \sum_{m=i}^{k} \frac{1}{m} \left(X_{k,m} - X_{k,m+1}\right)$. Next, we examine whether the equation holds for $n = k+1$. Eq. (6) for $n = k+1$ can be divided into two parts:

$$\begin{aligned} vol(U_{k+1,i}) &= \sum_{m=i}^{k+1} \frac{1}{m} \left(X_{k+1,m} - X_{k+1,m+1}\right) \\ &= \sum_{m=i}^{k} \frac{1}{m} \left(X_{k+1,m} - X_{k+1,m+1}\right) + \frac{1}{k+1} \left(X_{k+1,k+1} - X_{k+1,k+2}\right) \end{aligned} \tag{7}$$

For the first part, $\sum_{m=i}^{k} \frac{1}{m} \left(X_{k+1,m} - X_{k+1,m+1}\right)$, $X_{k+1,m}$ can be expressed as $X_{k+1,m} = x_m^m x_{m+1} \dots x_k$ $x_{k+1} = x_{k+1} \left(x_m^m x_{m+1} \dots x_k\right) = x_{k+1} X_{k,m}$. Similarly, $X_{k+1,m+1}$ can be expressed as 1) $x_{k+1} X_{k,m+1}$ for $m < k$ and 2) x_{k+1}^{k+1} for $m = k$. By expanding the equation, the first part becomes $x_{k+1} vol(U_{k,i}) - \frac{1}{k} x_{k+1}^{k+1}$. For the second part, $\frac{1}{k+1} \left(X_{k+1,k+1} - X_{k+1,k+2}\right)$, we note that $X_{k+1,k+2}$ is zero and $X_{k+1,k+1}$ is x_{k+1}^{k+1} according to Eq. (6). So, it simplifies to $\frac{1}{k+1} x_{k+1}^{k+1}$. Putting both parts together, $vol\left(U_{k+1,i}\right) = x_{k+1} vol\left(U_{k,i}\right) + \left(\frac{1}{k+1} - \frac{1}{k}\right) x_{k+1}^{k+1}$.

We show the correctness of the equation as follows. As defined in Eq. (2), $vol(U_{k+1,i})$ is the volume for the set of points in the score graph I_{k+1} for $k+1$ paths (axes), where the score for the i-th path (denoted as S_i) is the highest between paths, so leading to the selection of the i-th path. Note that, in the definition of the score graph, the range of S_i is 0–x_i (§4.1). For the calculation, the axes are indexed in the order $x_1 \geq x_2 \geq \dots \geq x_i \geq \dots \geq x_{k+1}$.[2]

Eq. (8) represents $vol(U_{k+1,i})$ calculated by integral over $(k+1)$-dimensional I_{k+1} by specifying the score range of each axis within the integral. Specifically, by definition in Eq. (2), points (connections) belonging to $vol(U_{k+1,i})$ can have S_i value from the range of 0–x_i for the i-th axis, which corresponds to $\int_0^{x_i} dS_i$ in Eq. (8). The other axes (e.g., $(i-1)$-th axis) have ranges of 0–S_i since S_i is the highest among the axes, as shown as $\int_0^{S_i} dS_{i-1}$. Note that from the $(i+1)$-th to $(k+1)$-th axes, upper integral limits are given by min, such as $\min(S_i, x_{i+1})$, because x_{i+1} can be smaller than S_i.

$$vol\left(U_{k+1,i}\right) = \int_0^{x_i} \int_0^{S_i} \cdots \int_0^{S_i} \int_0^{\min(S_i, x_{i+1})} \cdots \int_0^{\min(S_i, x_{k+1})} dS_{k+1} \cdots dS_{i+1} dS_{i-1} \cdots dS_1 dS_i \tag{8}$$

Similarly, $vol(U_{k,i})$ is as follows.

$$vol\left(U_{k,i}\right) = \int_0^{x_i} \int_0^{S_i} \cdots \int_0^{S_i} \int_0^{\min(S_i, x_{i+1})} \cdots \int_0^{\min(S_i, x_k)} dS_k \cdots dS_{i+1} dS_{i-1} \cdots dS_1 dS_i \tag{9}$$

Next, we expand $vol(U_{k+1,i})$ by dividing the range of S_i, 0–x_i, in Eq. (8) into two parts: 1) 0–x_{k+1} and 2) x_{k+1}–x_i. In the S_i range of 0–x_{k+1}, $\min(S_i, x_{i+1})$ is S_i. In the S_i range of x_{k+1}–x_i, $\min(S_i, x_{k+1})$ is x_{k+1}. The following equation demonstrates the expansion of $vol(U_{k+1,i})$:

$$\begin{aligned} vol\left(U_{k+1,i}\right) &= \underbrace{\int_0^{x_{k+1}} \int_0^{S_i} \cdots \int_0^{\min(S_i, x_{k+1})} dS_{k+1} \cdots dS_i}_{\text{(range } 0\text{–}x_{k+1})} + \underbrace{\int_{x_{k+1}}^{x_i} \int_0^{S_i} \cdots \int_0^{\min(S_i, x_{k+1})} dS_{k+1} \cdots dS_i}_{\text{(range } x_{k+1}\text{–}x_i)} \\ &= \int_0^{x_{k+1}} \left(S_i\right)^k dS_i + \int_{x_{k+1}}^{x_i} \int_0^{S_i} \cdots \int_0^{S_i} \underbrace{\int_0^{x_{k+1}} dS_{k+1}}_{= x_{k+1}} \cdots dS_i \\ &= \int_0^{x_{k+1}} S_i^k \, dS_i + x_{k+1} \int_{x_{k+1}}^{x_i} \int_0^{S_i} \cdots \int_0^{\min(S_i, x_k)} dS_k \cdots dS_i \\ &= \int_0^{x_{k+1}} S_i^k \, dS_i + x_{k+1} \left(\underbrace{\int_0^{x_i} \cdots dS_i}_{= vol(U_{k,i})} - \underbrace{\int_0^{x_{k+1}} \cdots dS_i}_{= \frac{1}{k} x_{k+1}^k} \right) \\ &= \left(\frac{1}{k+1} - \frac{1}{k}\right) x_{k+1}^{k+1} + x_{k+1} \, vol(U_{k,i}) \end{aligned} \tag{10}$$

That is, $vol(U_{k+1,i})$ is derived from $vol(U_{k,i})$ which concludes the mathematical induction.

[2]We just assign indices to the axes of I_{k+1} according to the values of x_i without altering any paths or connections.

VALO gravity calculation. Based on Eq. (6), we derive the method for calculating VALO gravity. Eq. (6) for I_n can be represented by matrix.

$$\begin{bmatrix} vol(U_{n,1}) \\ vol(U_{n,2}) \\ vol(U_{n,3}) \\ \vdots \\ vol(U_{n,n}) \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ & & \frac{1}{3} & \cdots & \frac{1}{n} \\ & & & \ddots & \vdots \\ & & & & \frac{1}{n} \end{bmatrix} \begin{bmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & & \ddots & \\ & & & 1 & -1 \\ & & & & 1 \end{bmatrix} \begin{bmatrix} X_{n,1} \\ X_{n,2} \\ X_{n,3} \\ \vdots \\ X_{n,n} \end{bmatrix} \tag{11}$$

$X_{n,i}$ values are calculated by multiplying the inverse matrix to Eq. (11), resulting in Eq. (12).

$$\begin{bmatrix} X_{n,1} \\ X_{n,2} \\ X_{n,3} \\ \vdots \\ X_{n,n} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ & 1 & 1 & \cdots & 1 \\ & & 1 & \cdots & 1 \\ & & & \ddots & \vdots \\ & & & & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & & & \\ & 2 & -2 & & \\ & & & \ddots & \\ & & & (n-1) & (-n+1) \\ & & & & n \end{bmatrix} \begin{bmatrix} vol(U_{n,1}) \\ vol(U_{n,2}) \\ vol(U_{n,3}) \\ \vdots \\ vol(U_{n,n}) \end{bmatrix} = \begin{bmatrix} vol(U_{n,1}) + vol(U_{n,2}) + vol(U_{n,3}) + \cdots + vol(U_{n,n}) \\ 2vol(U_{n,2}) + vol(U_{n,3}) + \cdots + vol(U_{n,n}) \\ 3vol(U_{n,3}) + \cdots + vol(U_{n,n}) \\ \vdots \\ nvol(U_{n,n}) \end{bmatrix} \tag{12}$$

From Eq. (12), we aim to obtain x_i. We find that by dividing x_i by x_1 (i.e., x_i/x_1), it can be simplified into a form that can be calculated solely by additions and multiplications. The example of x_2/x_1 is the following Eq. (13). We multiply both the numerator and the denominator of x_2/x_1 by $x_2 \times \cdots \times x_n$. Then, we can substitute x_i with $X_{n,i}$ and $vol(U_{n,i})$ using Eq. (6) and Eq. (12), respectively.

$$\frac{x_2}{x_1} = \frac{x_2^2 \times \cdots \times x_n}{x_1 \times x_2 \times \cdots \times x_n} = \frac{X_{n,2}}{X_{n,1}} = \frac{2vol(U_{n,2}) + \cdots + vol(U_{n,n})}{vol(U_{n,1}) + \cdots + vol(U_{n,n})} \tag{13}$$

We also calculate the ratios x_3/x_1 and x_4/x_1 similar to x_2/x_1 but omit the specific details due to the page limit. The results are Eq. (14). We generalize them into the ratio x_i/x_1 based on the patterns (mathematical induction) as Eq. (15). Note that Eq. (15) holds only when i is greater than 1. When i is 1, x_i/x_1 equals 1.

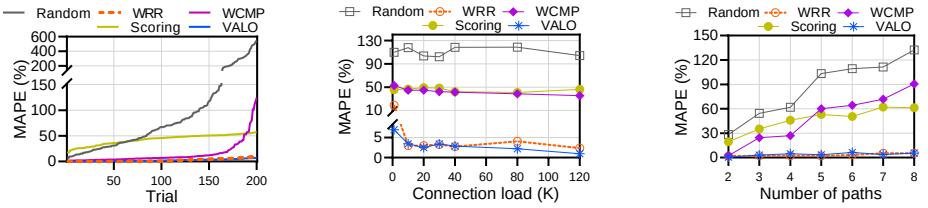
$$\begin{aligned} \frac{x_3}{x_1} &= \frac{X_{n,2}}{X_{n,1}} \times \left(\frac{X_{n,3}}{X_{n,2}}\right)^{\frac{1}{2}} = \frac{x_2}{x_1} \times \left(\frac{3vol(U_{n,3}) + \cdots + vol(U_{n,n})}{2vol(U_{n,2}) + \cdots + vol(U_{n,n})}\right)^{\frac{1}{2}} \\ \frac{x_4}{x_1} &= \frac{X_{n,2}}{X_{n,1}} \times \left(\frac{X_{n,3}}{X_{n,2}}\right)^{\frac{1}{2}} \times \left(\frac{X_{n,4}}{X_{n,3}}\right)^{\frac{1}{3}} = \frac{x_3}{x_1} \times \left(\frac{4vol(U_{n,4}) + \cdots + vol(U_{n,n})}{3vol(U_{n,3}) + \cdots + vol(U_{n,n})}\right)^{\frac{1}{3}} \\ &\vdots \end{aligned} \tag{14}$$

$$\frac{x_i}{x_1} = \prod_{k=2}^{i} \left(\frac{kvol(U_{n,k}) + vol(U_{n,k+1}) + \cdots + vol(U_{n,n})}{(k-1)vol(U_{n,k-1}) + vol(U_{n,k}) + \cdots + vol(U_{n,n})}\right)^{\frac{1}{k-1}} \tag{15}$$

In order for $vol(U_{n,i})$ to be aligned with the ratio of given weights (denoted as w_i), w_i is substituted into $vol(U_{n,i})$. Finally, with the substitution, x_i/x_1 becomes VALO gravity in Eq. (16).

$$\frac{x_i}{x_1} = \prod_{k=2}^{i} \left(\frac{kw_k + w_{k+1} + \cdots + w_n}{(k-1)w_{k-1} + w_k + \ldots + w_n}\right)^{\frac{1}{k-1}} \tag{16}$$

VALO then performs traffic splitting by using these VALO gravity values as its path weights. Note that VALO calculates VALO gravity using only Eq. (16) when the weights (w_i) are updated. We demonstrate its resource efficiency by presenting the scalability of VALO, such as CPU and memory usage, for increasing connection loads and paths in §5.2.



(a) Weight (b) Connection load (c) Path number

Fig. 10. Accuracy evaluation.

5 Evaluation

We implement VALO using OVS (version 2.9.8) on Linux kernel version 5.4.0. The VALO mechanism is integrated with the OVS routine that installs flow rules for traffic splitting. Our implementation includes VALO gravity calculation. We believe that VALO can be efficiently integrated with other software switches as a separate module. We release the VALO implementation in [10].

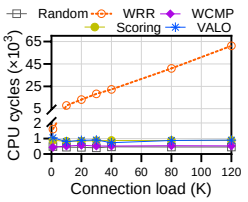
We compare five different traffic splitting techniques: random, WRR, WCMP, scoring, and VALO. The techniques are evaluated under a two-tier topology (Fig. 4). We evaluate VALO in four experiment sets: 1) micro-benchmarks (accuracy and resource-efficiency), 2) the overhead of weight fluctuations, 3) macro-benchmarks for end-to-end DC networking performance, and 4) the application of VALO on multipath routing. For micro-benchmarks, we measure accuracy and resource-efficiency. The experiments follow the setups and methods explained in §3.1. In addition to CPU cycles and latency, whose measurement methods are in §3.1, we also report the memory consumption of traffic splitting techniques. Memory consumption is measured by the GNU Time tool [31] that utilizes `wait4` system call to obtain memory usage. We vary three experiment parameters: weight, connection load, and path number as follows.

- **Weight**: We conduct 200 trials, each with randomly assigned path weights by selecting integer values from 1 to 100. The other parameters are fixed. For example, the connection load is fixed at 40K, and the number of paths is four.
- **Connection load**: The experiments are conducted with varying connection sizes ranging from 1K to 120K by sampling from CAIDA and ClassBench datasets. The number of paths is fixed at four, and the path weights are randomly assigned (integer values from 1 to 100).
- **Path number**: We vary the number of paths from two to eight. The connection load is fixed at 40K, and each path weight is randomly assigned from 1 to 100.

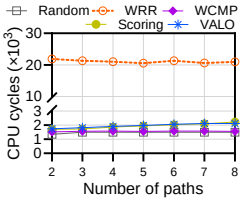
We also present the overhead and impact of path weight fluctuations by measuring CPU cycles and latency with similar methods as in our micro-benchmark. For macro-benchmarks, we report the average and 99th-percentile tail of FCT values for four DC workloads, as described in §3.1. Lastly, we demonstrate VALO's application in multipath routing, running VALO with weight determination techniques that consider network congestion. The number of experiment trials is similar to §3.1 (200 trials for micro-benchmarks and 10 trials for others), and average values are plotted.

5.1 Accuracy

Weight. Fig. 10a illustrates the errors in traffic splitting, as measured by MAPE (as in §3.2). The x-axis represents each experiment trial, with a total of 200 trials. In each trial, the accuracy is measured using the given path weights. The path weights differ between trials. We sort the trial results by their average MAPE values of the traffic splitting techniques, from smallest to largest. From the 200 trials, WRR and VALO outperform the other techniques, with each showing 2.5% and

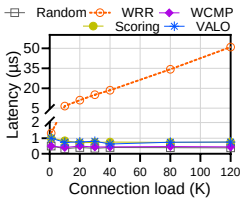


(a) Connection load

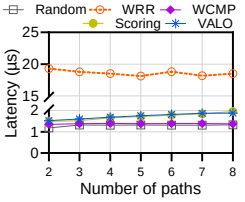


(b) Path number

Fig. 11. Resource-efficiency: CPU usage.

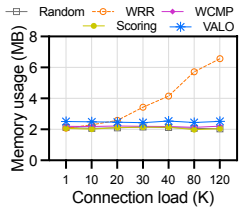


(a) Connection load

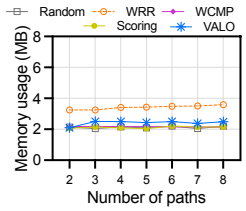


(b) Path number

Fig. 12. Resource-efficiency: latency.

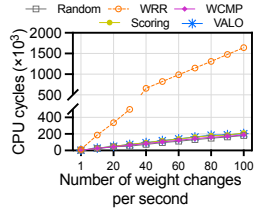


(a) Connection load

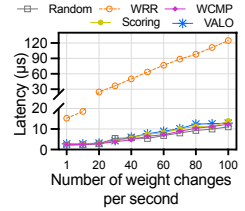


(b) Path number

Fig. 13. Resource-efficiency: memory usage.



(a) CPU usage



(b) Latency

Fig. 14. Impact of weight fluctuations.

2.3% average errors, and 10.5% and 7.3% maximum errors, respectively. Conversely, random, WCMP, and scoring demonstrate high errors, with averages of 104.2%, 14.1%, and 43%, and maximum errors of 546.5%, 124.4%, and 57%, respectively. Thus, VALO is, on average, 46.3×, 6.3×, and 19.1× more accurate than random, WCMP, and scoring, respectively.

Connection load. Fig. 10b shows the MAPE as the number of connections increases. Overall, the graph shows that VALO outperforms the other four techniques by 13.1× on average. Specifically, random shows the highest error (110.6% on average). WCMP (42.9%) and scoring (45.5%) exhibit the next highest errors, while the errors are fairly low with WRR (5.3%) and VALO (3.2%). As the connections increase, the MAPE values of the five techniques do not change much. These results show that VALO outperforms random, WCMP, and scoring (34.8×, 13.5×, and 14.3×, each) and effectively maintains its accuracy regardless of the number of connections.

Path number. Fig. 10c shows the MAPE as the number of paths increases. In the graph, the error of VALO is, on average, 9.4× better than those of the other techniques. Specifically, the error for random, WCMP, and scoring increases: random rises from 28.4% to 132.3%, WCMP from 2.4% to 90.6%, and scoring from 19.6% to 62.1%. Notably, WCMP presents a significant increase in error rate from two paths to eight paths (38.1×). This is because the distortion of path weights that reduces the accuracy in WCMP impacts significantly when the sum of path weights increases (§2.2.2). On the other hand, WRR and VALO exhibit relatively stable and very low errors, averaging 3.2% and 4%, respectively. Compared to random, WCMP, and scoring, VALO reduces the errors by 21.5×, 11.8×, and 11.8× on average, respectively.

5.2 Resource-efficiency

We evaluate the resource-efficiency by measuring CPU usage, latency, and memory usage of traffic splitting by varying three parameters that change in weight, connection load, and path number. Although we conduct experiments for all parameters, due to the page limit, we omit the results

of the path weight parameter because we observe that resource-efficiency results related to path weight show a similar tendency to those of path number changes.

CPU usage. Fig. 11a shows the CPU cycles of traffic splitting techniques as the number of connections increases. VALO exhibits a similar scale of latencies to random, WCMP, and scoring (all under 1000 CPU cycles). In contrast, WRR shows significantly higher CPU usage compared to the other techniques. Specifically, WRR's average CPU cycle is 26.8× higher than VALO's. In addition, as the number of connections increases, the CPU cycles of random, WCMP, scoring, and VALO remain stable (increasing by an average of 1.3×). On the other hand, the CPU cycles of WRR increase by 38.1×. Compared to VALO, WRR's increase is 26.7× greater.

Fig. 11b shows CPU cycles as the number of paths increases from two to eight. According to the results, all five techniques maintain consistent CPU usage over the path numbers (x-axis). WRR consumes the highest amount of CPU usage (21095 cycles on average). Random, WCMP, scoring, and VALO consume much less CPU than WRR—1478, 1561, 1941, and 1965 cycles on average. The average CPU usage of VALO is 10.7× improvement than WRR.

WRR performs table lookups and maintains complex structures to obtain accuracy (§2.2.2). So, WRR demonstrates its great accuracy compared to other traffic splitting techniques (as in Fig. 10). However, this comes at the expense of CPU resource efficiency that is significantly poor for WRR. On the contrary, the results show that our approach, VALO, significantly reduces CPU consumption and at the same time maintains accuracy similar to that of WRR (as demonstrated in Fig. 10).

Latency. Fig. 12a and Fig. 12b present the traffic splitting latency as the connection load and the number of paths increase. We observe similar tendencies in latency results as we see in the CPU usage results (Fig. 11). When the connection load increases (Fig. 12a), VALO achieves similar latencies to random, WCMP, and scoring (all under 1 μs). However, WRR shows the highest and increasing latency (~50.9 μs at 120K connections). VALO's latency is 25.4× better than WRR on average, and 67.7× better at maximum (120K connections).

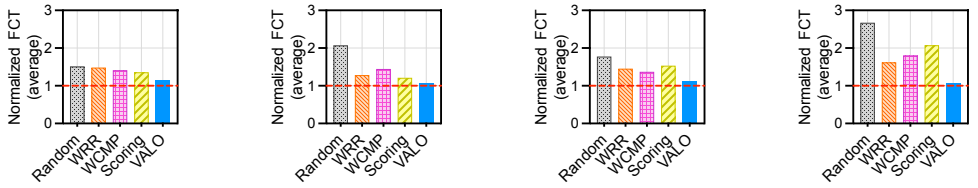
In addition, as the number of paths increases from two to eight (Fig. 12b), WRR exhibits the highest latency of 18.6 μs on average. In comparison, random, WCMP, scoring, and VALO show relatively similar latency levels (1.5 μs on average), which is 12.1× improvement from WRR. Similar to CPU usage results, VALO significantly improves the latency, which validates its effectiveness in both accuracy and resource-efficiency.

Memory usage. Fig. 13a and Fig. 13b show the memory usage of the traffic splitting techniques as the connection load and the number of paths increase. When the connection load increases (Fig. 13a), we observe that random, WCMP, scoring, and VALO exhibit consistent memory usage across all connection loads, averaging 2.2 MB. In contrast, WRR shows a significant increase in memory consumption: it increases 3.2× as the connection load increases from 1K to 120K. Compared to VALO, WRR's memory usage is 1.5× higher on average and 2.6× higher at peak (120K connections).

Next, as the number of paths increases (Fig. 13b), all techniques show stable memory usage. WRR shows the highest memory usage, averaging 3.4 MB, which is 41.4% higher than VALO. The others show relatively similar memory usage of 2.2 MB on average.

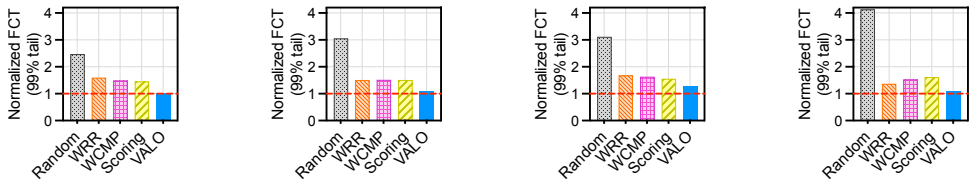
5.3 Overhead of Weight Fluctuations

We now evaluate the impact and overhead of traffic splitting techniques when path weights change. Instead of using the given path weights for each experiment trial as the experiments before, here, we change the path weights between 1 and 100 times per second, covering a range from slow to frequent changes. The path weights are given random integer values from 1 to 100. The number of paths is four, and the connection load is 10K. We measure both CPU usage and latency. CPU usage is calculated as the average number of CPU cycles consumed per second per core. Latency is measured as the average time for traffic splitting per packet during the path weight fluctuations.



(a) Web search (b) Data mining (c) Deep learning (d) In-memory cache

Fig. 15. Average FCT with end-to-end DC workloads.



(a) Web search (b) Data mining (c) Deep learning (d) In-memory cache

Fig. 16. 99th-percentile tail FCT with end-to-end DC workloads.

Fig. 14a and Fig. 14b show the CPU usage and latency measurements. First, in terms of CPU usage (Fig. 14a), WRR consumes the most CPU cycles of 826.2K on average. In contrast, VALO uses an average of 113.9K CPU cycles, which is 7.3× lower than WRR. In addition, on average, VALO consumes only 0.4% (scoring)–17.9% (random) more CPU cycles, which is not significantly high considering the improved accuracy of VALO.

Second, Fig. 14b presents the latency of five traffic splitting techniques. On average, VALO achieves latency of 7.6 μs, which is 8.45× lower than WRR (64.51 μs). In comparison, scoring achieves an average latency of 7.3 μs, making VALO's latency only 4.1% higher despite its computational overhead. This is because VALO requires only simple arithmetic calculations (Eq. 16).

5.4 Flow Completion Time of DC Workloads

We present the average and tail-end (99th-percentile) FCT values. First, Fig. 15 shows the average FCT for four DC workloads: web search (Fig. 15a), data mining (Fig. 15b), deep learning (Fig. 15c), and in-memory cache from Twitter (Fig. 15d). The measured FCT values are normalized to the baseline values in §3.4. So, a normalized FCT value close to 1 on the y-axis signifies that the traffic splitting technique (on the x-axis) achieves performance comparable to the ideal case.

In the results, the four existing techniques exhibit severely poorer FCT values than the baseline. Across the four DC workloads, random, WRR, WCMP, and scoring show FCT results that are higher than the baseline by factors of 1.5×–2.7×, 1.3×–1.6×, 1.4×–1.8×, and 1.2×–2.1×, respectively. On average, the increases in FCT for the four techniques are 1.8×, 1.3×, 1.4×, and 1.4×, respectively, across all workloads. On the contrary, VALO outperforms the other techniques across all DC workloads. VALO reduces the FCT by ~1.3×, ~1.9×, ~1.6×, and ~2.5× in web search, data mining, deep learning, and in-memory cache, respectively. In addition, the FCT of VALO differs from the baseline by only 6.5% (in-memory cache) to 14.9% (web search).

Next, Fig. 16 shows the 99th-percentile FCT values. We observe that the tail values follow similar trends to the average values in Fig. 15 across four workloads. Compared to VALO, the tail FCT values of random, WRR, WCMP, and scoring are 2.8×, 1.4×, 1.4×, and 1.4× higher on average. Also, the FCT of VALO differs from the baseline by 13% at the tail (on average for workloads).

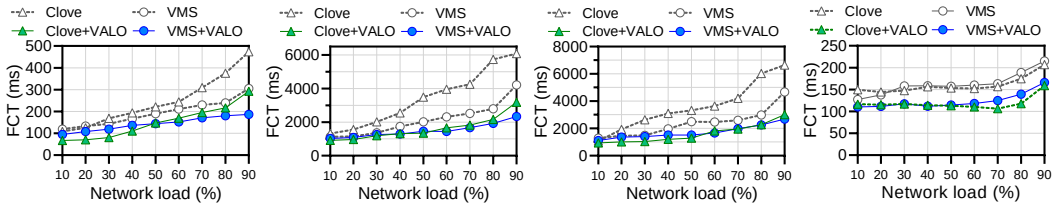


Fig. 17. Effectiveness of VALO on multipath routing.

These results show that the effectiveness of VALO in accuracy and resource-efficiency enhances the quality of major DC networking services on a macro scale as well.

5.5 VALO on Multipath Routing

Here, we present the effectiveness of VALO in multipath routing, which includes weight determination that updates weights based on network congestion or failures. We test two representative schemes for software switches: VMS [90] and Clove [45]. VMS monitors the congestion levels of paths and updates the path weights accordingly. It uses the random technique for traffic splitting. Specifically, for each packet, VMS randomly selects two paths and transmits the packet to the path with the higher weight. Similarly, Clove updates path weights but uses the WRR technique for traffic splitting. In addition to weight determination and traffic splitting, Clove introduces a packet grouping design. Instead of assigning all packets of a single connection to one path, Clove divides a single connection into multiple packet groups, termed flowlets, and assigns a path to each flowlet, making more fine-grained splitting. Specifically, Clove groups packets of a network connection based on packet arrival times. So, a new flowlet is created when the time gap between two consecutive packets exceeds a threshold (e.g., 150 μs [75]).

For VMS and Clove, we replace their traffic splitting techniques (random and WRR, respectively) with VALO. Four combinations are compared: VMS, Clove, VMS+VALO, and Clove+VALO. We implement the four combinations using OVS and evaluate using the same experiment settings in §5.4. We measure the FCT of four DC workloads, web search, data mining, deep learning, and in-memory cache (Twitter). We vary the network load (amount of network connections) with 10% to 90% of the network topology capacity, which is the identical configuration from the multipath routing studies [13, 15, 45, 90]. As the traffic is generated following the traces, network link utilization and congestion vary. The path weights are adjusted according to the weight determination techniques proposed in the VMS and Clove papers during each experiment trial.

Fig. 17 shows the FCT of four DC workloads. Across all workloads, VALO significantly improves FCT values of VMS and Clove. First, for VMS, VMS+VALO reduces the FCT of VMS by 23.9% (web search), 24.7% (data mining), 23.4% (deep learning), and 23.8% (in-memory cache) on average. As network load increases, the FCT of VMS increases by ~3.7× (Fig. 17b), while for VMS+VALO, it increases by ~2.4× (Fig. 17c). On average, VALO reduces the FCT increase in VMS by 1.5×. Next, for Clove, Clove+VALO decreases the FCT of Clove by 39.9%, 49.5%, 51.5%, and 25.7% on average in web search, data mining, deep learning, and in-memory cache. The FCT values for Clove and Clove+VALO increase by ~6.2× (Fig. 17c) and ~4.4× (Fig. 17a). On average, Clove+VALO reduces the FCT increase by 1.4×. These results indicate that VALO successfully enhances the multipath routing schemes in FCT performance.

6 Discussion

Hash functions of VALO. VALO uses two hash functions for packet classification and path selection. For packet classification, a hash function is used to obtain a connection ID from each packet. To decide which hash function to use, we test five representative hash functions—CRC, XOR, Pearson, Jenkins, and Murmur—on the CAIDA and ClassBench datasets used in §5. From the five, Jenkins causes only 0.01% hash collisions (generating the same connection IDs for different connection packets), while the others generate collisions of ~98.7%. So, we choose Jenkins.

For path selection, another hash function is used to generate HP-values (§4.1). We design VALO with the assumption that the HP-values would follow a uniform distribution. Thus, we also test the above five algorithms on the CAIDA and ClassBench datasets to check whether the HP-values generated by the hash function follow a uniform distribution. We conduct a chi-square goodness-of-fit test [39] that measures the similarity between the ideal uniform distribution and the generated values of the hash function. Among the five functions, Murmur shows the best similarity, while the others show ~79.9× poor similarities. So, we use Murmur for path selection.

VALO gravity calculation. VALO recalculates its gravity values only when path weights are updated, because gravity only depends on these weights (see Eq. (16)). Otherwise, VALO gravity remains unchanged if the path weights are unchanged. One might consider updating gravity values when new connections arrive or existing connections end, as this may alter the number of active connections and potentially change link capacity and congestion. Such changes are handled by weight determination techniques [13, 35], and it is orthogonal to VALO. VALO can work with the results from the weight determination techniques.

Accuracy under small network connections. In Fig. 10b, VALO demonstrates the high accuracy on network connection numbers of 1K to 120K for data centers [12, 23]. Here, we further evaluate VALO's accuracy for smaller network loads (e.g., 100 network connections in IoT sensor network [20, 52]). We vary the number of network connections from 10 to 100 under the same experiment settings as in §5.1. The additional results show that VALO presents 2.5× lower prediction errors (MAPE) on average for small connections than other techniques (random, WRR, WCMP, and scoring). The results demonstrate that VALO is still accurate for even small network loads.

Traffic splitting at different locations. This study focuses on traffic splitting from software switches across multiple paths of the networking fabric. However, traffic splitting can also occur in other locations. For instance, hardware switches within the optical networking fabric can perform traffic splitting [65], and NVLink can manage traffic splitting for data communication between GPUs within a single server. In addition, within the VM kernel, MPTCP [82] splits a network connection into multiple sub-connections, assigning them to different network interfaces. We observe that these different locations and environments for traffic splitting present unique challenges, which merit separate studies [48]. Given that this study's scope is focused on software switch-based traffic splitting, which is prevalent in DCs (e.g., Meta [57] and Google [42]) and industries (e.g., RedHat OpenStack [70] and LINE [3, 41]), we believe the challenges of software switch-based traffic splitting are significant and warrant a separate study, to which VALO contributes. We plan to extend VALO to other locations in future research.

Traffic splitting in different contexts. First, Ananta [62] presents Microsoft's cloud-scale load balancing that includes two distinct designs. The first design is traffic splitting, and Ananta uses the ECMP that is already compared with VALO under the name "random." The second design focuses on server load balancing, which is different from traffic splitting. Server load balancing distributes multiple user requests across multiple servers or applications, such as distributing web server requests to multiple backend servers. Ananta employs weighted random algorithm [94] for server load balancing, which randomly selects a server based on weights. To our knowledge, the

weighted random has not been applied to traffic splitting at all. To see its potential, however, we newly implement the weighted random specifically for traffic splitting. Under the same experiment settings with §5.1, we run the weighted random. The experiment results show that the weighted random has the average MAPE of 22.4%, which is 9.8× less accurate than VALO in Fig. 15a. Also, in terms of FCT for a web-search workload, the weighted random shows 9.6% longer FCT than VALO on average.

Second, Niagara [44] introduces a traffic splitting algorithm designed for an SDN controller, which pre-determines paths for potential network connections and installs flow rules on switches before the connections are established. While VALO and many other traffic splitting techniques (e.g., WRR and WCMP) assume a reactive routing—where individual switches select paths upon the arrival of new connections—Niagara takes a proactive approach at the controller level. In terms of selection algorithm, Niagara selects paths in a similar manner to the WRR technique described in §2.2.2. It uses a binary tree data structure for path selection, sequentially mapping all possible network connections to paths based on weights—similar to the weighted multipath table used in WRR (Fig. 3a). To support this structure, Niagara modifies the weights so that their sum equals the number of leaf nodes in the binary tree (i.e., power of two). This weight modification is in line with WCMP, and we compare WCMP with VALO in §5.1. Our results show that VALO achieves 6.3× higher accuracy than WCMP.

Third, weighted fair queueing (WFQ) [18] is a widely known packet scheduling algorithm that guarantees each queue of a link receives its fair share of the bandwidth according to its weight. It means that higher priority is given to queues that have transmitted fewer packets relative to their weights. However, to our knowledge, WFQ algorithm has not been used for traffic splitting. To explore its potential for traffic splitting, we newly design and implement WFQ on traffic splitting as follows. Instead of selecting a queue, we make WFQ select a path for each packet while retaining its weighted selection algorithm. In addition, to make the packets of the same connection be sent through an identical path, which is the mandatory condition of traffic splitting to avoid out-of-order packets (explained in §2.2), we use the routing result cache structure of WRR technique that stores the path decision for the first packet of each connection (§2.2.2). The WFQ for traffic splitting shows 16% worse prediction errors (MAPE) than VALO on average. Also, for the web-search workload, WFQ shows 23% longer FCT than VALO on average. In summary, VALO consistently outperforms even the algorithms originally designed for different contexts.

Hardware switch applicability. One might wonder whether VALO can be extended to hardware switches. Since existing hardware switches also use random and WRR techniques [6, 40], they are likely to encounter similar inaccuracy and resource inefficiency challenges. We believe implementing VALO in hardware switches is feasible because improving accuracy through VALO just requires multiplication operations in gravity calculations. Also, programmable switches that allow custom program codes to run on hardware (e.g., P4 [17]) are alternatives to hardware switches [16]. We plan to extend VALO to programmable switches in the near future.

7 Related Work

Hashing polarization in packet classification. Before path selection, a hash function classifies packets using their 5-tuple headers and obtains a network connection ID (§2). Different 5-tuple information, however, could produce the same network connection ID, a scenario known as hash polarization or collision. Various studies tried to avoid this limitation. For example, RePaC [91] leveraged linear properties in hash algorithms (e.g., CRC) to mitigate polarization. CLEO [43] used a machine learning approach to decrease hash collisions. Y Xu et al. [83] presented a hash reuse scheme. VALO is orthogonal to these studies and can, therefore, work together with them.

Software switch improvements. As software switches are widely used [74], previous studies improved various aspects [89]. For example, NuevoMatchUP [69] improved scalability in packet matching with OpenFlow rules. MFCGuard [22] improved the security and robustness of software switches by monitoring the flow rule masks. PISCES [72] extended the software switch functionality to allow for custom programmability in packet processing. NIKSS utilized eBPF to improve the network performance of customizable packet processing logic (e.g., P4) on switches [60]. However, to our knowledge, no studies enhanced the traffic splitting of software switches as VALO does.

8 Conclusion

Traffic splitting in software switches is essential in today's DC networking. However, we report that existing techniques face severe challenges: inaccuracy and resource-inefficiency. In this study, we introduce a novel approach, VALO, which incorporates score graph and VALO gravity into traffic splitting. Through the full implementation based on OVS, VALO achieves ~34.8× and ~67.7× improvements in accuracy and resource-efficiency, respectively, under real-world traces. Furthermore, VALO shows remarkable improvements (~2.8×) in tail FCT for four key real-world DC workloads. We hope VALO can boost a diverse range of DC networking applications.

Acknowledgments

We thank our shepherd, Florin Ciucu, and the anonymous reviewers for their insightful comments that helped us to improve this study. We also acknowledge the early input from Hoseok Kim and Junseok Lee. This research was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (RS-2021-NR060143), the NRF grant funded by the Korea government (MSIT) (RS-2024-00336564, RS-2023-NR077249), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by MSIT (RS-2024-00405128), ICT Creative Consilience Program by IITP grant funded by MSIT (IITP-2025-RS-2020-II201819), Google Cloud Research Credits, and a Korea University Grant. The corresponding authors are Gyeongsik Yang and Chuck Yoo.

References

[1] 2012. Data Center Access Design with Cisco Nexus 5000 Series Switches and 2000 Series Fabric Extenders and Virtual PortChannels. (2012). https://itnetworkingpros.files.wordpress.com/2014/04/c07-572829-01_design_n5k_n2k_vpc_dg.pdf [Accessed: Jan. 9, 2024].

[2] 2017. Open Network Operating System (ONOS) SDN Controller for SDN/NFV Solutions. (July 2017). https://opennetworking.org/onos/ [Accessed on 10/3/2024].

[3] 2018. Excitingly simple multi-path OpenStack networking: LAG-less, L2-less, yet fully redundant — openstack.org. (2018). https://www.openstack.org/summit/vancouver-2018/summit-schedule/events/21113/excitingly-simple-multi-path-openstack-networking-lag-less-l2-less-yet-fully-redundant, [Accessed: Jun. 19, 2024].

[4] 2018. The CAIDA UCSD Anonymized Internet Traces - 2018. (2018). https://www.caida.org/catalog/datasets/passive_dataset [Accessed: Jun. 1, 2023].

[5] 2019. The CAIDA UCSD Anonymized Internet Traces - 2019. (2019). https://www.caida.org/catalog/datasets/passive_dataset [Accessed: Jun. 1, 2023].

[6] 2021. VMware Multipathing policies in ESXi/ESX. (2021). https://kb.vmware.com/s/article/1011340 [Accessed: Aug. 23, 2023].

[7] 2023. Production Quality, Multilayer Open Virtual Switch. (2023). https://www.openvswitch.org/ [Accessed: Jul. 15, 2023].

[8] 2024. OpenDayLight: Automating networks of any size & scale. (2024). https://www.opendaylight.org/ [Accessed on 10/3/2024].

[9] 2024. Rendezvous hashing. (2024). https://en.wikipedia.org/wiki/Rendezvous_hashing [Accessed: Aug. 30, 2023].

[10] 2025. VALO source-code repository. (2025). https://github.com/yeonhooy/VALO-OVS-SIGMETRICS25.

[11] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2024. Harmony: A Congestion-free Datacenter Architecture. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24).*

329–343.
[12] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. 2010. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, Vol. 10. San Jose, USA, 89–92.
[13] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *2014 ACM SIGCOMM*. 503–514.
[14] Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. DCTCP: Efficient packet transport for the commoditized data center. (2010).
[15] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 537–549.
[16] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. 2023. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1469–1487.
[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[18] Wei Chen, Ye Tian, Xin Yu, Bowen Zheng, and Xinming Zhang. 2024. Enhancing Fairness for Approximate Weighted Fair Queueing With a Single Queue. *IEEE/ACM Transactions on Networking* (2024).
[19] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. 2013. A provider-side view of web search response time. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 243–254.
[20] Wonmi Choi, Yeonho Yoo, Kyungwoon Lee, Zhixiong Niu, Peng Cheng, Yongqiang Xiong, Gyeongsik Yang, and Chuck Yoo. 2024. Intelligent Packet Processing for Performant Containers in IoT. *IEEE Internet of Things Journal* (2024).
[21] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. 2016. Virtualized congestion control. In *2016 ACM SIGCOMM*. 230–243.
[22] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P Pezaros, Stefan Schmid, and Gábor Rétvári. 2019. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *15th International Conference on Emerging Networking Experiments And Technologies*. 292–304.
[23] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 254–265.
[24] 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 *15th USENIX symposium on networked systems design and implementation (NSDI 18)*. 373–387.
[25] Ivan Bjerre Damgård. 1989. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*. 416–427.
[26] Shaojiang Deng, Yantao Li, and Di Xiao. 2010. Analysis and improvement of a chaos-based Hash function construction. *Communications in Nonlinear Science and Numerical Simulation* 15, 5 (2010), 1338–1347.
[27] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. 2021. Distrinet: A mininet implementation for the cloud. *ACM SIGCOMM Computer Communication Review* 51, 1 (2021), 2–9.
[28] Paul Emmerich, Daniel Raumer, Sebastian Gallenmüller, Florian Wohlfart, and Georg Carle. 2018. Throughput and latency of virtual switching with Open vSwitch: a quantitative analysis. *Journal of Network and Systems Management* 26 (2018), 314–338.
[29] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 315–328.
[30] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.
[31] GNU. 2018. GNU Time. (2018). https://www.gnu.org/software/time/, [Accessed: Aug. 30, 2024].
[32] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *ACM SIGCOMM 2009*. 51–62.
[33] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[34] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.
[35] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 465–478.
[36] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Gu, Wes Felter, John Carter, and Aditya Akella. 2016. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *2016 ACM SIGCOMM*. 244–257.
[37] C Hopps. 2000. RFC2992: Analysis of an equal-cost multi-path algorithm. (2000).
[38] Petr Horacek. 2023. OvS Container Network Interface. (2023). https://github.com/k8snetworkplumbingwg/ovs-cni [Accessed: Jul. 15, 2023].
[39] Zhicong Huang, Erman Ayday, Jacques Fellay, Jean-Pierre Hubaux, and Ari Juels. 2015. GenoGuard: Protecting genomic data against brute-force attacks. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 447–462.
[40] Huawei. 2021. Configuring the ECMP Load Balancing Mode. (2021). https://support.huawei.com/enterprise/en/doc/EDOC1100137942/60585466/configuring-the-ecmp-load-balancing-mode [Accessed: Jul. 5, 2023].
[41] Hirofumi Ichihara and T Tsuchiya. 2019. LINE Data Center Networking with SRv6. (2019).
[42] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.
[43] Heesang Jin, Minkoo Kang, Gyeongsik Yang, and Chuck Yoo. 2019. CLEO: Machine learning for ECMP. In *15th International Conference on emerging Networking EXperiments and Technologies*. 1–3.
[44] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. 2015. Efficient traffic splitting on commodity switches. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–13.
[45] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. 2017. Clove: Congestion-aware load balancing at the virtual edge. In *13th International Conference on emerging Networking EXperiments and Technologies*. 323–335.
[46] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 157–170.
[47] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 1–6.
[48] Jialong Li, Haotian Gong, Federico De Marchi, Aoyu Gong, Yiming Lei, Wei Bai, and Yiting Xia. 2024. Uniform-Cost Multi-Path Routing for Reconfigurable Data Center Networks. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 433–448. https://doi.org/10.1145/3651890.3672245
[49] Xing Li, Xiaochong Jiang, Ye Yang, Lilong Chen, Yi Wang, Chao Wang, Chao Xu, Yilong Lv, Bowen Yang, Taotao Wu, et al. 2024. Triton: A Flexible Hardware Offloading Architecture for Accelerating Apsara vSwitch in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 750–763.
[50] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. 2021. {eMRC}: Efficient Miss Ratio Approximation for {Multi-Tier} Caching. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 293–306.
[51] Abhijeet Mahapatra, Santosh K. Majhi, Kaushik Mishra, Rosy Pradhan, D. Chandrasekhar Rao, and Sandeep K. Panda. 2024. An Energy-Aware Task Offloading and Load Balancing for Latency-Sensitive IoT Applications in the Fog-Cloud Continuum. *IEEE Access* 12 (2024), 14334–14349. https://doi.org/10.1109/ACCESS.2024.3357122
[52] Sam Mansfield, Kerry Veenstra, and Katia Obraczka. 2022. Modeling communication over terrain for realistic simulation of outdoor sensor network deployments. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 6, 4 (2022), 1–22.
[53] Regis Francisco Teles Martins, Rodolfo da Silva Villaça, and Fábio Luciano Verdi. 2020. Bitmatrix: a multipurpose sketch for monitoring of multi-tenant networks. *Journal of Network and Systems Management* 28, 4 (2020), 1745–1774.
[54] Jiří Matoušek, Gianni Antichi, Adam Lučanskỳ, Andrew W Moore, and Jan Kořenek. 2017. ClassBench-ng: Recasting ClassBench after a decade of network evolution. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 204–216.
[55] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1148–1164. https://doi.org/10.1145/3503222.3507769
[56] Microsoft. 2023. Software Defined Networking (SDN) in Azure Stack HCI and Windows Server. (April 2023). https://learn.microsoft.com/en-us/azure-stack/hci/concepts/software-defined-networking (Accessed on 10/3/2024).

[57] Timothy Prickett Morgan. 2023. META Platforms is determined to make ethernet work for AI. (2023). https://www.nextplatform.com/2023/09/26/meta-platforms-is-determined-to-make-ethernet-work-for-ai/ [Accessed: Mar. 15, 2024].
[58] NADDOD. 2023. High-Performance GPU Server Hardware Topology and Cluster Networking. (2023). https://www.naddod.com/blog/high-performance-gpu-server-hardware-topology-and-cluster-networking-2 [Accessed: Mar. 15, 2024].
[59] OpenStack. 2022. Open vSwitch: Self-service networks. (2022). https://docs.openstack.org/neutron/queens/admin/deploy-ovs-selfservice.html [Accessed: Jul. 15, 2023].
[60] Tomasz Osiński, Jan Palimąka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. 2022. A novel programmable software datapath for software-defined networking. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*. 245–260.
[61] Michele Paolino, Nikolay Nikolaev, Jeremy Fanguede, and Daniel Raho. 2015. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 86–92.
[62] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. 2013. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 207–218.
[63] Ruxandra-Stefania Petre et al. 2012. Data mining in cloud computing. *Database Systems Journal* 3, 3 (2012), 67–71.
[64] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 117–130.
[65] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. 2022. Jupiter evolving: transforming Google's datacenter network via optical circuit switches and software-defined networking. In *ACM SIGCOMM 2022*. 66–85.
[66] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. 2024. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 691–706.
[67] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. 2022. PLB: congestion signals are simple and effective for network load balancing. In *ACM SIGCOMM 2022*. 207–218.
[68] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving datacenter performance and robustness with multipath TCP. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 266–277.
[69] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2022. Scaling Open vSwitch with a Computational Cache. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.
[70] RedHat. 2024. Configuring an OVS-DPDK deployment | Red Hat Product Documentation. (2024). https://docs.redhat.com/en/documentation/red_hat_openstack_platform/17.1/html/configuring_network_functions_virtualization/config-dpdk-deploy_rhosp-nfv, [Accessed: Sep. 15, 2024].
[71] Luigi Rizzo and Giuseppe Lettieri. 2012. VALE, a switched Ethernet for virtual machines. In *8th International Conference on Emerging Networking Experiments and Technologies*. 61–72.
[72] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A programmable, protocol-independent software switch. In *2016 ACM SIGCOMM*. 525–538.
[73] Yuki Tsujimoto, Yuki Sato, Kenichi Yasukata, Kenta Ishiguro, and Kenji Kono. 2024. PvCC: A vCPU Scheduling Policy for DPDK-applied Systems at Multi-Tenant Edge Data Centers. In *Proceedings of the 25th International Middleware Conference*. 379–391.
[74] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open vSwitch dataplane ten years later. In *2021 ACM SIGCOMM*. 245–257.
[75] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 407–420.
[76] VMWare. 2022. NSX Virtual Distributed Switch. (2022). https://docs.vmware.com/en/VMware-NSX-T-Data-Center/3.2/administration/GUID-A075811A-03EC-4F79-A9F1-1F1CF52722DB.html [Accessed: Mar. 15, 2024].
[77] VMWare. 2023. 3-Node Cluster with vSAN Reference Design. (2023). https://docs.vmware.com/en/VMware-Edge-Compute-Stack/1.0/ecs-enterprise-edge-ref-arch/GUID-355C386D-16C0-49DA-9775-9959ADBD0326.html [Accessed: Mar. 15, 2024].
[78] Fu Wang, Haipeng Yao, Qi Zhang, Jingjing Wang, Ran Gao, Dong Guo, and Mohsen Guizani. 2021. Dynamic distributed multi-path aided load balancing for optical data center networks. *IEEE Transactions on Network and Service Management*

19, 2 (2021), 991–1005.
[79] Peng Wang, Hong Xu, Zhixiong Niu, Dongsu Han, and Yongqiang Xiong. 2016. Expeditus: Congestion-aware load balancing in clos data center networks. In *Seventh ACM Symposium on Cloud Computing*. 442–455.
[80] Jon Whiteaker, Fabian Schneider, and Renata Teixeira. 2011. Explaining packet delays under virtualization. *SIGCOMM Comput. Commun. Rev.* 41, 1 (Jan. 2011), 38–44. https://doi.org/10.1145/1925861.1925867
[81] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 50–61.
[82] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. 2011. Design, implementation and evaluation of congestion control for multipath TCP. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.
[83] Yunhong Xu, Keqiang He, Rui Wang, Minlan Yu, Nick Duffield, Hassan Wassel, Shidong Zhang, Leon Poutievski, Junlan Zhou, and Amin Vahdat. 2022. Hashing Design in Modern Networks: Challenges and Mitigation Techniques. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 805–818.
[84] Gyeongsik Yang, Changyong Shin, Jeunghwan Lee, Yeonho Yoo, and Chuck Yoo. 2022. Prediction of the resource consumption of distributed deep learning systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–25.
[85] Gyeongsik Yang, Yeonho Yoo, Minkoo Kang, Heesang Jin, and Chuck Yoo. 2021. Bandwidth isolation guarantee for SDN virtual networks. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.
[86] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang
[87] Chaobing Zeng, Fangming Liu, Shutong Chen, Weixiang Jiang, and Miao Li. 2018. Demystifying the performance interference of co-located virtual network functions. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 765–773.
[88] Nannan Zhang, Wei Wang, Xiaofeng Xin, Yuanwei Liu, Hangguan Shan, and Aiping Huang. 2024. Low-Delay Ultra-Small Packet Transmission With In-Network Aggregation via Distributed Stochastic Learning. *IEEE Transactions on Communications* 72, 5 (2024), 2655–2669.
[89] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, Luigi Iannone, and James Roberts. 2019. Comparing the performance of state-of-the-art software switches for NFV. In *15th International Conference on Emerging Networking Experiments And Technologies*. 68–81.
[90] Yiran Zhang, Jun Bi, Zhaogeng Li, Yu Zhou, and Yangyang Wang. 2020. VMS: Load balancing based on the virtual switch layer in datacenter networks. *IEEE Journal on Selected Areas in Communications* 38, 6 (2020), 1176–1190.
[91] Zhehui Zhang, Haiyang Zheng, Jiayao Hu, Xiangning Yu, Chenchen Qi, Xuemei Shi, and Guohui Wang. 2021. Hashing Linearity Enables Relative Path Control in Data Centers. In *USENIX Annual Technical Conference*. 855–862.
[92] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. 2023. Scalable tail latency estimation for data center networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 685–702.
[93] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. 2014. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Ninth European Conference on Computer Systems*. 1–14.
[94] Zhuping Zou, Yulai Xie, Kai Huang, Gongming Xu, Dan Feng, and Darrell Long. 2019. A docker container anomaly monitoring system based on optimized isolation forest. *IEEE Transactions on Cloud Computing* 10, 1 (2019), 134–145.
[95] Annus Zulfiqar, Ben Pfaff, William Tu, Gianni Antichi, and Muhammad Shahbaz. 2023. The Slow Path Needs an Accelerator Too! *SIGCOMM Comput. Commun. Rev.* 53, 1 (April 2023), 38–47. https://doi.org/10.1145/3594255.3594259

Received January 2025; revised April 2025; accepted April 2025