Reinforcement Learning-based In-network Load Balancing

Hesam Tajbakhsh[†] [‡] , Ricardo Parizotto[*], Alberto Schaeffer-Filho[*], Israat Haque[†]
[†]*Dalhousie University, Canada*, [*]*UFRGS, Brazil*

***Abstract*—Ensuring consistent performance becomes increasingly challenging with the growing complexity of applications in data centers. This is where load balancing emerges as a vital component. A load balancer distributes network or application traffic across various servers, resources, or pathways. In this article, we present P4WISE, a load balancer designed for software-defined networks. Operating on both the data and control planes, it employs reinforcement learning to distribute computational loads with granularity at inter and intra-server levels. Evaluation results demonstrate a remarkable 90% accuracy in predicting the optimal load balancing strategy of P4WISE in dynamic scenarios. Notably, unlike supervised or unsupervised methods, it eliminates the need for retraining when the environment undergoes minor or major changes. Instead, P4WISE autonomously adjusts and retrains itself based on observed states within the data center.**

***Index Terms*—Load Balancing, Data Plane Programmability, Reinforcement Learning.**

I. INTRODUCTION

MAINTAINING a consistently reliable performance poses a considerable challenge in data centers (DCs), where many servers execute applications that are becoming more intricate and widely used. This is where load balancing comes into play as a crucial strategy. A Load Balancer (LB) distributes network or application traffic among multiple servers, resources, or paths [1], [2]. The primary objective is to judiciously utilize resources, enhance fault tolerance, and, most importantly, boost the overall performance and responsiveness of applications. To achieve these objectives, both academia and industry have explored diverse types of LBs that operate at different layers of the network stack, including L4 [3], [4], and L7 [5], [6], and that are deployed as either software or hardware solutions [7]. With the recent advancement of programmable switches, a growing trend is to delegate load-balancing tasks to the data plane [3], [8], [9], [4], [10], [11]. An in-network LB functions directly within the network infrastructure of a data center, efficiently distributing incoming traffic across multiple servers or paths. An in-network LB implemented on a programmable switch offers distinct advantages, such as freeing server resources for applications and also the ability of processing packets in real-time at line rate (Gbps).

Existing in-network LBs deploy established algorithms such as equal-cost multi-path (ECMP) and join the shortest queue (JSQ) [12] at the inter-server granularity in DCs. With Moore's law reaching a plateau, modern servers are now outfitted with application-specific accelerators such as SmartNICs and GPUs rather than relying solely on adding extra CPU cores to enhance application performance [13], [14]. Consequently, these LBs struggle to deliver promising results since they overlook accelerator architectures and their associated constraints. P4MITE [10] and P4HAULER [11] are accelerator-aware LBs designed to operate on programmable switches. After demonstrating that accelerators such as SmartNICs can quickly become overwhelmed without proper load balancing as requests may experience delays or get dropped, both research efforts introduce mechanisms to collect information from the servers and their accelerators and allocate each to process the load accordingly.

Nevertheless, existing research efforts are not tailored to accommodate changes within DCs. Information Technology (IT) companies maintain an ongoing infrastructure refresh cycle. For instance, Meta and Google continuously update their servers, including their accelerators and other components, ensuring consistent upgrades without prolonged gaps [15]. Additionally, changes in applications and their traffic rates occur more frequently. Utilizing configurable LBs like P4HAULER [11] necessitates constant and rapid configuration adjustments, which can be laborious. To address this, machine learning (ML) shows a promising solution adopted by certain LBs like LBAS [4] and LLB [16] for automating the prediction of suitable load-balancing policies. However, limitations within the data plane and the packet-processing architecture (such as memory constraints, the number of stages in the processing pipeline, and ALU operations) prevent the implementation of complex operations (e.g., floating point operations) required by ML algorithms.

Consequently, most existing LBs that utilize ML resort to employing simpler models, which are not suitable for scenarios involving significant changes [4], [16]. Furthermore, as they rely on supervised learning, these LBs necessitate collecting new data, model retraining, and deploying the updated model into the system upon any alterations, which often requires reconfiguring the entire switch pipeline. We should also mention that QCMP [17] uses Reinforcement Learning (RL) for load-balancing, but it is limited to balancing the load among multiple paths if the traffic pattern changes, and it does not support resource allocation.

This article presents P4WISE, an in-network accelerator-aware LB that uses RL to determine the optimal load balancing policy. It allocates resources for load processing at a fine (accelerator) level of granularity, similarly to P4MITE [10] and P4HAULER [11]. In addition, P4WISE employs RL to monitor the environment and identify suitable load-

[‡] Hesam Tajbakhsh is the corresponding author.

balancing policies. At the same time, P4WISE must deal with the resource constraints and operational limitations of the data plane and apply a hybrid (control and data plane) load balancing mechanism. Specifically, we proactively deploy multiple load balancing policies in the switch data plane to minimize the communication between the data and control planes while applying the new policy for load balancing. When a change occurs, if the switch identifies an appropriate policy based on the current utilization in the data plane, it immediately applies it to the available LB module, e.g., a dynamic round robin scheduler, in the data plane. On the other hand, P4WISE performs RL training and RL policy computation asynchronously in the control plane. Put differently, while the switch processes the packets according to the configured LB policies, it periodically receives network condition updates, including the statuses of various computation resources (e.g., CPU, GPU, and SmartNIC). If the status changes to a known condition, the switch activates one of the deployed LB policies. Otherwise, it requests the control plane to find the appropriate one. Thus, P4WISE controller iteratively updates the load balancing policies in the data plane, adapting them based on network states until the optimal configuration is reached.

To assess P4WISE, we integrate our solution into an existing simulator [18], which simulates ML inference traffic. Additionally, we devised a proof of concept prototype for P4WISE on the Tofino programmable switch [19]. All of our implementation is also publicly available on a GitHub repository [20]. Our comprehensive evaluation reveals that P4WISE can identify the optimal policy with 90% accuracy even under worst-case conditions. Moreover, we compare P4WISE against a supervised learning approach that necessitates a dataset for training. Our findings demonstrate that while the supervised approach fails when configurations change in the data center, P4WISE dynamically adjusts to system alterations, consistently identifying the most effective LB policy in modified environments.

In summary, our contributions are the following:

1) We introduce P4WISE, the first in-network load balancer that employs RL for load balancing at the accelerator granularity.
2) We implement a prototype of P4WISE on a programmable Tofino switch and in a simulator [21] to evaluate and compare its performance against existing works.
3) The evaluation results establish the superiority of P4WISE over the current state-of-the-art in selecting the optimal load-balancing policy, particularly in scenarios where system configurations undergo changes.

II. Background and Motivation

A. In-Network Load Balancing

Programmable switches allow dynamically changing the behavior of programmable data plane devices. The switch functionality is often expressed using domain-specific languages, such as P4 [22], and compiled into a packet processing architecture. A concrete example of packet processing architecture is the Protocol Independent Switch Architecture (PISA), which generalizes the Reconfigurable Match-Table (RMT) [23] model.

In-network load balancing capitalizes on the flexibility of RMT to balance traffic in the data plane at line rate. The LB splits connections between instances of applications running on servers. Switches use a load balancing policy to choose a destination for a new connection, also keeping a state about the decisions taken for individual connections. Further, subsequent packets from an established connection are forwarded to the same destination, maintaining per-connection consistency. By allowing those operations to occur at line rate, in-network load balancing offers advantages compared to server-only load balancers, reducing latency and improving throughput for clients' applications. However, existing load balancers can not either adapt to changes quickly [11] or use the benefits from applications running on accelerators, e.g., GPU or SmartNICs [3], [24], [25], [4], [16].

B. Reinforcement Learning

Reinforcement Learning (RL) is a subfield of artificial intelligence that refines its decisions through interactions with the environment. An RL agent interacts with its environment, observing the current state and choosing *actions* based on a *policy*[1]. Several well-known policies are commonly used for reinforcement learning agents, including, but not limited to, random, greedy, and ϵ-greedy strategies. Under a random policy, the agent selects actions uniformly at random, which ensures maximum exploration but disregards any learned knowledge. In contrast, a greedy policy directs the agent to always choose the action with the highest estimated value, fully exploiting its current knowledge. However, this can lead to suboptimal results if the agent fails to explore potentially better actions. The ϵ-greedy policy offers a balance between these two approaches: with probability $1-\epsilon$, the agent selects the best-known (greedy) action, while with probability ϵ, it chooses a random action, allowing for controlled exploration while still leveraging past learning.

After taking an action by the selected policy, the agent receives a *reward* from the environment. The future decisions of the agent change based on the reward to select another action in similar situations in the future, aiming to maximize its cumulative rewards over time [26]. RL algorithms can be model-based or model-free [27]. Model-based RL is an approach where the agent learns a model or representation of the environment's dynamics. This model is used to simulate and predict how the environment will respond to different actions taken by the agent. In other words, the agent builds an internal model of the world to estimate the consequences of its actions. In contrast, model-free reinforcement learning comes into play when the environment cannot be entirely modeled or predicted. A few notable examples of model-free algorithms are Q-Learning [28], Deep Q-Network DQN [29], and SARSA [30].

Furthermore, we can categorize model-free algorithms into off-policy and on-policy types. In off-policy algorithms, the

[1]This policy refers the one RL agent employs for computation, which is different from load balancing policies in the switch.

agent improves the target policy while interacting with the environment using a different behavior policy to generate actions. Examples of off-policy algorithms include Q-Learning [28] and Deep Q-Network (DQN) [29]. For instance, during a Q-Learning [28] episode, the agent might use a random, ϵ-greedy, or greedy policy to interact with the environment, then update the Q-Table using a greedy approach. On the other hand, in on-policy algorithms, the target policy and behavior policy are the same. The agent continuously updates and improves the policy being used to interact with the environment. An example of an on-policy RL algorithm is SARSA [30].

In this research, we utilize the Q-Learning algorithm supported by a Q-Table. The Q-Table serves as a lookup structure that holds the estimated Q-Values for every state-action pair, indicating the expected future reward of taking a particular action in a specific state. This enables the agent to make decisions by selecting the action with the highest Q-Value in its current state. After each action (a), the state s of the environment changes to s' and the agent updates the corresponding Q-Value using the Bellman Equation [28], [31], refining its estimates to improve decision-making over time:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \quad (1)$$

In Equation 1, α and γ denote the `learning rate` and the `discount rate`, respectively, and have a value from 0 to 1. α determines the size of the adjustments made to the model's parameters at each step. γ is a parameter that represents the preference for current rewards over future rewards. If γ is close to 0, the agent cares mostly about immediate rewards and is short-sighted. Yet, if γ is close to 1, the agent values future rewards more heavily, making it far-sighted and encouraging long-term planning.

C. Why RL-based Load Balancing?

Machine learning offers significant benefits when addressing issues involving the handling of extensive datasets, predicting outcomes, or making decisions, especially in situations where explicitly programming a solution is difficult due to the complexity or variability of the data. Among ML techniques, supervised and unsupervised learning techniques are unsuitable for load-balancing decisions in scenarios with many changes since those techniques would require a comprehensive dataset encompassing all possible combinations, including the number of servers, accelerators, their capacities, and so forth, which would be an infeasible task.

Thus, we conduct initial experiments to justify our claim and position the need for a suitable learning-based load balancer design. In the first experiment, we deploy 128 identical servers, each equipped with one SmartNIC. In the second experiment, we upgrade the SmartNICs to a version that is 20% more powerful. In the third experiment, we add an additional SmartNIC, identical to the original one, to each server. The computing power of each resource is defined by the number of floating-point operations per second (FLOP/s). We choose these values based on measurements from our testbed: 91 GFLOP/s for the servers and 17.6 GFLOP/s for the SmartNICs. We also adopted the same load balancing policies as in [11], as they extensively evaluated various policies in-network. Specifically, their results demonstrated that the weighted round-robin with weights of 5:0 or 5:1 would yield promising results as the server processor is five times more powerful than the SmartNICs.

We gather various configuration and performance metrics like input load (requests per second), weights to resources, and resource utilization (e.g., CPU utilization) from all three experiments. We collect 480 samples for the first experiment of identical servers with a SmartNIC, which is enough to train the considered multi-layer perceptron neural network with three layers, each comprising ten nodes. In other words, the model converges with this amount of data. The trained models achieve an accuracy of 97% in predicting the optimal load-balancing policy. The trained model is then applied to the second and third experiments, i.e., upgrading the SmartNICs with 20% more capacity and adding an additional SmartNIC to each server, respectively. Substituting the SmartNICs with ones that are 20% more powerful drops accuracy to 80%, and adding the second SmartNIC to each server further reduces the accuracy to 66%. This decline in model performance is attributed to the model being trained on a different configuration, indicating that changes in system configuration necessitate retraining or adjustment of the model.

Specifically, the above observation and challenges motivate us to employ reinforcement learning to tackle this load-balancing problem. Unlike supervised and unsupervised learning, RL models do not require pre-training on all conceivable inputs. Instead, they can adapt and refine their parameters through trial-and-error procedures within the system. Furthermore, in response to system changes, RL models can dynamically adjust their parameters to accommodate such alterations.

III. P4WISE DESIGN

This section presents P4WISE, a system that employs Reinforcement Learning (RL) in programmable switches to efficiently balance load across servers and their respective accelerators. Our design is based on the following principles:

Load balancing using Dynamic Weighted Round-Robin (DWRR). Due to data plane constraints, such as the lack of floating-point operations and the limited number of pipeline stages, deploying complex load-balancing mechanisms is challenging. To address this, we leverage WRR as a practical yet flexible solution. By dynamically changing WRR weights (DWRR) in the data plane, we emulate various load-balancing behaviors (e.g., device prioritization) in a hardware-friendly manner. This approach adapts to real-time computing resource states, enabling near-optimal traffic distribution without violating data plane limitations [4], [16].

Asynchronous RL model. Storing all possible states and their corresponding weights directly in the data plane would impose a significant memory overhead, which is impractical given the limited hardware resources of network switches. Conversely, relying on the control plane to provide updated weights for every state change introduces latency, resulting in sub-optimal

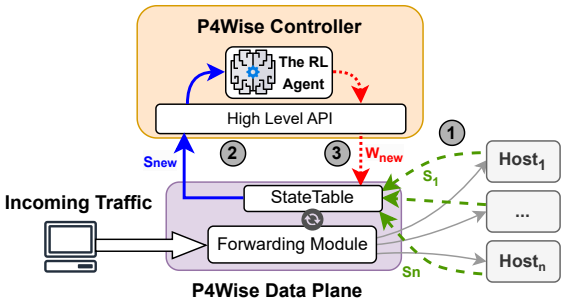


Fig. 1: P4WISE overview.

load balancing. To address this trade-off, the RL, located in the control plane, proactively offloads a set of pre-computed weights to the data plane, allowing it to respond promptly to resource changes. When an unrecognized state is encountered, the data plane notifies the control plane. The RL agent either applies an existing optimal weight configuration immediately or temporarily applies the closest matching set of weights from the trained options and simultaneously utilizes Q-Learning to iteratively explore and update weights in the data plane until an optimal set of weights is found.

In the subsequent sections, we provide detailed explanations of various components within our RL model. To simplify the discussion, we operate under the assumption that our system comprises multiple servers with equivalent processing capabilities. Each of these servers is equipped with a SmartNIC, and all the SmartNICs possess identical processing capacities. It is worth noting that the server capacities differ from those of the SmartNICs. We should also mention that the homogeneity at different levels, like having identical servers or SmartNICs, is prevalent due to ease of management [32]. In this case, P4WISE can be deployed into Top Of Rack (TOR) switches to perform load balancing for all servers and their accelerators in the same rack.

A. P4WISE *Overview*

Figure 1 provides an overview of P4WISE. To enhance clarity, we have omitted the depiction of accelerators for the servers (hosts) on the right side of the figure. Each server can have one or multiple accelerators (e.g., SmartNICs, GPUs, and TPUs). Furthermore, agents are running on the servers and their accelerators, responsible for monitoring and transmitting updates to the switch periodically[2].

The switch in P4WISE, positioned at the center of Figure 1, is composed of two primary components: the Forwarding Module and the StateTable. The Forwarding Module integrates several subcomponents responsible for tasks such as packet processing and load balancing. Packet processing is handled through elements like Match-Action tables and Bloom filters, as described in prior work [11], [8], [10]. Load balancing is managed by additional submodule that implements a weighted round-robin policy, building on the techniques introduced in our earlier research [11]. In summary, using this module, we can assign a weight to each device, and the switch splits and forwards the load to devices accroding the weights.

While the hosts and their accelerators process clients' requests, the monitoring agents continuously measure and transmit the current resource states to P4WISE's switch, as indicated by the dashed green lines in Figure 1 (Step 1). Within the data plane, switches assess whether the reported states from the monitoring agents match any previously observed status in *StateTable*. If a match is found, the data plane configures its blocks accordingly. Otherwise, it forwards the new states to the controller, which is responsible for executing the RL agent, also referred to as P4WISE's agent. Section III-B provides more details about the data plane.

The controller, on the other hand, relies on a set of high-level APIs for communication with the data plane. The specific used APIs may vary depending on the switches in use. The RL agent uses these APIs to retrieve information from the data plane, shown in solid blue arrows in Figure 1, Step 2. Initially, the RL agent acquires any required information from the data plane. We will explain this information in Section III-C. Using this information, the RL agent then calculates the appropriate weights (W_{new}) using Q-Learning and applies them to the switch, depicted in red dotted lines in Figure 1, Step 3.

B. P4WISE*'s Data Plane*

Figure 2 depicts the data plane configuration of P4WISE. In the following, we elaborate on each functional component.

Connection Packet Handling. The switch identifies its type after receiving a new packet. If the packet is a connection packet, it undergoes processing along the upper pathway (illustrated by the green solid arrows). Specifically, the switch verifies whether the packet corresponds to an existing connection using a bloom filter. If the connection is already established, the switch retrieves the previously designated destination (`DIP`) from the *ConnTable*. Conversely, if it is a new connection, the switch initially selects the destination server (`SID`) from the *ServerTable*. Subsequently, it determines the `DIP` based on the active weights in the Dynamic Weighted Round Robin (DWRR) module.

Update from Resource Monitoring Agents. When a packet from the agents monitoring resources arrives (shown as blue dashed lines), the switch verifies if the current state matches an observed state in *StateTable*. More specifically, *StateTable* acts like a cache to store previously observed states. If it is an unseen state (i.e. cache miss), the switch forwards the packet to the P4WISE agent in the controller to determine the proper weights. Conversely, if the state is already observed in the table (i.e. cache hit), the appropriate weights are promptly copied to the DWRR module to activate the policy corresponding to the weights for subsequent packets.

Update from the Controller. When the RL agent residing in P4WISE's controller is asked by the data plane for an unknown state, the agent computes new weights using Q-Learning (described in detail in the following subsection)

[2]The agents monitoring the resources' status should not be mixed up with the P4WISE agent in the control plane. The first ones do the measurements and send them to the switch, while the latter is the RL agent.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2025.3621126

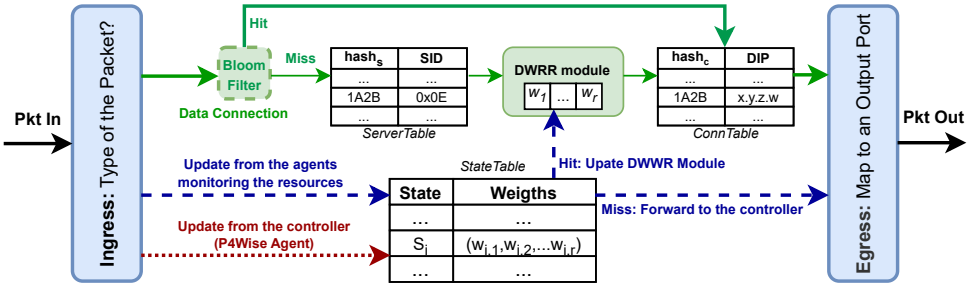


Fig. 2: P4WISE's Data Plane Layout.

and sends them back to the data plane, where the switch updates the *StateTable* (indicated by the red dotted line). If the optimal weight is determined after a single iteration, it means that the RL agent has sufficient knowledge of the state and environment. Otherwise, the agent continues adjusting the weights in the data plane and monitoring the environment until the optimal value is found. The primary goal is to minimize latency and maximize rewards. However, the agent does not measure latency directly. Instead, it infers performance based on resource metrics that most significantly impact latency; for instance, it monitors CPU utilization for CPU-intensive applications. This strategy avoids the delays associated with waiting for end-to-end latency measurements, allowing the agent to respond more promptly by continuously observing resource usage patterns.

C. RL Model Formulation

This section outlines the components of the RL model developed for the P4WISE system, which leverages Q-Learning decision making.

State Space (S): While the RL agent has access to various environmental features, incorporating too many of them can significantly increase the complexity of decision-making. To ensure efficiency, it is crucial to define a concise yet informative state space that accurately represents the environment. As discussed in Section II-C and prior work [10], [11], the agent must decide when to hire or release accelerators based on system needs. Therefore, one essential parameter in the state is whether each accelerator is currently active. Additionally, decisions are made based on the utilization levels of the servers and their active accelerators. We aggregate this information by computing the average utilization across all devices currently in use— servers plus only accelerators that are active. Finally, the RL agent requires knowledge of the current load distribution weights, which it may adjust as necessary. With these components, the state representation of the system is defined as follows:

$$S = \{U_{avg}, W\} \tag{2}$$

In the defined state representation, U_{avg} indicates the average utilization (in percentage) of active components (including the server and any engaged accelerators). The vector W specifies the load distribution weights for the server and accelerators, determining how incoming tasks are allocated. The agent infers whether accelerators are in use by checking the assigned weights in the vector W. Figure 3 illustrates the Q-Table required for Q-Learning. The possible range of average utilization is divided into n intervals, each of width $\frac{100}{n}$. For each interval, the system can either be using accelerators or not, resulting in $2 \times n$ distinct states. Increasing n improves granularity but enlarges the Q-Table. Lastly, all Q-Values are set to 0.

State		Q-Values		
Utilization Range	Are the accelerators in use?			
R_1	No	0	0	0
R_2	No	0	0	0
⋮	⋮		⋮	
R_n	No	0	0	0
R_1	Yes	0	0	0
R_2	Yes	0	0	0
⋮	⋮		⋮	
R_n	Yes	0	0	0

Fig. 3: Initialized Q-Table.

Action Space: When the RL agent takes action, it stores weights on the switch that are used to distribute the workload across all devices, encompassing servers and their accelerators. The agent has the option to either generate weights for each end-host individually or incrementally update the weights until reaching the optimal configuration. The first option involves using a weight vector that enlarges the action space, resulting in an extended training period. Nevertheless, we mitigate this issue by constraining the action space to three actions: 1) incrementing accelerators' weights by one, 2) decrementing accelerators' weights by one, and 3) maintaining the current weights. With this limited set of actions, the agent retains the ability to apply a diverse range of weights to the system without exploring a huge space of options during training.

Reward Function: We created a reward function that guides the RL agent towards minimizing latency. Algorithm 1 describes the procedure for computing the reward. Commencing at line 1, it involves retrieving the current state. Subsequently, with the utilization of resources and their corresponding weights, the agent computes the 99th percentile latency at line 2 and the average utilization of the hosts at line 3. Note that this methodology is employed in the simulator; in an actual implementation, agents operating on servers

and accelerators can measure the 99th percentile latency and convey it, along with other metrics, to the switch.

After completing the outlined computations, the algorithm evaluates three conditions to ascertain if the agent has reached a terminal state. In a terminal state, the agent has either chosen an inappropriate set of weights, leading to system failure or discovered the optimal weights that exhibit the best performance. The first `if` statement at line 4, which checks if utilization has surpassed 100%, denotes the identification of an unsuitable set of weights, suggesting a potential system failure. Conversely, the subsequent two `if` statements at lines 7 and 10 scrutinize the selection of optimal weights.

The `if` statement at line 7 assesses whether optimal latency is achieved by exclusively utilizing the servers. Conversely, at line 10, it determines whether the server—being the more potent resources—are not underutilized, when all resources are in use. This assessment is influenced by our findings that show prioritizing servers is necessary if they possessed sufficient capacity to manage the load. We should also note that `latency_threshold` and `utilization_threshold` vary depending on the application running in the data center, and we discuss how to determine them in Section V-A.

Finally, at the end of the reward calculation, if none of the conditions are triggered, it indicates that the agent has not reached a terminal state and it should continue its operation.

Algorithm 1 Reward Function

1: $U, W \leftarrow read\_current\_state()$
2: $p99\_lat \leftarrow calculate\_lat(U, W)$
3: $avg\_util\_servers \leftarrow calculate\_util\_servers(U)$
4: **if** $max(U) > 100$ **then**
5: **return** $-(max\_reward)$
6: **end if**
7: **if** $only\_cpus(U)$ & $p99\_lat < latency\_threshold$ **then**
8: **return** max_reward
9: **end if**
10: **if** $all\_devices(U)$ & $p99\_lat < latency\_threshold$ & $avg\_util\_servers > utilization\_threshold$ **then**
11: **return** max_reward
12: **end if**
13: **return** -1

D. Load Balancing with P4WISE

Algorithm 2 outlines the load balancing process undertaken by P4WISE. The Q-Table (QT) and weights (W) are initialized in the control plane and data plane, respectively (lines 1 and 2). Subsequently, the algorithm monitors the environment in the data plane (line 3). The system's load-balancing procedure is triggered upon detecting a change in the system's state, assessed through utilization monitoring (line 4). If the state is an observed state, the switch pulls the appropriate weights from the *StateTable* in the data plane (line 6). This table functions as a cache to store the appropriate weights of previously seen states.

Otherwise, if needed, the P4WISE agent executes the second `while` loop (line 8), continuing until a terminal state is reached, characterized by either achieving the optimal weight

Algorithm 2 Load Balancing with P4WISE.
DP and CP stands as Data Plane and Control Plane.

1: $initialize\ QT$ ▷ CP
2: $initialize\ W$ ▷ DP
3: **while** $True$ **do**
4: **if** $change\_state()$ & $reward(U,\ W) < 0$ **then** ▷ DP
5: **if** $observed\_state()$ **then** ▷ DP
6: $read\_from\_StateTable()$
7: **else**
8: **while** ! $final\_state(U, W)$ **do** ▷ CP
9: $action = \pi(U,\ W)$
10: $W\ =\ update\_weight(W,\ action)$
11: $U',\ W'\ =\ apply\_the\_weights(W)$
12: $r = reward(U',\ W')$
13: $Update\_QT()$
14: $U,\ W = U',\ W'$
15: **end while**
16: $update\_StateTable()$ ▷ DP
17: **end if**
18: **end if**
19: **end while**

or an incorrect weight completely in the control plane. Within this second loop, P4WISE agent initially selects an `action` using an ϵ-greedy policy π (line 9). Subsequently, the new weights corresponding to the chosen action are computed (line 10). These updated weights are then applied to the system, and the resulting new states (U', W') are obtained from the environment (line 11). The agent proceeds to compute the reward in the new state (line 12). Finally, at line 13, the agent updates the Q-Value of the taken action using the Bellman Equation [28], [31] to optimize the Q-Values. We will study the appropriate α and γ in Section V-A. Finally, if the agent reaches the optimal weights, it updates *StateTable* (line 16) to use them in the future if required.

IV. EXPERIMENTAL SETUP

We implement a prototype of P4WISE and evaluate it over a testbed and a simulator to demonstrate its performance and scalability.

Prototype Implementation: The implementations of P4WISE's switch and controller are based on the prior accelerator-aware load balancer, P4HAULER [11], [18], which are written in $\sim$ 400 lines of P4-16 and 150 lines of Python3, respectively. We also map our P4 application to the Tofino Native Architecture (TNA). In particular, we utilize the switch's available registers for *ConnTable*, *ServerTable*, and *StateTable*. Additionally, we construct the *DWWR* module using the programmable blocks from [11]. When the switch receives new status information from the resources, it forwards the packet to P4WISE's resource monitoring agent. The agent then calculates the best weights and employs Tofino APIs to update the *StateTable* for future operations. Furthermore, monitoring scripts running on the resources (i.e., the servers and the SmartNICs) are written in approximately 250 lines of Python3 code and utilize the *psutil* library to measure resources' usage.

Simulator: We have developed a proof of concept (PoC) prototype for the P4WISE controller in Python, consisting of approximately 100 lines of code. This prototype is accessible in [21]. Our PoC is evaluated over a realistic simulator proposed in [11]. Specifically, It simulates the processing of each computing resource (CPUs and SmartNICs) and their interactions based on the specifications of a real testbed. Then, the performance of this simulator is compared against that testbed for machine learning inference to ensure the dependability of the chosen simulator. Using this simulator, we deploy 64, 128, or 256 servers, each with two SmartNICs (unless otherwise stated). We demonstrate the scalability and effectiveness of P4WISE over this simulator in Sections V. We should also mention that this simulator is our principal platform for our assessments, and we always adopt it unless stated otherwise.

Testbed: The testbed comprises two servers connected by a Wedge 100BF-32X 32-port programmable switch equipped with a 3.2Tbps Tofino ASIC [19]. One serves as the client while the other functions as the server; each hosts a SoC SmartNIC. The server hardware is equipped with an Intel(R) Xeon(R) Silver 4210R CPU at 2.4GHz, featuring 10 cores and 32GB of memory. The SmartNIC is a dual-port SFP28 device, operating on PCIe Gen3.0/4.0 x8 from the BlueField(R) G-Series, boasting 16 cores, 16GB of on-board DDR4 RAM, and enabled crypto accelerators. The deployment feasibility of P4WISE is measured over this testbed.

Applications: To evaluate P4WISE, we generate inference workloads for two trained machine learning applications. The first application employs KNN [33] for unsupervised image classification on the MNIST dataset [34]. The Second application uses BM25 [35], a natural language processing model that works on the SciFact dataset [36]. We run each application individually and let P4WISE handle the transmitted requests. Moreover, since KNN is more computationally intensive, it is considered as the application in the simulation and on the testbed, unless stated otherwise. Finally, we set up monitoring agents on both the server and the accelerators to transmit measured data every $10ms$, following the methodology described in [11].

Load Balancing Policies: To compare P4WISE with the state-of-the-art, we deploy two algorithms, priority (PRT) and Weighted Round Robin (WRR), on the testbed to distribute the inference load of the two mentioned machine learning applications. PRT prioritizes the most powerful device, and once the most powerful one is fully utilized, PRT employs other resources. On the other hand, WRR distributes the load according to the computing capacity of each resource, e.g., the server CPU is five times more powerful than the smartNIC CPU [11].

V. P4WISE'S EVALUATION

This section starts with a series of experiments to find the most effective parameters to tune our model in subsection V-A. We need to establish key parameters such as α and γ, along with the thresholds for latency and utilization. Once we identify the optimal parameters, we move on to evaluate the trained model of P4WISE at different scales in comparison to P4HAULER in subsection V-B. Finally, in subsections V-C and V-D, we compare the effectiveness of P4WISE against a supervised learning approach and other accelerator-aware policies.

A. P4WISE *Tuning*

We divide this subsection into two main parts. The first focuses on tuning the parameters within the reward function, while the second analyzes the environment to identify optimal values for the Bellman equation parameters, α and γ. For the policy π, we use an ϵ-greedy approach with $\epsilon = 0.1$, meaning the agent selects the action with the highest Q-Value 90% of the time. As demonstrated in Section V-C, this setting yields promising results. However, in environments with frequent changes, a higher ϵ value may enhance adaptability. Additionally, we partition the average utilization range in the Q-Table into 20 equal subranges ($n = 20$), resulting in 40 rows total, as previously described in Section III-C.

*1) **Tuning the Reward Function**:* Two thresholds in the reward function need to be determined: $latency\_threshold$ and $utilization\_threshold$. According to our initial investigation, the system's 99$^{\text{th}}$ percentile E2E latency is consistently less than 1 second for machine learning tasks like KNN-based inference using the best policies priority (PRT) or weighted round robin (WRR) [11]. Consequently, we consider this value the maximum acceptable E2E latency in the reward function.

To identify the optimal value for $utilization\_threshold$, we conducted an experiment with 128 servers, considering three threshold values as illustrated in Figure 4. Instead of P4WISE's agent, which utilizes a Q-Learning-wise RL, the agent in this experiment only switches between PRT and WRR policies. The goal is to find at which threshold we should change the policy. With $utilization\_threshold = 50\%$, P4WISE responds too quickly, changing the policy once the servers' utilization reaches 50%, leading to higher 99$^{\text{th}}$ percentile rates at around $1.3K$ requests per second (RPS). Conversely, for $utilization\_threshold = 70\%$, which implies continuing to use the server up to 70% utilization, it exhibits sub-optimal performance with rates ranging from $1.6K$ to $1.7K$ RPS but eventually shifts to appropriate weight settings at higher rates. The threshold value $utilization\_threshold = 60\%$, however, demonstrates the best performance, responding to system conditions optimally at rates around $1.4K$ to $1.5K$ RPS. Furthermore, at this utiliztion, the 99$^{\text{th}}$ percentile E2E latency reaches $1s$ as well. We adopted $utilization\_threshold = 60\%$ for all subsequent experiments in this section.

The last parameter to configure within the reward function is max_reward, and in our experiments, we have chosen it as 10. Given that we receive a -1 reward at each step, any value greater than 1 is suitable for this parameter. Its primary purpose is to indicate whether we have reached a terminal state.

*2) **Tuning Bellman Equation's Parameters**:* The Bellman equation involves two parameters, both ranging from 0 to 1. The first parameter, denoted as α or the `learning rate`,

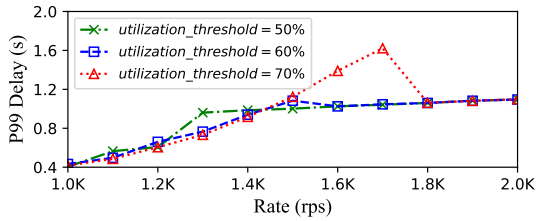


Fig. 4: Tuning *utilization_threshold* for P4WISE.

influences the granularity of learning. A smaller α value enhances precision but extends the training duration. The second parameter, γ or the `discount rate`, denotes the significance of the subsequent state in the learning process.

In our experiments, P4WISE converges with different values for the mentioned parameters when setting a coarse value for γ, indicating that future rewards are heavily given significant importance in the decision-making process; nevertheless, the number of iterations required for convergence may vary. Figure 5 illustrates the accuracy of reaching a set of proper weights at the end of 100 recent episodes, which demonstrates that P4WISE can predict the right weights even with coarse values like $\alpha = 0.3$ or 0.5, but considering that with $\alpha = 0.5$ the system converges faster. These values are suitable in our non-complex setup, where we have only one type of accelerator, and the convergence occurs in a few steps. In the remainder of the manuscript, we consider $\alpha = \gamma = 0.5$.

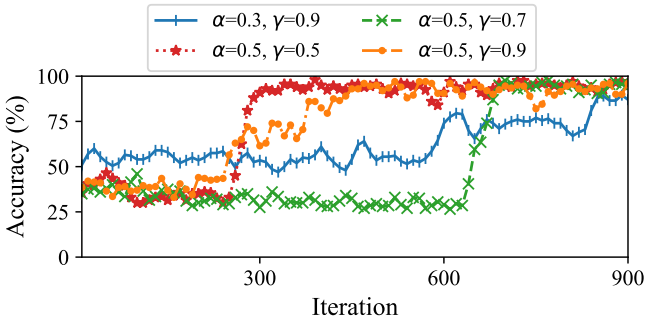


Fig. 5: Tuning Bellman equation's parameters.

Expected Convergence. While it is not possible to determine the exact number of iterations required for Q-Learning to converge, prior studies—such as [28]—have analyzed its time complexity. The size of the state and action spaces typically has the most significant impact on convergence time, while parameters such as the discount factor γ and learning rate α also influence the overall training duration. Based on these considerations, the Q-Learning, as observed, should converge within a few hundred iterations in our setting as we have 40 states and 3 possible actions.

B. P4WISE *at Different Scales*

In previous sections, we trained P4WISE agent using the mentioned parameters, in an environment with 128 servers and two SmartNICs per server. Here, we assess the performance of the trained agent across different scales. Because P4Wise is expected to run on a TOR switch, which typically has a number of connected devices limited by the number of ports, ranging from 64 to 256 ports, we consider this scale in our evaluation. We adjust the server count, conducting evaluations for each configuration, and compare the performance of P4WISE with the best policies of P4HAULER where applicable.

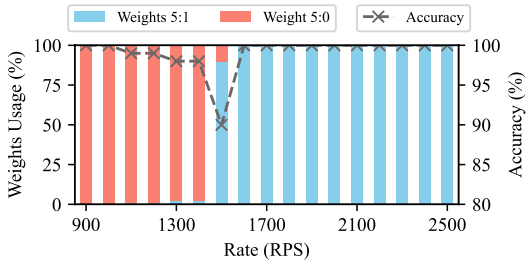


Fig. 6: P4WISE's outcome for 128 servers after training.

Figure 6 depicts the performance of P4WISE's agent with the specified parameters after training, prior to any changes in the setup. On the plot's first y-axis, we observe the percentage of weight usage between $5:1$ and $5:0$ for various rates, while the secondary y-axis displays the accuracy of correct weight selection by P4WISE. We assume that $5:0$ is the right weight for rates less than $1.4k$ RPS, and $5:1$ for higher rates. With this consideration, at turning point of $1.4k$ RPS, P4WISE's accuracy is 90%. For other points, the accuracy of correct weight selection is above 90%. It is important to highlight that the agent tests various weights during training; however, few may yield promising results.

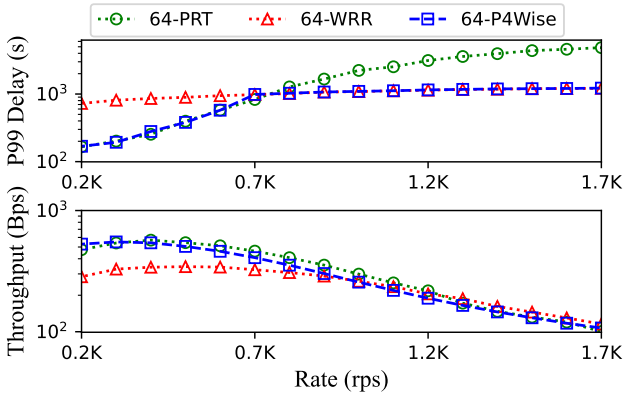


Fig. 7: Balancing the load among 64 servers with P4WISE.

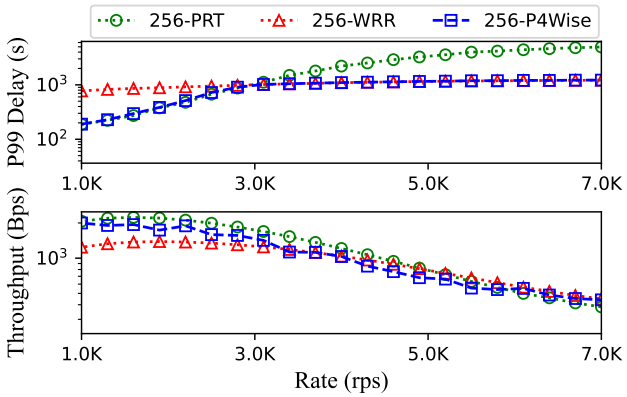


Fig. 8: Balancing the load among 256 servers with P4WISE.

Figure 7 illustrates the 99th percentile delay and throughput for a data center comprising 64 servers. As expected, P4WISE demonstrates a 99th percentile delay closely mirroring the optimal policy, PRT, at lower rates (below 700 RPS), and converges toward the behaviour of WRR at higher rates (above 700 RPS). Similarly, the throughput of P4WISE aligns with the higher throughput values between PRT and WRR. P4WISE's key advantage lies in its ability to automatically select the optimal policy, whereas P4HAULER requires a network administrator to manually determine the best settings (e.g., appropriate weights) to achieve optimal performance. We should also notice that P4WISE's throughput is slightly lower than the optimal as the agent may not choose the action with the highest Q-Value with the possibility of ϵ, considered in the policy π.

The same pattern is observed at various rates for a system with 256 servers, as depicted in Figure 8. Up to a rate of $3.0K$ RPS, P4WISE's 99th latency and throughput closely match those of P4HAULER's PRT policy. Beyond this point, P4WISE adjusts its weight to mimic P4HAULER's WRR policy for higher rates, and it achieves comparable performance as well.

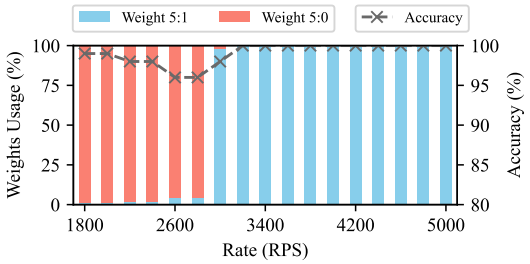


Fig. 9: P4WISE's outcome for 256 servers.

Figure 9 showcases the selected weights and the accuracy of weight selection for P4WISE in the scenario with 256 servers. The identified weights correspond to our observations in Figure 8. For rates up to $3k$ RPS, the weights correlate with PRT with an accuracy of 95%. However, at higher rates, a ratio of $5:1$ is applied.

C. P4WISE *vs. Supervised Learning*

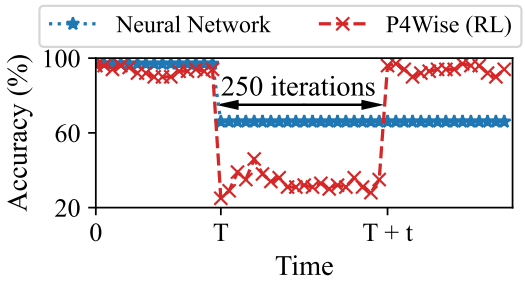


Fig. 10: P4WISE vs. Supervised Learning approach.

One of the advantages highlighted in Section II-C regarding P4WISE is its ability to continuously adapt to new hardware configurations without relying on training data. In other words, it can reconfigure itself in the face of environmental changes. Essentially, P4WISE can actively evaluate the rewards of actions to validate their correctness, enabling it to change the weights in the environment dynamically. In contrast, a supervised learning approach necessitates the collection of data, training of the model, and redeployment of the model. In this experiment, depicted in Figure 10, we examine a scenario with 128 servers, each equipped with one SmartNIC. Following data collection, we trained a neural network with three layers, each consisting of ten nodes, to select among predefined weights. The accuracy of this neural network in this configuration is approximately 97%. Conversely, we employed P4WISE online, achieving an accuracy exceeding 90% with this setup. At $time = T$, we introduced a second SmartNIC to all servers, observing a decrease in the accuracy of the supervised learning model to 65%, since the neural network is not trained with any data with two SmartNICS. Conversely, P4WISE required approximately 250 iterations, if we reset its Q-Table and start over, to learn the environmental changes and regain its accuracy.

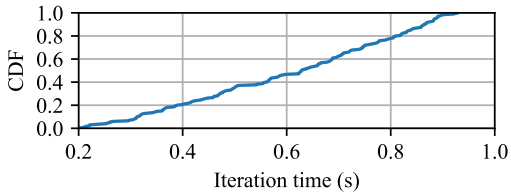


Fig. 11: Distribution of iterations' time for training P4WISE.

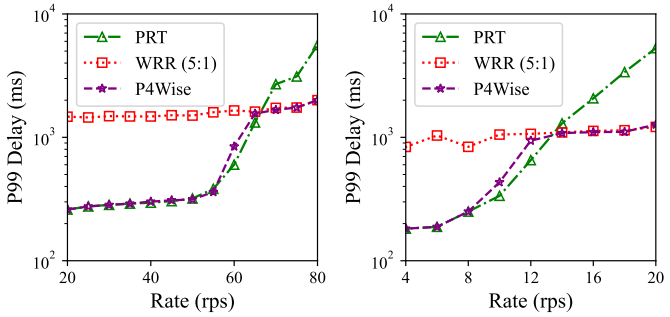
Figure 11 illustrates the time distribution across 250 iterations required to train the P4WISE agent. In this simulation, ran on one core of our server, to ensure comprehensive training over all possible inputs, we varied the input rate randomly and maintained each rate until the RL agent converged to the optimal weights. For each input rate, P4WISE performs a few iterations to update the Q-Table, and in each iteration, P4WISE applies and evaluates a set of weights, and waits until it observes the outcomes of the weight. Since the waiting times vary for each application[3], we eliminate it from the results, and focus solely on the duration taken by the P4WISE agent to compute the weights and update the Q-Table. According to Figure 11, on average, each iteration requires roughly $0.6s$. That means, the overall 250 iterations are done in less 3 minutes. In other words, upon a drastic change in the system, P4WISE adapts to the new configurations within a few minutes if the RL model executes in the user space.

D. P4WISE *vs.* P4HAULER

Up to this point, we evaluated P4WISE through simulations. In this subsection, we conducted two additional experiments in our hardware testbed to compare P4WISE with P4HAULER, the state-of-the-art in-network load balancer. We consider two applications, KNN and BM24[4]. While P4WISE automati-

[3]The monitoring agents transmit updates every $10ms$; however, a slightly longer delay may be required for observable changes to take effect. Nonetheless, this delay is negligible in comparison to the computation time of the RL agent

[4]Since we had tuned the parameters for KNN, we did the tuning again for BM25 application.



(a) BM25 (b) KNN

Fig. 12: P4WISE vs P4HAULER's policies.

cally applies a promising policy using reinforcement learning, P4HAULER provides a framework enabling network administrators to implement various load balancing algorithms. We activate PRT and WRR for P4HAULER in this experiment as they show the best results for P4HAULER[11].

Figure 12 illustrates the 99th percentile of the end-to-end latency for inference requests load-balanced by P4WISE and P4HAULER. The figure reveals that none of P4HAULER's algorithms consistently achieve optimal latency across different rates. At lower rates, PRT outperforms WRR, while at higher rates, WRR surpasses PRT by distributing the load according to each resource's capacity. In contrast, P4WISE consistently identifies and applies weights that mimic the performance of the best algorithm. For instance, in Figure 12a, PRT demonstrates the best end-to-end latency up to a rate of 65 RPS for BM25, while WRR outperforms PRT for higher rates. However, P4WISE exhibits latency similar to PRT's for rates below 65 RPS and similar to WRR's for rates higher than 65 RPS. This trend is also observed in Figure 12b, which illustrates KNN's end-to-end latency. P4WISE's latency closely resembles PRT's for rates below 14 RPS and converges towards WRR for rates higher than 18.

Key Takeaways: In dynamic data center environments, load balancing approaches that rely on static or supervised learning techniques tend to underperform since they cannot optimally capture upgrades and changes in the configuration of servers, accelerators and other components. We found that by adding reinforcement learning capabilities to an in-network load balancer enables it to autonomously adapt to new configurations resulting from infrastructure refresh cycles, without manual intervention.

VI. RELATED WORK

With the recent advancements in programmable data planes within software-defined networking, there has been a growing trend of offloading various applications to switches operating the data plane. These applications encompass a wide range, including but not limited to distributed systems (e.g., NetCache [37] and NetGVT [38]), machine learning/deep learning (e.g., NetPixel [39]), and security (e.g., PoirIoT [40], [41]), and load balancing (e.g., SilkRoad [3], Cheetah [9], P4MITE [10] and P4HAULER [11]). Among the mentioned load balancers, P4MITE and P4HAULER stand out as the first and the second ones that distribute load at the granularity of accelerators. P4MITE runs a naive load balancer; however, P4HAULER supports multiple load balancing policies. It requires extensive manual configuration and tuning whenever the environment changes. To overcome this limitation, this study introduces P4WISE, which maintains the fine-grained load balancing of P4HAULER while eliminating the need for manual tuning through machine learning.

Table I lists a few notable load balancers leveraging AI/ML, yet not operating at the accelerator level. In a nutshell, the first four load balancers focus on distributing the network load among multiple paths in the network. On the other hand, the last two distribute the load among the available servers.

Load distribution among network paths. Learned Load Balancing (LLB) [16] leverages neural networks to optimize load distribution in a network. Rather than relying on traditional static load balancing methods, LLB dynamically adapts and improves over time based on real-time network conditions. The machine-learning model learns patterns by analyzing factors such as network traffic, server capacity, and latency and makes intelligent decisions regarding load distribution. This learning-based approach enables load balancers to optimize resource utilization, minimize congestion, and enhance overall network performance. Learned load balancing offers a flexible and scalable solution that can adapt to changing network dynamics, making it a valuable tool in modern network management. In [16], Chang *et al.* adopt a SmartNIC to generate the weights as not only they can execute inference but also, due to hardware performance predictability, it is feasible to estimate an upper bound of the latency for the load balancing process.

TABLE I: In-network load balancers utilizing AI/ML.

Load Balancer	Attitude in LBing	Deployment
LLB [16]	The network paths	In a SmartNIC
QCMP [17]	The network paths	In a Programmable Switch
CrossBal [42]	The network paths	In a Programmable Switch
RWCMP [43]	The network paths	Simulation
LBAS [4]	The servers	In a Programmable Switch
Training Wheel [44]	The servers	No specified details

Load balancing is further compounded by the dynamic nature of network traffic and the growing diversity of workloads traversing the network. QCMP [17] is a load balancing solution based on reinforcement learning. QCMP is integrated into the data plane, allowing for rapid policy adjustments in response to fluctuations in traffic. The implementation of QCMP utilizes P4 on a Tofino switch and employs BMv2 within a simulation environment. CrossBal [42] introduces a hybrid load balancing approach that leverages Deep Reinforcement Learning (DRL) to prioritize optimizing high-impact elephant flows. The DRL agent is designed to make effective use of network links while keeping the action space minimal, enabling the agent to rapidly acquire load balancing skills. Additionally, CrossBal maintains the ability to swiftly adapt to network modifications by actively monitoring and switching routes in the data plane.

Lim *et al.* [43] address the challenge of efficiently distributing traffic in data center networks with multi-rooted topologies to optimize bisection bandwidth. They introduce a load-balancing approach called Reinforcement Weight-Cost Multipath (RWCMP) that employs reinforcement learning to determine the ideal traffic split ratios for egress ports and route multiple flows simultaneously.

Unlike the research efforts above, which distribute network traffic across various paths, P4WISE distributes computing loads among servers' CPUs, GPUs, and smart NICs.

Load distribution among servers. LBAS [4] presents a fast switch-based load balancer that considers the states of application servers. In contrast to previous load balancers that solely focus on distributing incoming traffic, this load balancer considers the conditions of the servers to make intelligent load-balancing decisions. By utilizing the controller in the network architecture, the load balancer can efficiently monitor the states of the application servers quickly. This information is then used in a regression model to dynamically distribute the incoming traffic based on the current states of the servers, such as their CPU utilization and response times. The load balancer aims to optimize the overall performance of the system by effectively utilizing the available server resources and preventing overload on any particular server.

RL models present a robust alternative to manually designed heuristics within networked systems. However, they can exhibit unpredictable behavior, raising safety concerns. In [44], Mao *et al.* addressed this challenge by introducing a method that learns dynamically but switches to a straightforward, well-established fallback policy if the system encounters an unsafe state space. This approach ensures adequate feedback for RL to acquire an effective policy while maintaining safety. Nonetheless, they assessed their method over workload changes in a very limited setup.

In summary, although previous studies have addressed environmental changes through the adoption of RL [17] or regression models [4], they cannot account for various resources (CPU, GPU, SmartNIC) to achieve more effective load balancing. As accelerator-aware load balancing proved to be efficient, we consider developing an RL-based load balancer that is accelerate-aware while distributing load among computing resources, i.e., making it programmable and intelligent.

VII. Conclusions

This paper presents P4WISE, a novel reinforcement learning (RL)-based in-network load balancer that is accelerator-aware. It dynamically distributes workloads across various computing resources—including data center servers and accelerators—to adapt to changing environmental conditions and application demands. The load balancing functionality is divided between the control and data planes. The control plane hosts the RL agent, which analyzes collected data to learn and assign optimal weights to computing resources based on their current status. Meanwhile, the data plane implements a Dynamic Weighted Round-Robin (DWRR) mechanism to distribute load according to the learned weights. It also tracks resource utilization on servers and their accelerators, performing packet forwarding while ensuring per-connection consistency.

Extensive evaluations of simulators and testbeds demonstrated the effectiveness and feasibility of P4WISE in distributing loads among various computing resources. Specifically, it outperforms state-of-the-art accelerator-aware load balancers in adapting load-balancing policies to the changing application demands and the conditions of computing resources. As part of future work, we plan to extend P4WISE for deployment in environments with heterogeneous servers. Since more configurations are required, this extension will likely increase communication between the control and data planes, which warrants further investigation and potential optimization.

References

[1] S. K. Mishra, B. Sahoo, and P. P. Parida, "Load balancing in cloud computing: a big picture," *Journal of King Saud University-Computer and Information Sciences*, vol. 32, no. 2, pp. 149–158, 2020.

[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] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 15–28, 2017.

[4] J. Zhang, S. Wen, J. Zhang, H. Chai, T. Pan, T. Huang, L. Zhang, Y. Liu, and F. R. Yu, "Fast switch-based load balancer considering application server states," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1391–1404, 2020.

[5] R. Gandhi, Y. C. Hu, and M. Zhang, "Yoda: A highly available layer-7 load balancer," in *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1–16, 2016.

[6] S. Rajagopalan, "An overview of layer 4 and layer 7 load balancing," *Computer Networks, Big Data and IoT: Proceedings of ICCBI 2020*, pp. 663–672, 2021.

[7] N. Nimse, "An introduction to dedicated server load balancing." https://www.linkedin.com/pulse/introduction-dedicated-server-load-balancing-nisha-nimse, 2023 (accessed December 1, 2023).

[8] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. M. Jr., P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 667–683, 2020.

[9] T. Barbette, E. Wu, D. Kostić, G. Q. Maguire, P. Papadimitratos, and M. Chiesa, "Cheetah: A high-speed programmable load-balancer framework with guaranteed per-connection-consistency," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 354–367, 2021.

[10] H. Tajbakhsh, R. Parizotto, M. Neves, A. Schaeffer-Filho, and I. Haque, "Accelerator-aware in-network load balancing for improved application performance," in *2022 IFIP Networking Conference (IFIP Networking)*, pp. 1–9, 2022.

[11] H. Tajbakhsh, R. Parizotto, A. Schaeffer-Filho, and I. Haque, "P4hauler: An accelerator-aware in-network load balancer for applications performance boosting," *IEEE Transactions on Cloud Computing*, 2024.

[12] X. Zhou, N. Shroff, and A. Wierman, "Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers," *Performance Evaluation*, vol. 145, p. 102146, 2021.

[13] J. Shalf, "The future of computing beyond Moore's law," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190061, 2020.

[14] T. Xing, H. Tajbakhsh, I. Haque, M. Honda, and A. Barbalace, "Towards portable end-to-end network performance characterization of smartnics," in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 46–52, 2022.

[15] S. Moss, *Google increases server life to six years, will save billions of dollars*. Data Center Dynamics, 2023 (accessed May 1, 2024). https://www.datacenterdynamics.com/en/news/google-increases-server-life-to-six-years-will-save-billions-of-dollars/.

[16] B. Chang, A. Akella, L. D'Antoni, and K. Subramanian, "Learned load balancing," in *24th International Conference on Distributed Computing and Networking*, pp. 177–187, 2023.

[17] C. Zheng, B. Rienecker, and N. Zilberman, "Qcmp: Load balancing via in-network reinforcement learning," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, pp. 35–40, 2023.

[18] T. Hesam, "P4Hauler." https://github.com/PINetDalhousie/P4Hauler/, 2022.

[19] Intel, *Explore the Power of Intel® Programmable Ethernet Switch Products*. Intel, 2023 (accessed December 1, 2023). https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[20] "P4Wise." https://github.com/PINetDalhousie/P4Wise, 2024.

[21] T. Hesam, "P4Wise." https://github.com/PINetDalhousie/P4Wise, 2022.

[22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[23] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[24] J. Zhang, Y. Gao, S. Wen, T. Pan, and T. Huang, "Loom: Switch-based cloud load balancer with compressed states," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pp. 1–11, IEEE, 2021.

[25] C. Zeng, L. Luo, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, *et al.*, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[26] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[27] E. Odemakinde, "Model-based and model-free reinforcement learning: Pytennis case study." https://neptune.ai/blog/model-based-and-model-free-reinforcement-learning-pytennis-case-study, 2023 (accessed December 1, 2023).

[28] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[29] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep q-learning," in *Learning for dynamics and control*, pp. 486–489, PMLR, 2020.

[30] Z.-x. Xu, L. Cao, X.-l. Chen, C.-x. Li, Y.-l. Zhang, and J. Lai, "Deep reinforcement learning with sarsa and q-learning: A hybrid approach," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 9, pp. 2315–2322, 2018.

[31] V. Valkov, "Solving an mdp with q-learning from scratch — deep reinforcement learning for hackers (part 1)." https://venelinvalkov.medium.com/solving-an-mdp-with-q-learning-from-scratch-deep-reinforcement-learning-for-hackers-part-1-45d1d360c120, 2023 (accessed December 1, 2023).

[32] T. Sarkar, *Heterogenous and Homogenous Cloud Platforms — Cyfuture Cloud*. Cyfuture Cloud, 2024 (accessed May 1, 2024). https://cyfuture.cloud/blog/understanding-heterogenous-and-homogenous-cloud-platforms/.

[33] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "KNN model-based approach in classification," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pp. 986–996, Springer, 2003.

[34] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[35] S. Robertson, H. Zaragoza, *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[36] D. Wadden, S. Lin, K. Lo, L. L. Wang, M. van Zuylen, A. Cohan, and H. Hajishirzi, "Fact or fiction: Verifying scientific claims," in *EMNLP*, 2020.

[37] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 121–136, 2017.

[38] R. Parizotto, B. Mello, I. Haque, and A. Schaeffer-Filho, "NetGVT: offloading global virtual time computation to programmable switches," in *Proceedings of the Symposium on SDN Research*, pp. 16–24, 2022.

[39] H. Siddique, M. Neves, C. Kuzniar, and I. Haque, "Towards network-accelerated ml-based distributed computer vision systems," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 122–129, IEEE, 2021.

[40] C. Kuzniar, M. Neves, V. Gurevich, and I. Haque, "IoT device fingerprinting on commodity switches," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9, IEEE, 2022.

[41] C. Kuzniar, M. Neves, V. Gurevich, and I. Haque, "Poiriot: Fingerprinting iot devices at tbps scale," *IEEE/ACM Transactions on Networking*, 2024.

[42] B. Coelho and A. Schaeffer-Filho, "Crossbal: Data and control plane cooperation for efficient and scalable network load balancing," in *2023 19th International Conference on Network and Service Management (CNSM)*, IEEE, 2023.

[43] J. Lim, J.-H. Yoo, and J. W.-K. Hong, "Reinforcement learning based load balancing for data center networks," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pp. 151–155, IEEE, 2021.

[44] H. Mao, M. Schwarzkopf, H. He, and M. Alizadeh, "Towards safe online reinforcement learning in computer systems," in *NeurIPS Machine Learning for Systems Workshop*, 2019.

Hesam Tajbakhsh Hesam Tajbakhsh earned his doctoral degree from Dalhousie University in Halifax, Canada, where he conducted his research within the Programmable and Intelligent Networking (PINet) Laboratory, under the supervision of Dr. Israat Haque. His research interests span across multiple aspects of computer science, including software-defined networking, performance evaluation, data center networking, networking for machine learning, and machine learning for networking.

Ricardo Parizotto Ricardo Parizotto holds a Ph.D. in Computer Science from the Federal University of Rio Grande do Sul (UFRGS, 2024). During his Ph.D. he was a visiting researcher at Dalhousie University, Canada. He is currently an Assistant Professor at the Federal University of the Fronteira Sul (UFFS). His research interests include in-network computing systems, fault tolerance and network security.

Alberto Schaeffer-Filho Alberto E. Schaeffer-Filho holds a Ph.D. in Computer Science (Imperial College London, 2009) and is an Associate Professor at Federal University of Rio Grande do Sul (UFRGS), Brazil. Dr. Schaeffer-Filho is a CNPq-Brazil Research Fellow and his areas of expertise are network/service management, network virtualization, software-defined networking, and security and resilience of networks. He has authored over 120 papers in leading peer-reviewed journals and conferences related to these topics, and also serves as TPC member for important conferences in these areas, including: CNSM, NetSoft, IEEE/IFIP NOMS and IEEE ICC. He served as the general co-chair for SBRC 2019, TPC co-chair for IFIP/IEEE IM 2021, TPC co-chair for SBRC 2021, TPC co-chair for SBSeg 2022 and TPC co-chair for CNSM 2025. He is also a member of the IEEE.

Israat Haque Dr. Israat Haque is an Associate Professor in the Faculty of Computer Science at Dalhousie University, where she leads the Programmable and Intelligent Networking (PINet) Lab. Her expertise is in distributed and networking systems and security. She leverages network programmability to develop cutting-edge, high-performance, secure, and dependable systems for AI/ML, Big Data, cloud, telecommunication, and IoT systems. She is also interested in applying data-driven approaches to solve practical and relevant problems. Dr. Haque was recognized as an ACM/IEEE N2Women Rising Star in 2021 for her research and leadership contributions. Subsequently, she received Digital Nova Scotia's Thinking Forward Award 2022 for training the next generation of tech talents. In 2024, she received the Alumni Honour Award from her Alma Mater, University of Alberta.