CrossBal: Data and Control Plane Cooperation for Efficient and Scalable Network Load Balancing

Bruno L. Coelho, Alberto E. Schaeffer-Filho
Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
{blcoelho, alberto}@inf.ufrgs.br

***Abstract*—Load balancing network traffic through multiple shortest-paths has become common practice to efficiently utilize the network infrastructure. Despite widespread adoption, Equal-Cost Multi-Path (ECMP) delivers performance far from optimal. Several load balancing solutions utilize Weighted-Cost Multi-Path (WCMP), splitting incoming traffic between links proportionally to link weights. However, implementing WCMP requires the controller to update match+action rules whenever the weights must be changed, introducing a delay before the appropriate traffic split can be applied. Additionally, weighted traffic splits are applied over network flows without regard to flow characteristics or needs. We propose CrossBal, a hybrid load balancing system based on Deep Reinforcement Learning (DRL) that focuses its efforts on high-impact elephant flows. The DRL agent is modeled to be able to efficiently utilize network links while minimizing the action space, allowing the agent to quickly learn how to load balance. Further, CrossBal can quickly react to network changes by monitoring and switching active routes directly in the data plane. Our evaluation shows that CrossBal can efficiently utilize network resources, using most available links, while also reducing link utilization imbalance. We also evaluate the elephant flow detection employed by CrossBal, showing how it can quickly identify elephant flows while efficiently utilizing switch resources.**

***Index Terms*—Load Balancing, Traffic Engineering, Deep Reinforcement Learning, Machine Learning, Programmable Data Planes, Elephant Flow**

I. Introduction

Current intra-domain routing solutions present limitations in properly trying to load balance network traffic. Equal-Cost Multi-Path (ECMP) evenly splits traffic between multiple equal-cost paths. Due to its simplicity, ECMP is readily available in commercial switches [1]. However, ECMP suffers from severe performance drawbacks, being unable to achieve adequate performance [2]. Weighted-Cost Multi-Path (WCMP) extends ECMP by adding weights to each hop, increasing performance and resilience to network asymmetry. Several systems propose techniques for calculating optimal weights for WCMP [3]–[7]. However, updating link weights during congestion requires control plane intervention, which introduces considerable delay. On the other hand, load balancing solutions that rely entirely on the data plane are limited to specific topologies [8], [9] or simple heuristics [10], [11].

In addition to the aforementioned deficiencies, existing load balancing systems based on traffic splits generally do not consider the characteristics or needs of each network flow. Elephant flows are high-throughput, long-lasting flows that tend to have a large impact on the network [12]. While elephant flows may constitute a small portion of total flows, a few large flows tend to contribute more to the overall network traffic than a large amount of small flows [13], [14]. Considering the impact that elephant flows have on the network, intelligent rerouting of these flows can severely improve network utilization [15]. Additionally, as elephant flows are long-lived, we have more chances to reroute them.

Given the importance of load balancing elephant flows, a system capable of identifying and rerouting these flows is required. However, the identification of elephant flows requires monitoring up to terabits per second of network traffic. While a control plane solution can enable complex techniques for identifying elephant flows, SDN controllers cannot process network traffic at these rates [16]. An alternative is to use the data plane of networking devices to aid in the identification of elephant flows. While emerging programmable switches [17] allow us to reconfigure the packet processing pipeline, they are still subject to limitations, as these devices tend to have a few tens of MBs of memory, a restricted set of logical and arithmetic operations, limitations on memory accesses, and a strict time budget to process each packet [18].

In addition to efficiently and accurately detecting elephant flows, a load balancer must be able to react to changes in the network state, such as transient congestion. Considering these requirements, we propose CrossBal, a hybrid load balancing system that combines an intelligent control plane with a reactive data plane. CrossBal employs a Deep Reinforcement Learning agent in the control plane, responsible for intelligently selecting routes that maximize the performance of the network. Additionally, by having the control and data planes collaborate to identify elephant flows, CrossBal avoids scalability and delay issues that are introduced by performing per-packet computation in the controller. Finally, CrossBal is able to quickly detect and react to congestion in active paths by employing mechanisms directly in programmable data planes.

In summary, this work presents the following contributions:

- **Design and implementation of CrossBal:** a hybrid machine learning-aided load balancing system capable of identifying and rerouting elephant flows, as well as detecting and reacting to congestion in selected paths;
- **Evaluation of a PoC prototype:** using BMv2[1] in an emulated environment with realistic network topologies and workloads;

[1]https://github.com/p4lang/behavioral-model

978-3-903176-59-1 ©2023 IFIP

- **Design of a deep reinforcement learning agent:** which is capable of actively load balancing network flows.

II. Background and Motivation

This section provides necessary background information on Deep Reinforcement Learning and programmable data planes.

A. Deep Reinforcement Learning

In Reinforcement Learning (RL), an agent interacts with its *environment* in each timestep $t \in 1, 2, ...$ In each iteration, the agent observes a state $s \in S$ and chooses an action $a \in A$ according to its policy π [19]. Afterwards, the agent receives a reward r according to a *reward function* and transitions to a new state $s' \in S$ according to a *transition function*. RL algorithms can learn an optimal policy π even without any explicit knowledge of the reward or transition functions [20].

In order to be able to efficiently utilize Reinforcement Learning algorithms in systems with complex environments, researchers have proposed the use of *Deep Neural Networks* (DNNs) as a function approximator for RL [21] - a technique known as Deep Reinforcement Learning (DRL).

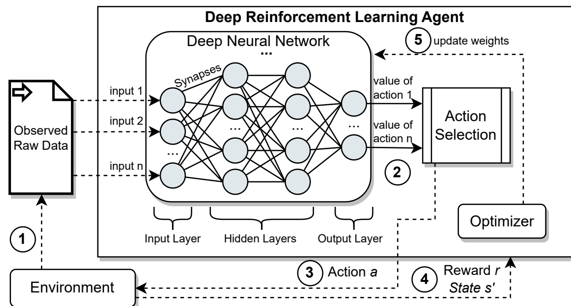


Fig. 1: Steps of an iteration of a Deep Learning Agent

Figure 1 illustrates the main aspects of a Deep Reinforcement Learning (DRL) agent. First, (1) the DRL agent *observes raw data* which describes its current state s. The agent utilizes a DNN to identify similar states, improving scalability and efficiency. Then, (2) the DNN outputs the expected *value of each action* for the observed state as the value of each neuron in the output layer. Next, (3) the agent *selects an action* a based on its strategy. This typically involves choosing between *exploiting*, i.e., choosing the action with the highest expected value, or *exploring* a random action, based on its exploration parameters. After acting, (4) the agent receives a *reward* r and *transitions* to a *new state* s'. Finally, (5) the agent updates its *internal weights* based on the reward received.

B. Programmable Data Planes

Programmable data planes allow network operators to redefine the packet-processing pipeline through domain-specific languages, such as P4 [17]. In the PISA architecture, the data plane is mainly composed of a parser, an ingress, and an egress processing blocks [9]. The first step in the pipeline is the *parser*, responsible for parsing protocol headers. Next, the *ingress* block processes packets and selects an egress port. In this block, the network operator can define *match+action* tables, matching on arbitrary keys, such as packet header fields or custom metadata, and invoking actions defined by the operator. Actions can be defined based on simple arithmetic and logic primitives, as well as architecture-specific functions. This includes reading and storing data in registers, allowing stateful processing. The egress processing block is identical to the ingress processing block, except the output port has already been selected. Both blocks have an independent amount of resources, such as SRAM and arithmetic and logic units. State-of-the-art programmable switches have a few tens of MBs of SRAM and a limited amount of pipeline stages [18].

III. CrossBal: Cross-Plane Load Balancing

CrossBal is a hybrid load balancing system that combines an intelligent control plane with reactive data plane processing. In this section, we present CrossBal, starting with an overview of the approach (§III-A), followed by the key design elements, including elephant flow detection (§III-B), DRL agent (§III-C) and how to react to short-lived network congestion (§III-D).

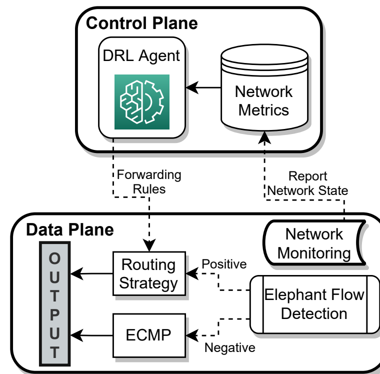


Fig. 2: CrossBal employs cross-plane collaboration.

A. Approach Overview

Figure 2 presents an overview of our approach. CrossBal relies on two *key principles* to balance network utilization in an efficient and scalable manner: *(i)* cross-plane cooperation for combining line rate reaction to network changes at the data plane with more intelligent decisions at the control plane; and *(ii)* scalable traffic rerouting for flows that have the highest impact on the network (e.g., elephant flows and heavy hitters) as opposed to dealing with every flow in an equal manner.

The workflow starts with each programmable data plane device monitoring the state of the network. Simultaneously, the data plane is also responsible for performing elephant flow detection at line rate. Both the network status and detected elephant flows are reported to a logically centralized controller, with a global view of the network. The controller employs a Deep Reinforcement Learning (DRL) agent to actively compute the new *top-n* optimal routes to forward these flows of interest, and reconfigure the data plane devices.

CrossBal employs two control-loops that work together for performing load balancing. There is a *slower, but more*

intelligent, control-loop at the control plane, which is fed with network monitoring data and is used by the DRL agent to compute the optimal routes. However, programmable data plane devices also apply a *faster, but simpler, control-loop* to probe, monitor, and rapidly switch between a subset of active routes selected by the DRL agent. By having the data plane cooperate with the control plane in multiple aspects, CrossBal achieves intelligent and reactive load balancing of the network.

There are several challenges that directly influenced the following *design aspects* of CrossBal: the identification of elephant flows, the modeling of a Deep Reinforcement Learning agent, and allowing data plane devices to actively participate in route selection. These will be discussed next.

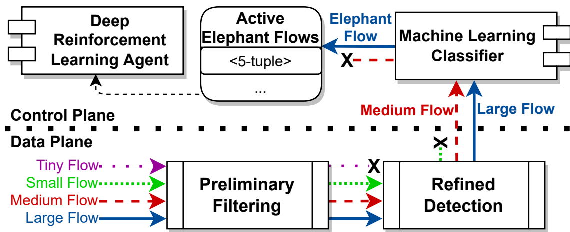


Fig. 3: Overview of the cross-plane elephant flow detection.

B. Identifying Elephant Flows at Line-Rate

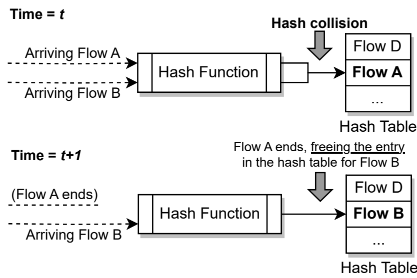
Identifying elephant flows at line-rate is challenging in high-throughput networks, where network traffic rates can reach terabits per second [16]. Despite its flexibility, a centralized controller is incapable of performing per-packet classification without incurring latency overheads. Programmable switches [17] can be used to offload part of this task.

CrossBal *decomposes the detection* of elephant flows into three levels of complementary mechanisms, balancing the tradeoffs between fast and lightweight detection in the data plane with more accurate and heavyweight detection in the control plane (Figure 3):

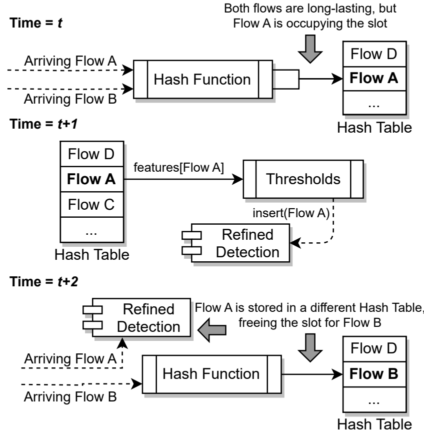
- ***Preliminary filtering:*** the data plane implements a threshold-based detection that tracks the number of bytes, number of packets, and duration for each active flowlet[2]. The main aspect of this mechanism is that it must handle a large number of flows, thus limiting the amount of processing and storage available for each flow. Therefore, as shown in Figure 3, this step acts as an early filtering of low-throughput and short flows in order to save hardware resources. A further optimization is proposed and evaluated (§V-D), where only packets larger than a threshold are accounted for. By utilizing this strategy, the number of packets of a flowlet can be seen as a lower bound of the number of bytes transmitted by the flowlet. This can lower the number of bits utilized to store this information, saving precious on-board memory. The same reasoning can be applied to track the number of flowlet timeouts of a flow rather than the entire duration.
- ***Refined detection:*** While the threshold-based mechanism mentioned above can exclude a large number of small and short flows, it may lead to a high number of false positives if used by itself. To address this, CrossBal employs further mechanisms to detect elephant flows. The intuition is that because the preliminary threshold-based filtering already excluded a large number of unimportant flows, it is now possible to implement a slightly *refined detection* mechanism over the remaining flows of interest, as shown in Figure 3. In particular, CrossBal implements a classification tree in the form of *if-else* statements in the programmable data plane. As there is a smaller number of flows to consider, it is possible to dedicate slightly more on-board memory to track *features* of each flowlet. In our PoC prototype, we track simple statistics of the inter-arrival-time and packet size of each flowlet. We leave a more thorough feature selection as future work.
- ***Cross-plane detection:*** Although CrossBal implements multiple mechanisms for the identification of elephant flows directly in the data plane, ensuring line-rate processing requires sacrificing accuracy for efficiency. In order to provide a more accurate detection of elephant flows, CrossBal employs cross-plane collaboration, as shown in Figure 3. This builds upon the *preliminary filtering* (which reduces the amount of flows of interest) and upon the *refined detection* (which tracks additional features for relevant flows). Therefore, CrossBal implements a classifier in the control plane that receives the information tracked by the data plane. As the controller provides a more flexible programming model, and considering the features extracted by the data plane, we implement a Random Forest in the control plane, providing higher accuracy than a single classification tree.

[2]Flowlets are bursts of packets of a flow separated by an idle interval.

CrossBal utilizes hash tables to implement the *preliminary filtering* and the *refined detection*. Since onboard memory is a scarce resource, collisions in the hash tables are unavoidable. However, due to the multi-stage elephant flow detection spanning both the data and control planes, hash collisions do not cause elephant flows to pass undetected. Figure 4 shows examples of hash collisions that may happen during the detection of elephant flows in the data plane. Particularly, when one of the colliding flows is a short-lived flow (Figure 4a), this flow tends to complete while the elephant flow is still active. This means that the elephant flow will eventually be able to utilize the hash table entry once the short-lived flow expires. In another scenario (Figure 4b), when the hash of two (undetected) elephant flows lead to the same table entry, the first flow (A) will occupy the slot. Eventually, the *preliminary filtering* will recognize flow A as being a potential elephant flow, inserting it in the list of flows tracked by the *refined detection*. This way, flow A will be removed from the first hash table, freeing the slot for the second flow (B). This same reasoning is applicable to hash collisions in the *refined detection*, as flows are eventually exported to the controller, freeing the occupied slot.



(a) Hash collision between a mice flow and an elephant flow.



(b) Hash collision between two elephant flows.

Fig. 4: Relevant scenarios where hash collision may happen.

The control plane is responsible for the final classification of potential elephant flows. Hardware limitations of data plane devices impose restrictions on the complexity of the classification models implemented, and as such the control plane is a more advantageous place to implement a complex machine learning classifier with high accuracy. Once an elephant flow is identified, it is inserted in a list of flows to be actively rerouted by the DRL agent that executes in the controller (next section).

C. Deep Reinforcement Learning Agent

The Deep Reinforcement Learning (DRL) agent is responsible for selecting routes for active elephant flows, with the objective of improving network utilization. In particular, the modeling of the state, action space, and rewards impact how well the agent can achieve its goal.

State Space. The state must include all the information the agent needs in order to take an appropriate decision at a given time, but real time data collection poses several scalability challenges. In particular, switches must refrain from exporting unnecessary information and, at the same time, must minimize the amount of redundant information, which may negatively impact the learning rate of the DRL agent.

Our approach: In order to avoid the aforementioned problems, we model the state based on the utilization of each link in the network. More formally, the state S observed by the agent is a vector of the utilization Ui, j of each link i of every switch j:

$$S = (U_{1,1}, U_{1,2}, ..., U_{1,n}, U_{2,1}, ..., U_{2,n}, ..., U_{n,m})$$

The link utilization of the ports of each switch in the network is enough for the agent to understand the current state of the available network resources. Further, the agent is able to learn the relationship between routes and links by observing how each action taken affects the utilization of links.

As the agent is used to reroute active elephant flows, the state must also consider the endpoints of the flow. However, using the 5-tuple of the flow requires the agent to learn the mapping of IP addresses, leading to slower learning. Instead, we map the source and destination IP addresses to one-hot vectors representing the source and destination edge switches.

Action Space. The action space must allow the agent to make decisions on how to reroute active (elephant) flows. A naïve approach would be to map each possible route to an action. However, this would generate a large number of possible actions, leading to slow learning. An alternative would be to model each action as one hop, as shown in Figure 5. In this approach, an end-to-end route would require several hops, i.e., several actions being taken in a sequence. This leads to the problem of reward assignment, as a sequence of actions would be required to obtain a single reward.

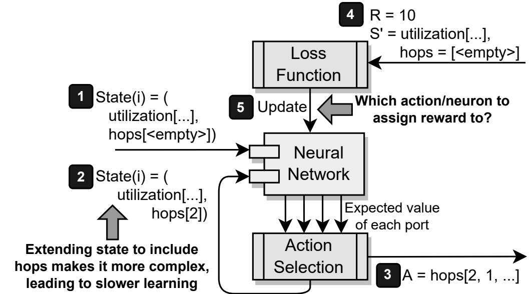


Fig. 5: Mapping actions to hops leads to reward assignment issues.

Our approach: We model the action space based on a set of predefined end-to-end routes for each pair of source-destination edge switches. First, this design eliminates the need for multiple actions per end-to-end route. Instead, the agent can observe the correlation between choosing a route (taking an action) and changes in the utilization of the links in the network (observing the next state). Additionally, restricting the action space to a set of predefined routes results in a well-defined number of possible actions. This avoids potential issues that might arise if the agent had to consider every single possible end-to-end route in the network. Finally, the set of end-to-end routes for each source-destination pair should remain the same throughout the life of the DRL agent, i.e., from training to the end of its use in load balancing. Computing short paths with diversity can be done with algorithms such as KSPD [22]. The computed paths can be used by the agent to effectively distribute the load over the network links while

keeping the number of possible actions small. While topology changes require recomputing the static routes and retraining the agent with the new routes, Graph Neural Networks (GNNs) could make the agent robust to topology changes [23]. We leave this as future work.

Reward. In the context of network load balancing, the reward is linked to the utilization of network resources. Although in traditional DRL the impact of an action in the environment is reflected immediately, for load balancing the state of the links may take some time to update. If the utilization of the network links is queried immediately after the action is taken, it will not properly reflect the impact of the selected action.

Our approach. Considering the main objective of load balancing the network, we model the reward $R(s, a)$ after the max link utilization of the network. After comparing different formulas based on the link utilization to compute the reward (§V-C), we observed that (1) produced the best results. Additionally, in order for the new state to reflect the consequences of the action taken, we poll link statistics from switches t milliseconds after an action is taken.

$$R(s, a) = \frac{1}{MaxLinkUtilization(s')} \quad (1)$$

D. Reacting to Short-Lived Network Congestion

The DRL agent is used to reroute elephant flows as soon as they are detected. As explained above, the agent takes into consideration the current utilization of the links in the network to select an optimal route with respect to network load balancing. Additionally, when no new elephant flows are detected, the agent is used to periodically select new routes for already active elephant flows. This is useful when the utilization of certain links in the network changes considerably, causing alternative routes to become more favorable. However, due to the fact that this control loop is only executed periodically, it may not be capable of reacting to short-lived congestion.

Thus, in order to be able to quickly react to network changes, CrossBal implements mechanisms for switching active paths directly in the data plane. Figure 6 presents an overview of this mechanism, where *(i)* the programmable data plane actively monitors the quality of the installed routes, and *(ii)* upon detecting congestion, *(iii)* the forwarding device switches to a less congested pre-computed route without control plane intervention.

CrossBal achieves this by having the controller install multiple routes for each active elephant flow. As the DRL agent computes the expected value of each route, we select the N routes with highest expected value. Also, the data plane devices are responsible for periodically probing each of the installed routes. By calculating the RTT of each route, the programmable switch is capable of detecting congestion along paths and selecting an alternate route. Spraying a flow's data packets through paths we wish to probe could lead to packet reordering at the destination end-host. This can negatively affect the performance of transport protocols such as TCP [24]. Instead, CrossBal periodically creates probe

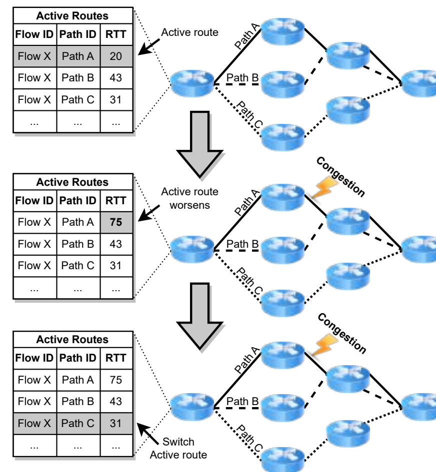


Fig. 6: Mechanism for switching active paths

packets to measure the RTT of active paths. Probe packets are created using the *clone* feature of programmable switches when it has been longer than t ms since the last probing for this elephant flow. The cloned packets have their payload removed and a custom probing header inserted. Each active route is probed in order to measure its RTT. As the control plane is only required initially to install the multiple routes, we can quickly detect and react to short-lived congestion. An elephant flow is rerouted when there is a noticeable increase in measured RTT in the active route. Therefore, even if every route is experiencing congestion, CrossBal will only reroute once per probing interval.

IV. Architecture

CrossBal comprises a series of modules divided in the control and data planes. The architecture of our system is shown in Figure 7. The data plane of programmable devices includes modules for monitoring network links, routing elephant flows, detecting new elephant flows, monitoring active end-to-end routes, and switching active end-to-end routes. Algorithm 1 provides the pseudo-code of the packet processing pipeline of the P4 switches used by CrossBal.

Upon receiving a packet that does not belong to an elephant flow, the programmable switch applies a `Preliminary Filtering` to exclude short-lived and small flows (lines 14-23 of Algorithm 1). For each flowlet, we use registers to track a few simple features, such as the number of bytes, packets, and flowlet duration. When the features of a flowlet exceed predefined thresholds, the flowlet is set to be processed by a second module, responsible for refined detection. The `Refined Detection` is only applied over a smaller subset of flows, enabling the data plane to keep track of more complex features for each tracked flow (lines 10-13 of Algorithm 1). The features tracked in this module include the minimum, maximum, and total inter-arrival-time and packet length of each flow. These features are tested against a set of chained conditions in order to identify potential elephant

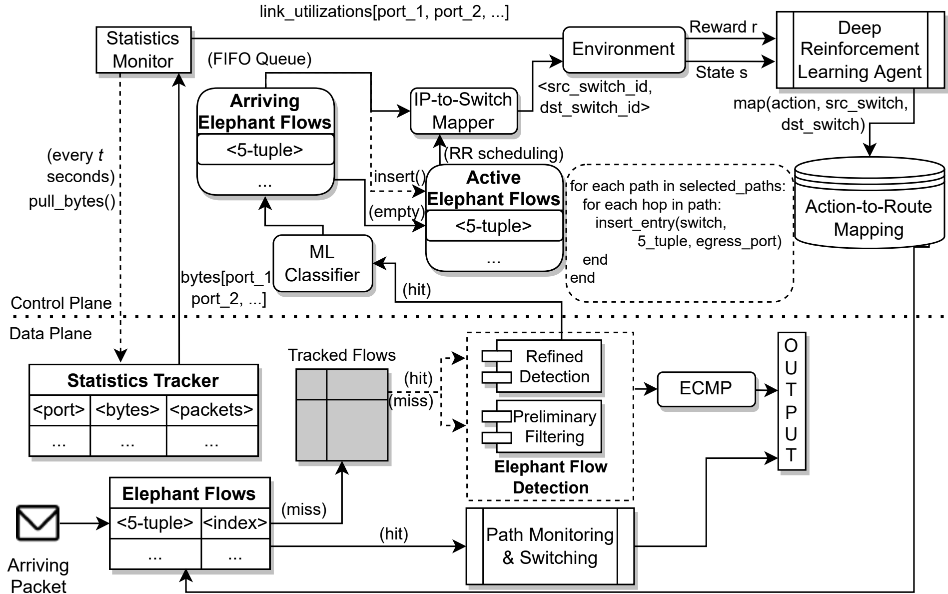


Fig. 7: Architectural implementation of CrossBal.

Algorithm 1: Data plane packet processing pipeline

Data: $pkt \leftarrow$ Packet In
Data: $flow \leftarrow pkt.5_tuple$
1 **if** $flow$ *is in* $elephant_flows$ **then**
2 $rtt_diff \leftarrow active_route.curr_rtt - active_route.prev_rtt$;
3 **if** $rtt_diff \geq$ *RTT Threshold* **then**
4 $active_route[flow] \leftarrow min(installed_routes[flow])$;
5 $time_since_probing \leftarrow curr_time - last_probe[flow]$;
6 **if** $time_since_probing \geq$ *Probing Interval* **then**
7 $create_probes(installed_routes[flow])$;
8 $egress_port \leftarrow active_route[flow].egress_port$;
9 **else**
10 **if** $flow$ *is in* $refined_detection.tracked_flows$ **then**
11 $features[flow] \leftarrow compute_features(flow)$;
`// If statements are automatically generated`
12 **if** $feature_1[flow] \geq$ *Feature 1 Threshold* ***AND*** $feature_3[flow] <$ *Feature 3 Threshold* **then**
13 $notify_controller(flow, features[flow])$;
14 **else**
15 **if** *Bytes Optimization is enabled* **then**
16 **if** $pkt.length >$ *Length Threshold* **then**
17 $packets[flow] \leftarrow packets[flow] + 1$;
18 $bytes[flow] \leftarrow packets[flow] *$Length Threshold;
19 **else**
20 $bytes[flow] \leftarrow bytes[flow] + pkt.length$;
21 $flow_duration \leftarrow curr_time - flow_start[flow]$;
22 **if** $bytes[flow] \geq$ *Bytes Threshold* ***AND*** $flow_duration \geq$ *Duration Threshold* **then**
23 $refined_detection.track(flow)$;
24 $egress_port \leftarrow ecmp(pkt.5_tuple)$;

flows (line 12 of Algorithm 1). This is achieved by converting a Classification Tree to a series of conditions. If the flow is labeled as a potential elephant flow, it is exported to the controller for a final classification, along with the computed features of that flow (line 13 of Algorithm 1).

The data plane further implements the `Statistics Tracker`, which is responsible for monitoring statistics of each switch port. Each switch exports this local information to the controller, which computes the link utilization of every link in the network. The `Environment` utilizes the computed link utilization, along with a FIFO queue and a Round-Robin list of detected Elephant Flows, to produce the State observed by the `DRL Agent`. The agent is queried to compute the expected value of the top-k routes for a given elephant flow. Among these, the N actions with highest expected values are translated into routes, which can be efficiently achieved by looking up the `Action-to-Route Mapping` table[3]. The N routes are then installed in each forwarding device.

The controller periodically queries the `DRL agent` to select routes for active elephant flows. However, this control loop may not be able to quickly react to short-lived congestion. Therefore, the data plane implements mechanisms for `Path Monitoring and Switching` (lines 1-8 of Algorithm 1). Each active elephant flow has N routes installed in the data plane, which can freely switch between them. Each route has a different register array[4] to keep track of its two last observed RTTs. The route selection happens by electing an active route, which remains selected until the RTT of that route worsens by a certain threshold (e.g., 200%). Upon detecting a degradation in the selected route, the data plane picks the route with the lowest last measured RTT (lines 2-4 of Algorithm 1).

V. Evaluation

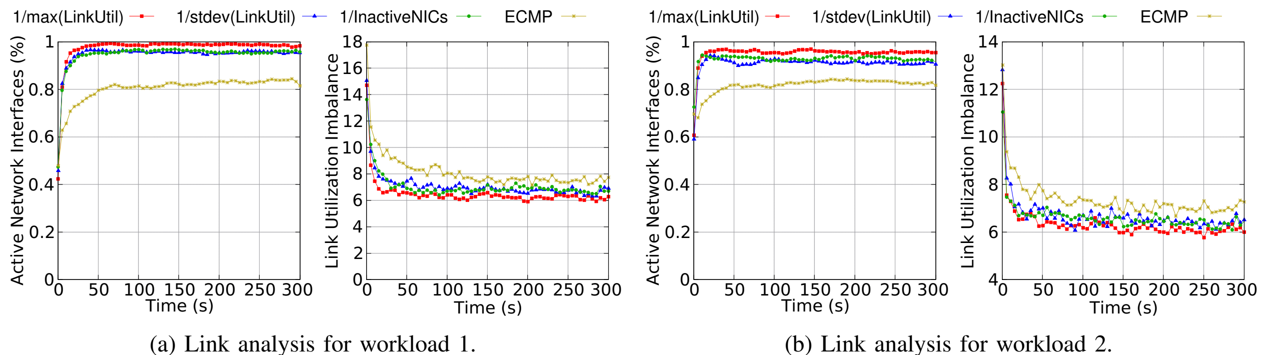
We implemented and evaluated a prototype of CrossBal in order to validate its design. Our experiments aimed to evaluate how well CrossBal could perform load balancing, as well as understand how key parameters might impact its performance.

A. Prototype

The prototype includes data plane software written in P4 and control plane software written in Python 3. The P4 source-code

[3]The mapping of actions to routes is computed offline for each pair of Source-Destination switches.

[4]The index used for each active elephant flow is configured by the controller upon installing new routes, allowing the controller to avoid any collisions.



(a) Link analysis for workload 1.

(b) Link analysis for workload 2.

Fig. 8: Link Utilization Imbalance and Active NICs.

was written for the BMv2[5] software switch. We used graph-tool v2.45 to compute ECMP routes and the top-k routes that constitute the action space of the agent. We also used Scapy v2.5.0 to send and receive packets between the controller and the software switches[6]. In order to facilitate and speedup some of the computing, we used NumPy v1.23.4. Finally, to implement the Deep Reinforcement Learning agent, we used PyTorch v1.12.1 for the DNN and Gym v0.26.2 to create a custom Reinforcement Learning environment. We used a DNN with 2 hidden layers with 512 neurons each. The layers are connected by ReLU activation functions, except for the output layer. We leave a more thorough exploration of the configuration of the DNN as future work.

B. Methodology

We emulated a network topology using Mininet. Inspired by [25], we used iGen to generate a "two-trees" intradomain mesh, obtaining a Hub & Spokes topology. This realistic class of topology is characterized by nodes with aggregation function (high degree of connectivity) [25]. Due to hardware limitations in the setup used in our experiments, we restricted the topology to 15 switches and the link speeds to 50 Mb/s. Similarly to [25], the workload used was also inspired by [8] [10]. Considering the restrictions on the topology used, we also reduced the flow sizes in the original workload. The flow size distribution of the workloads used in our experiments is described in Table I. Each switch in the topology has one host connected directly to it. Each host independently generates requests according to a Poisson distribution based on the workload and the desired network load. In our experiments, we had the DRL agent select N=3 out of K=10 precomputed routes for each elephant flow. N and K are parameters that should be set by the network operator based on the characteristics of the respective network topology, such as the average number of redundant paths between endpoints. Our prototype uses a simple greedy heuristic to compute routes, but algorithms such as KSPD [22] can be used to efficiently compute the shortest routes with diversity.

[5]BMv2 is the most recent version of the reference P4 software switch. Accessible at: https://github.com/p4lang/behavioral-model

[6]BMv2 switches currently do not support the full set of operations defined by P4Runtime, such as reading and writing to registers.

Workload A	
Flow Size	**Distribution**
20 KB	0.5
200 KB	0.3
2 MB	0.1
20 MB	0.1

Workload B	
Flow Size	**Distribution**
10 KB	0.2
100 KB	0.4
1 MB	0.2
10 MB	0.2

TABLE I: Workloads used in our evaluation.

C. Link Utilization Analysis

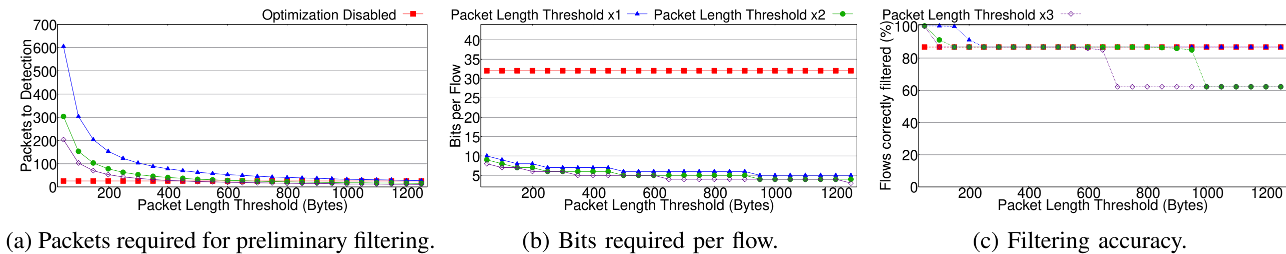
We implemented and compared three different reward functions for the DRL agent (§III-C): (A) $\frac{1}{max(link_util)}$, (B) $\frac{1}{stdev(link_util)}$, and (C) $\frac{1}{inactive_nics}$, where $link_util$ is an array with the utilization of every link in the network and $inactive_nics$ is the proportion of NICs not being utilized.

Figure 8a shows an analysis of the ratio of active links during our experiments. We can observe that CrossBal effectively utilizes nearly all available links in the network, while ECMP is incapable of utilizing as many links concurrently. Further, we can observe that the agent trained with Reward Function A, $\frac{1}{max(link_util)}$, quickly learns how to actively use nearly every link in the network, effectively distributing the workload. Additionally, Figure 8a compares the Link Utilization Imbalance[7] attained by each approach. CrossBal performs a better job at balancing link utilization across network links compared to ECMP. As before, we can observe that the agent trained with Reward Function A, $\frac{1}{max(link_util)}$, outperforms the agents trained with the other reward functions. Figure 8b shows the ratio of active NICs and the link utilization imbalance when running the same experiments with a different workload, where a similar behavior was observed.

D. Elephant flow detection optimizations

The preliminary filtering mechanism (§III-B) in the data plane must be able to filter a small number of possible elephant flows out of a large number of flows. Therefore, it is crucial to optimize the per-flow processing and storage requirements as much as possible. An optimization mentioned in Section III-B is to filter packets according to a specific threshold, only accounting for packets that are not too small. This way, rather

[7]Link Utilization Imbalance is a metric that takes into account the maximum, minimum, and average link utilization [8].



(a) Packets required for preliminary filtering. (b) Bits required per flow. (c) Filtering accuracy.

Fig. 9: Analysis of parameters for elephant detection optimization with a 30KB threshold.

than counting bytes, it becomes possible to *count packets*, while still having a lower bound of the size of the flow.

However, only accounting for the lower bound of the size of a flow may cause detection to take longer. For instance, with a packet length threshold of 500 bytes, it would take 20 packets (of at least 500 bytes) to reach a threshold of 10KB. However, by counting bytes, 7 packets of 1500KB (typical MTU value) would be enough to reach that same threshold. Therefore, a further optimization would be to adjust the *assumed size* of the packets without changing the packet length threshold.

Figure 9a shows the number of packets required for the `Preliminary Filtering` to forward a flow to the `Refined Detection` according to different thresholds for packet length, considering a detection threshold of 30KB[8]. We can observe that, by increasing the packet length threshold, we also require less packets to detect possible elephant flows. Additionally, Figure 9b shows that we increase memory efficiency with a larger packet length threshold as the number of bits required for each flow decreases. However, Figure 9c shows that increasing the packet length threshold also decreases the filtering accuracy, i.e., flows incorrectly reported as possible elephant flows and forwarded to the next step, the `Refined Detection` module. Therefore, with different parameters, we can choose a trade-off between detection speed, memory efficiency, and detection accuracy.

VI. Related Work

Several network load balancing systems have been proposed in the literature. Table II compares the related work, highlighting some of their main characteristics, such as the plane responsible for the routing decision and the technique employed to generate paths.

Firstly, ***data plane load balancers*** rely exclusively on data plane processing to implement their routing strategy. Due to the limitations of the programmable hardware, these systems typically employ simple heuristics to *generate routes*. Generally speaking, heuristic-based route generation (and *selection*) can lead to suboptimal network utilization. Further, per-hop (decentralized) path selection [9], [10], [26] can lead to worse routes than fine-grained, end-to-end (centralized) path selection [8], [11]. Finally, some data plane load balancers are limited to datacenter *topologies* [8], [9], [26].

[8]We configured the threshold to this value after an analysis based on our workloads and parameters. We expect network administrators to select appropriate parameters based on knowledge of their network.

Secondly, ***control plane load balancers*** implement a variety of *path generation* and *path selection* strategies. Path generation based on heuristics may lead to suboptimal network utilization when compared to sophisticated machine learning strategies. Fine-grained end-to-end path selection [15], [27], [28] may lead to better routes at the cost of greatly reduced responsiveness to transient congestion and scalability due to control plane involvement. On the other hand, load balancers that implement link weights (WCMP) path selection [3]–[7], [31] are generally more scalable, as the control plane is not included in the path selection of each flow. However, WCMP requires control plane intervention to change link weights, which limits responsiveness to transient congestion.

Thirdly, ***end-host load balancers*** [29], [30] are highly scalable and responsive to transient congestion, as each host is only responsible for its own flows. However, decentralized (and often heuristic-based) *path generation* and *selection* can lead to suboptimal link utilization. Further, these types of load balancers require modifying end-hosts, which limits deployability to specific cases, such as datacenters or cloud.

Finally, an emerging class of ***hybrid load balancers*** combine reactive data plane processing, enabling high responsiveness to transient congestion, with superior *path generation* by the control plane, leading to efficient network utilization. However, we believe existing work can be improved upon, as current strategies are limited to heuristic-based *path generation* and *selection* [25]. By employing a Deep Reinforcement Learning agent, CrossBal can select the best routes for each rerouted flow. Further, by focusing its efforts on elephant flows, CrossBal minimizes the number of flows to be actively rerouted. Finally, the fast decision loop in the data plane can quickly react to transient congestion on installed paths.

VII. Conclusion

We have presented CrossBal, a hybrid load balancer that combines an intelligent decision loop based on a Deep Reinforcement Learning agent in the control plane, with a reactive decision loop in the programmable data plane. We highlighted key aspects of the modelling of the agent, comparing the performance of CrossBal with different reward functions. Our evaluation shows that CrossBal outperforms ECMP at balancing the workload over available network links. Finally, of the related work highlighted, only a few [15], [27] focus their efforts on elephant flows. As elephant flows are large and long-lasting flows that tend to have a high impact on the

TABLE II: Comparison of related work

Decision Plane	Path Generation	Path Selection	Examples	Benefits	Limitations
Data Plane	Heuristics	Per-hop	HULA [9], LetFlow [10] BurstBalancer [26]	Data plane decision-making and per-hop selection lead to high scalability and responsiveness.	Heuristic-based path generation and per-hop selection can lead to suboptimal network utilization.
		Fine-grained	CONTRA [11] CONGA [8]	Data plane decision-making leads to high responsiveness.	Path generation and selection is based on heuristics, while fine-grained path selection limits scalability.
Control Plane	Heuristics	Fine-grained	Hedera [15], Mahout [27] Chameleon [28]	Controller has a global view of the network, leading to better path selection.	Control Plane involvement compromises scalability and responsiveness. Heuristic-based path generation and selection.
		Link Weights	Le at al. [3], DOTE [4] Magnouche et al. [6]	Path selection based on link weights is highly scalable.	Path selection based on WCMP can lead to suboptimal network utilization. Controller involvement limits responsiveness.
	Machine Learning	Link Weights	DRL-TE [5] Valadarsky et al. [7]	Machine Learning-based approach can lead to better link weights.	Path selection based on WCMP can lead to suboptimal network utilization. Controller involvement limits responsiveness.
End Host	Heuristics	Fine-grained	Hermes [29] PLB [30]	End-host decision-making is highly scalable and generally responsive.	Requires modifying end-hosts, severely limiting deployability. Heuristic-based approach can lead to suboptimal path selection.
Hybrid	Heuristics	Fine-grained	Pizzutti et al. [25]	Combines data plane responsiveness with control plane visibility and path generation.	Heuristic-based path generation with controller involvement, limiting scalability and causing suboptimal network utilization.

network, focusing on these flows can improve control plane scalability, as there are significantly fewer flows to reroute.

Acknowledgments

This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq (grant #311276/2021-0) and FAPESP (grant #2020/05152-7 - PROFISSA).

References

[1] V. Gavriluţ, A. Pruski, and M. S. Berger, "Constructive or optimized: An overview of strategies to design networks for time-critical applications," *ACM Comput. Surv.*, vol. 55, no. 3, feb 2022.

[2] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu, "Load balancing in data center networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2324–2352, 2018.

[3] V. A. Le, T. T. Le, P. L. Nguyen, H. T. T. Binh, and Y. Ji, "Multi-time-step segment routing based traffic engineering leveraging traffic prediction," in *IM '21*, 2021, pp. 125–133.

[4] Y. Perry, F. V. Frujeri, C. Hoch, S. Kandula, I. Menache, M. Schapira, and A. Tamar, "DOTE: Rethinking (predictive) WAN traffic engineering," in *NSDI '23*. USENIX, Apr. 2023, pp. 1557–1581.

[5] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM 2018*. IEEE, 2018, p. 1871–1879.

[6] Y. Magnouche, P. T. A. Quang, J. Leguay, X. Gong, and F. Zeng, "Distributed utility maximization from the edge in ip networks," in *IM '21*, 2021, pp. 224–232.

[7] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route," in *HotNets-XVI*. ACM, 2017, p. 185–191.

[8] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *SIGCOMM '14*. ACM, 2014, p. 503–514.

[9] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *SOSR '16*. ACM, 2016.

[10] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *NSDI '17*. USENIX, Mar. 2017, pp. 407–420.

[11] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Contra: A programmable system for performance-aware routing," in *NSDI '20*. USENIX, Feb. 2020, pp. 701–721.

[12] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *SIGCOMM '11*. ACM, 2011, p. 254–265.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *SIGCOMM '10*. ACM, 2010, p. 63–74.

[14] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *SIGCOMM '09*. ACM, 2009, p. 51–62.

[15] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *NSDI'10*. USENIX, 2010, p. 19.

[16] P. Jurkiewicz, "Boundaries of flow table usage reduction algorithms based on elephant flow detection," in *IFIP Networking '21*. IEEE, 2021, pp. 1–9.

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014.

[18] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *HotNets-XVI*. ACM, 2017, p. 150–156.

[19] Y. Zhan and J. Zhang, "An incentive mechanism design for efficient edge learning by deep reinforcement learning approach," in *IEEE INFOCOM 2020*. IEEE, 2020, pp. 2489–2498.

[20] F. Restuccia and T. Melodia, "Deepwierl: Bringing deep reinforcement learning to the internet of self-adaptive things," in *IEEE INFOCOM 2020*. IEEE, 2020, pp. 844–853.

[21] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira, "Verifying learning-augmented systems," in *SIGCOMM '21*. ACM, 2021, p. 305–318.

[22] H. Liu, C. Jin, B. Yang, and A. Zhou, "Finding top-k shortest paths with diversity," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 488–502, 2018.

[23] K. Rusek, J. Suárez-Varela, A. Mestres, P. Barlet-Ros, and A. Cabellos-Aparicio, "Unveiling the potential of graph neural networks for network modeling and optimization in sdn," in *SOSR '19*. ACM, 2019, p. 140–151.

[24] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh, "Juggler: A practical reordering resilient network stack for datacenters," in *EuroSys '16*. ACM, 2016.

[25] M. Pizzutti and A. E. Schaeffer-Filho, "Adaptive multipath routing based on hybrid data and control plane operation," in *IEEE INFOCOM 2019*. IEEE, 2019, p. 730–738.

[26] Z. Liu, Y. Zhao, Z. Fan, T. Yang, X. Li, R. Zhang, K. Yang, Z. Zhong, Y. Huang, C. Liu, J. Hu, G. Xie, and B. Cui, "Burstbalancer: Do less, better balance for large-scale data center traffic," in *ICNP '22*, 2022, pp. 1–13.

[27] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *IEEE INFOCOM 2011*. IEEE, 2011, pp. 1629–1637.

[28] A. Van Bemten, N. Ðerić, A. Varasteh, S. Schmid, C. Mas-Machuca, A. Blenk, and W. Kellerer, "Chameleon: Predictable latency and high utilization with queue-aware and adaptive source routing," in *CoNEXT '20*. ACM, 2020, p. 451–465.

[29] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *SIGCOMM '17*. ACM, 2017, p. 253–266.

[30] M. A. Qureshi, Y. Cheng, Q. Yin, Q. Fu, G. Kumar, M. Moshref, J. Yan, V. Jacobson, D. Wetherall, and A. Kabbani, "Plb: Congestion signals are simple and effective for network load balancing," in *SIGCOMM '22*. ACM, 2022, p. 207–218.

[31] M. Parham, T. Fenz, N. Süss, K.-T. Foerster, and S. Schmid, "Traffic engineering with joint link weight and segment optimization," in *CoNEXT '21*. ACM, 2021, p. 313–327.