PROGRAMMABLE AND INTELLIGENT ACCELERATOR-AWARE LOAD BALANCERS IN DATA CENTERS

by

Hesam Tajbakhsh

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
December 2023

Dalhousie University is located in Mi'kma'ki, the
ancestral and unceded territory of the Mi'kmaq.
We are all Treaty people.

I dedicate this thesis to my beloved wife, Sara, who has been my unwavering source of support, encouragement, and inspiration throughout this academic journey. Your love, patience, and belief in me have been the driving force behind my success.
To my parents and my sister, thank you for your continuous encouragement, understanding, and sacrifices. Your unwavering support has been a constant source of strength.
This thesis is a tribute to all of you, as you have been my pillars of strength, and I am forever grateful for your presence in my life.

Table of Contents

List of Tables

List of Figures

Abstract

The slowdown in CPU progress prompted system designers to incorporate diverse programmable accelerators (e.g., graphics processing unit (GPU), smart network interface card (SmartNIC) to address the insufficient computational capacity needed for various components within computer systems. While these programmable accelerators enhance computational capabilities, they possess distinct architectures and capacities compared to standard CPUs. Thus, it is essential to judiciously distribute the computing tasks among servers and their accelerators to avoid performance degradation. Software-defined networking is a paradigm that enables network programmability for agile and efficient network management and operations. Programmable hardware (e.g., switch) recently became a promising alternative for task distribution decisions. A programmable switch can process packets in real-time at line rates (Tbps) significantly faster than legacy server-based load balancers (LBs). Furthermore, such in-network load balancers can reduce the delay in decision-making by cutting off the latency for sending packets from the switch to load-balancing servers. There are several load balancers deployed in programmable switches, but none incorporate the capabilities of accelerators in their designs.

In this thesis, we propose the first in-network accelerator-aware load balancers for performance improvement of machine learning applications in data centers. The first load balancer is called *P4Mite*, which deploys agents in application processing servers and accelerators to measure their capacity and shares these statuses with the switch. It uses this information and load balancing policies (e.g., weighted round robin) to dispatch loads among servers and their accelerators. However, *P4Mite* supports a limited number of policies. Thus, we introduce *P4Hauler*, which provides a load-balancing framework to support a wide range of policies. Within this framework, we propose configurable building blocks that operators can dynamically select to implement various policies on-the-fly without rebooting the switch and interrupting its services. In addition to knowing the policies and statuses of accelerators, an LB must be aware of traffic condition, which makes the LB operation tedious. Thus, we propose *P4Wise*, a learning-based LB, to select the most suitable distribution policy automatically.

We implement a prototype of the proposed load balancer and deploy it on a testbed consisting of a programmable switch (Intel Tofino), SmartNICs (Mellanox BlueField), and legacy servers to demonstrate deployment feasibility and efficiency over existing solutions. Then, we develop a realistic simulator to show the performance at scale. Specifically, *P4Hauler* can handle 27% more load compared to traditional LBs using only a single accelerator. In the case of hundreds of servers with multiple accelerators, the performance improvement is proportional to the number of available accelerators. Finally, *P4Wise* consistently selects appropriate weights with an accuracy of at least 90%. Furthermore, it responds to changes in the environment by adapting the load balancing approach accordingly.

List of Abbreviations

AA	Accelerator Aware
AD	Anomaly Detection
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DIP	Direct IP
DL	Deep Learning
DQN	Deep Q-Learning
DRL	Deep Reinforcement Learning
E2E	End to End
ECMP	Equal Cost Multi Path
FCT	Flow Completion Time
FPGA	Field Programmable Gate Array
GFLOP	Giga Floating-point OPeration
GPU	Graphics Processing Unit
IC	Image Classification
IP	Interner Protocol Address
JIQ	Join the Idle Queue
JSQ	Join the Shortest Queue

KNN	K-Nearest Neighbors
LB	Load Balancer
LED	Local Estimation Driven
LUR	Least Utilized Resource
ML	Machine Learning
MLP	Multilayer Perceptron
NIC	Network Interface Card
NLP	Natural Language Processing
NN	Neural Network
P4	Programming Protocol-independent Packet Processors
PCC	Per-Connection Consistency
PCIE	Peripheral Component Interconnect Express
PDP	Programmable Data Plane
PISA	Protocol Independent Switch Architecture
Po2	Power-of-Two
PRT	Prioritization
RA	Resource Aware
RAM	Random Memory Access
RDMA	Remote Direct Memory Access
RL	Reinforcement Learning
RPC	Remote Procedure Call
RR	Round Robin

SDN	Software Defined Networking
SoC	System On a Chip
SRAM	Static Random Memory Access
SSD	Solid State Drive
TAN	Tofino Native Architecture
TCAM	Ternary Content Aaddressable Memory
TCP	Transmission Control Protocol
TPU	Tensor Processing Unit
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit Program
VIP	Virtual IP
WCMP	Weighted Cost Multi Path
WRR	Weighted Round Robin

Acknowledgements

I would like to take this opportunity and convey my sincere gratitude to the individuals and institutions who have played a crucial role in the completion of this thesis.

I express my sincere gratitude to my thesis advisor, Dr. Israat Haque, for her invaluable guidance, expertise, and mentorship throughout this research journey. Her insightful feedback and unwavering support have played a pivotal role in shaping this study. Additionally, I extend my heartfelt thanks to Dr. Alberto E. Schaeffer-Filho for his valuable assistance and insightful guidance.

I would like to acknowledge the participants of my study who generously contributed their time and insights. Their willingness to be part of this research was critical to its success. Specifically, I would like to thank Ricardo Parizotto, Carson Kuzniar, Jack Zhao, Tong Xing, Dr. Miguel Neves, and Dr. Antonio Barbalace.

Your collective support, guidance, and encouragement have been essential in bringing this thesis to completion. I am truly grateful for your kindness and assistance.

Chapter 1

Introduction

1.1 Overview

The deceleration in CPU advancements seen at the beginning of the 21st century [1] pushed system designers to introduce various types of programmable accelerators to cope with the lack of computation required for different components of computer systems. SmartNICs, for example, are accelerators developed to operate on Gbps communication links and alleviate the load from the CPUs. GPUs, programmable SSD, and FPGA are other examples of programmable accelerators introduced to add more resources alongside CPUs for graphical computations, storage, and miscellaneous, respectively.

Although programmable accelerators add some computation power to the system, they have different architectures than regular CPUs. For instance, designers use wimpy ARM-based CPUs for embedded devices like SmartNICs. The reason is that ARM architecture is more energy-efficient than x86 [2]. As a result, the performance falls by running applications intended for regular servers on accelerators with diverse architectures. Thus, it becomes crucial to distribute the workload effectively between servers and their accelerators.

Load balancers (LBs) are utilized to enhance application performance by distributing packets across multiple devices [3]. Generally, load balancers employ policies such as *Weighted Round Robin* (WRR), *Join-the-Shortest-Queue* (JSQ), *Join-the-Idle-Queue* (JIQ), *Local-Estimation-Driven* (LED) [4], to determine the packet's destination. Additionally, a policy must guarantee *per-connection consistency*, meaning that the load balancer forwards all packets from a connection to the same end server. Traditionally, these load balancers have been deployed on middleboxes.

Nevertheless, there has been a recent shift towards utilizing programmable data plane (PDP) hardware as a promising alternative for load-balancing tasks and scheduling decisions [5]. Instead of relying on a centralized solution running on a dedicated server, deploying a load balancer within the network can yield quicker responses and reduce flow completion time. Several existing studies deployed in the data plane have specifically focused on developing load balancers that ensure per-connection consistency while effectively dispatching packets to servers [6]. Recent solutions have concentrated on increasing the capacity of switches to handle concurrent connections [7] or optimizing resource usage, such as memory consumption [8]. However, the current load balancers, including Tiara [9], Cheetah [10], SilkRoad [6], and Beamer [11], lack visibility of accelerators. In other words, these load balancers consider an accelerator to be equivalent to a server, which can lead to imbalances between servers and their corresponding accelerators.

Furthermore, aside from the issue of limited visibility, current in-network load balancers only enforce rigid, hard-coded policies within the switch source code, lacking the ability to be dynamically modified. However, the effectiveness of different policies varies depending on the characteristics of the applications [12]. For instance, an application that heavily relies on computational resources may benefit from policies prioritizing devices with powerful CPUs. Unfortunately, existing in-network load balancers require network operators to manually configure the switch to apply a new load-balancing policy and activate it for supporting a different application [13]. This requires rebooting the switch and can lead to unexpected service interruptions.

This study introduces *P4Mite*, *P4Hauler*, and *P4Wise*, three load balancers specifically designed to function on programmable switches, such as the Intel Tofino. To be more specific, the functioning of all three load balancers relies on agents operating on servers and accelerators. These agents gather statistics and transmit the monitored resource statuses of these devices directly to the data plane in the switch. The switch maintains a record of the current status of each resource and utilizes these values to make decisions regarding the destination for incoming requests.

In the case of *P4Mite*, our focus is on SmartNICs, which are specialized network interface cards (NICs) equipped with processors capable of processing packets either on the network path or off the path. Many prominent vendors, including NVIDIA and Broadcom, have developed a range of SmartNICs to augment the computational capabilities of data center machines' CPUs. Our objective is to assess the potential of SmartNICs for processing CPU-intensive applications and explore how a load balancer that is aware of these accelerators can optimize overall performance by efficiently utilizing each available resource. In *P4Mite*, the agents reported whether a target computing resource was available. However, for more intricate policies that involve collecting multiple resources or performing computations within the switch, it is not practical to store all potential outcomes due to memory limitations.

Once we show that distributing the load at the granularity of accelerators can improve the performance, we extend and generalize the initial design and introduce *P4Hauler*, which is both accelerator-aware and policy-agnostic, enabling dynamic policy updates without requiring a switch reboot, in the opposite of existing solutions. Put differently, *P4Mite*, stands as an example of *P4Hauler*. *P4Hauler*'s flexibility allows it to adapt its operations according to varying network and application conditions. It is worth mentioning that *P4Mite* and *P4Hauler* contribute to reducing the CPU resource burden on servers in two ways: firstly, by offloading a portion of the application load onto accelerators, and secondly, by ensuring that servers and accelerators are not involved in load balancing operations. As a result, computing resources can be freed up to handle application requests more efficiently.

Based on our observations in *P4Hauler*, we have found that there is not a single policy that consistently delivers required performance in all network conditions (e.g., varying traffic rates, number of servers or accelerators). Specifically, in *P4Hauler*, network administrators activate policies for a given application that remains unchanged during its entire service time irrespective of network dynamics, which can adversely impact the end-to-end latency and throughput. Furthermore, in the event of substantial deployment changes, such as modifications to the existing resources or the addition of new resources, operators must update the policy configurations accordingly. In order to tackle this challenge, we have adopted reinforcement learning (RL) and introduced *P4Wise*. It is a hybrid system where the RL agents get trained in the control plane interacting with the environment. In contrast, the data plane switch gets configured with the chosen policy and its weights to distribute loads accordingly. Thus, *P4Wise* makes decisions on behalf of the operator by continuously monitoring resource usage and employs an RL algorithm to determine the most effective load-balancing strategy based on the observed information.

We implemented prototypes of *P4Mite*, *P4Hauler*, and *P4Wise* on an Intel Tofino

[14] programmable switch. Our prototypes are publicly available at our Git repository [15, 16]. The prototypes are deployable on data centers' networks to distribute the load among servers and their accelerators. Specifically, we evaluated the proposed systems over popular machine learning applications (e.g., image processing or natural language processing); however, our proposed load balancers can support any required applications. The results reveal that *P4Hauler* can support significantly more load, proportional to the number and capacity of the accelerators, compared to its counterpart. For instance, *P4Hauler* can increase the maximum request rate by 27% and reduce the flow completion time by 13% with only one wimpy additional accelerator (SmartNIC). Furthermore, simulations with hundreds of servers, each with a couple of SmartNICs, show that we can handle around 50% more loads. Furthermore, our evaluation reveals that when appropriately configured, *P4Wise* consistently selects the optimal load-balancing policy in different networking conditions. This implies that not only does it exhibit similar performance enhancements to *P4Hauler* during peak load conditions, but it also eliminates the need for network administrators to continuously update policies when the system is not under high load.

1.2 Research Contributions

In this article, we contribute in the following ways:

- We introduce *P4Mite*, *P4Hauler*, and *P4Wise*, three systems designed for load balancing in an environment comprising servers and hardware accelerators. At the core of them lies a generalized switch that enables load balancing at an accelerator-level granularity.

- We assess the potential of accelerators like SmartNICs in *P4Mite* for CPU-intensive applications' performance boosting using an in-network load balancer.

- In the generalized system, *P4Hauler*, we introduce an on-the-fly configurable design, meaning the network operator can enable the appropriate load-balancing mechanism according to running applications.

- We create prototypes of *P4Mite* and *P4Hauler* using a testbed that includes a Tofino switch, traditional servers, and accelerators. Additionally, we assess the performance of all systems and compare them with similar load balancers. To promote reproducibility and further exploration, the source codes are available in [15, 16].

- *P4Mite*'s quantitative results prove that the accelerators, like SmartNICs, have considerable computing power, and the load balancer should utilize the capacity wisely. Moreover, The evaluation results validate the superior performance of *P4Hauler*'s policies in terms of flow completion time and throughput, while also ensuring per-connection consistency.

- Finally, *P4Wise* shows that learning-based policy selection effectively reacts to dynamic network behaviours while balancing loads among various computing resources.

1.3 Thesis Organization

The remaining chapters of the thesis are structured as follows: In Chapter 2, we provide the background information and present relevant findings that conform to the basis for our work. Subsequently, in Chapter 3, we examine related studies and their connection to our research. Once we have acquired sufficient knowledge, we proceed to elucidate the design, implementation, and experimental outcomes of *P4Mite* in Chapter 4. Similarly, in Chapter 5, we elaborate on our extended load balancer, *P4Hauler*, following the same structure. We present *P4Wise* in Chapter 6. Finally, we engage in conclusions and future work of our research in Chapter 7.

Chapter 2

Background and Motivation

This section presents the necessary background to understand our contribution along with the motivation for designing *P4Mite*, *P4Hauler*, and *P4Wise*.

2.1 Software-Defined Networking

Software-defined networking (SDN) is a new networking paradigm that introduces flexibility, scalability, and adaptability to network infrastructure by separating the control plane and data plane in network devices [17]. The control plane (or controller) serves as the intelligence behind SDN, while the data plane executes simple operations dictated by the controller. Specifically, the control plane assumes the role of network observer, possessing comprehensive system-wide information. This vantage point empowers it to make informed decisions regarding traffic management in the network. Conversely, the data plane, also referred to as the forwarding plane, pertains to the network devices under the controller's jurisdiction, which are responsible for the transmission of network traffic.

The key principle underlying the data plane is to ensure simplicity and speed. Typically, the data plane resides within hardware components responsible for forwarding packets, such as switches and routers, optimizing their performance [17, 18]. OpenFlow is the de facto protocol that enables communications between data and control planes. OpenFlow has undergone thorough examination in academic research and garnered substantial adoption within the industry [19]. OpenFlow 1.5 is the latest version at the time of writing this thesis and includes support for dozens of actions on network traffic, such as Output, Drop, and SetField. However, these actions are not mandatory for packet processing [20]. Additionally, the specific set of actions can vary between different OpenFlow versions, and there is potential for introducing new actions in future releases. SDN has been successfully adopted both in wired [21, 22, 23, 24, 25, 26, 27] and wireless [28, 29, 30, 31, 32, 33, 34] networking to offer better performance, agility, and scalability.

2.2 Programmable Switches in the Data Plane

Traditional switches typically execute a predefined set of operations. With the introduction of programmable switches, however, network owners can configure the switches to perform various functions by utilizing domain-specific languages and running programs in the data plane model [35]. There are different programming models for packet processing, such as data flow abstractions and the protocol independent switch architecture (PISA) [36]. Compilers are responsible for generating compatible programs for each mode. In this research, we employ the PISA architecture, as depicted in Figure 2.1. In this model, the switch parses incoming packets and applies a series of operations based on *match+action* tables. These tables can be programmed by the network owner to implement different functionalities.

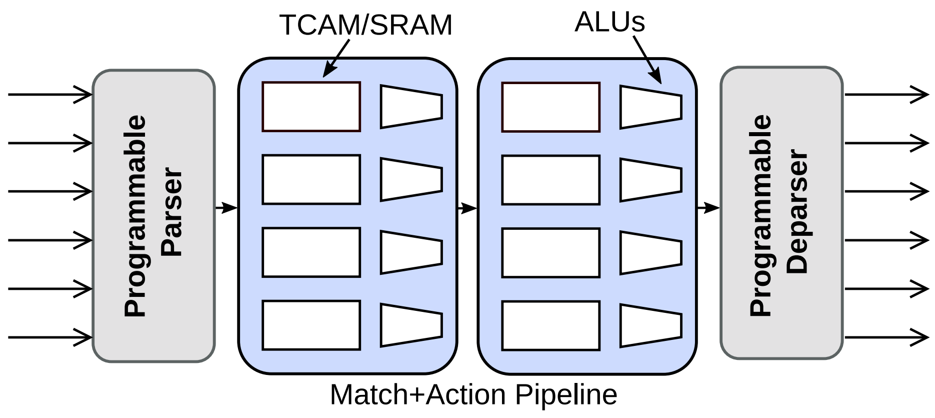


Figure 2.1: PISA Architecture.

Programming Protocol-independent Packet Processors or P4 is a domain-specific language designed for data plane programming. P4 is also distributed as an open-source and non-profit code supported by P4 Language Consortium. P4-14 and P4-16 are two versions introduced in 2014 and 2016, respectively. Given Figure 2.2 [37], the P4 compiler generates the required Runtime file for the target control and data plane, defining the pipeline. While the packets go through the pipeline, the program extracts the headers of layers, applies *match+action*, and finally encapsulates the packets for retransmitting. In this pipeline, the programmer can change the information of packets to implement different protocols and applications.

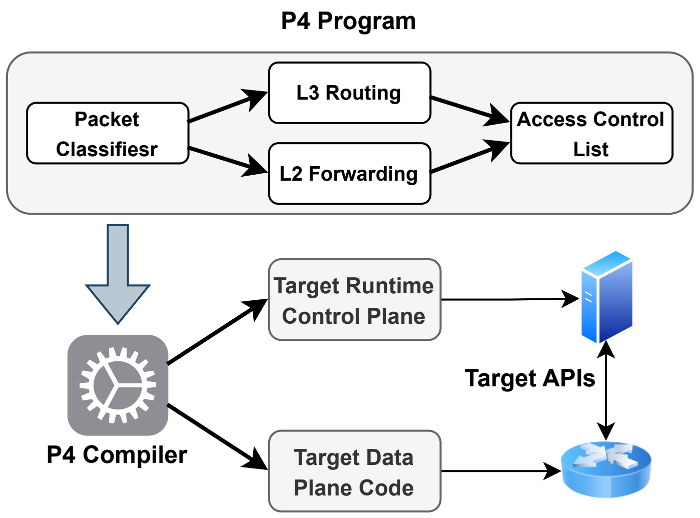


Figure 2.2: P4 Program Architecture.

In *P4Mite*, *P4Hauler*, and *P4Wise*, we employ a programmable switch executing an accelerator-aware load balancer inside the network getting updates from different devices in the network. While the load balancer keeps the status of existing flows, it dispatches the new requests to the best destination (e.g., fastest CPU). A programmable switch is a worthy option for deploying the load balancer because of three reasons:

1. The ALU in the switch is capable of performing the load balancing duty efficiently at the line rate of link (processing packets at Tbps).

2. The resource on the end servers will be free to process client's requests instead of involving in load balancing.

3. Our switch-based load balancers avoid sending too many requests to the end servers by load balancing at a fine granularity.

2.3 Programmable accelerators

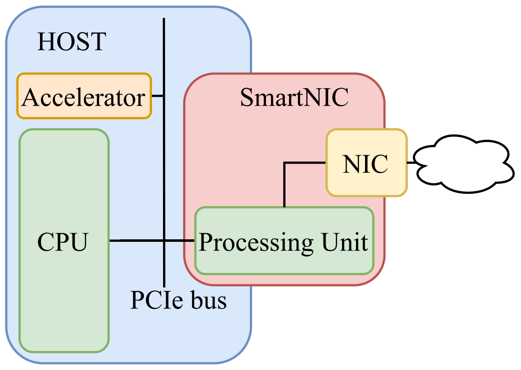
Programmable accelerators are reconfigurable devices employed in computer systems to enhance the speed of specific tasks. Typically, these accelerators are connected to the server's CPU via Peripheral Component Interconnect Express (PCIE) slots [38]. Examples of such accelerators include SmartNICs, SSDs, and GPUs. Noteworthy instances include Google's AutoML, which utilizes Tensor Processing Units (TPUs) [39], and Microsoft Azure's machine learning services, which employ FPGAs [40]. Many accelerators possess their dedicated operating systems, such as Linux, as seen in Mellanox BlueField SmartNIC and Broadcom Stingray [41, 42]. This characteristic grants them programmability and accessibility through the standard TCP/IP stack via IP over PCIE tunnelling [43]. Alternatively, some accelerators, lacking an operating system, rely on their firmware to communicate over TCP/IP. The most commonly used method to communicate with these non-TCP/IP-supported accelerators, including GPUs and programmable SSDs, is Remote Direct Memory Access (RDMA) [44].

Accelerators are frequently employed to offload applications or specific components of applications, thereby enhancing their overall performance in comparison to running them solely on a standard CPU [43, 45, 46, 47]. System designers offload different applications to programmable accelerators, including machine learning training [39] and inference [48], databases [49], and web servers [50]. While accelerators offer performance advantages for these applications, running them without proper consideration for load offloading can result in performance degradation. To address this, it is essential to design capability-aware load balancing for accelerators when utilizing such computing resources, thus mitigating potential issues arising from inappropriate load distribution.

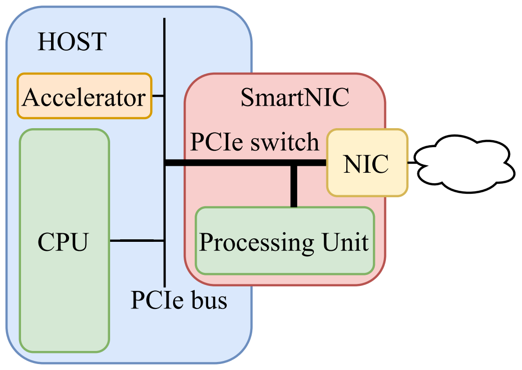
2.3.1 SmartNICs

In this study, while our designs, which we will discuss later, are versatile and can be applied to various accelerators, we have chosen to utilize a Mellanox SmartNIC as an accelerator to assess our prototype. This section provides a brief introduction to SmartNICs. Subsequently, in section 2.4, we will present some quantitative data to demonstrate the impact of this type of accelerator on our research. Noteworthy that by using an accelerator with limited resources, we stress our load balancers for making decisions more frequently and observing more accurate outcomes.

Taking a look at Figure 2.3, we can observe two distinct architectures employed in SmartNICs. The first architecture, depicted in 2.3a, positions the SmartNIC's processing unit directly on the networking path. In contrast, the second architecture,



(a) On-path Architecture



(b) Off-path Architecture

Figure 2.3: Two architectures for SmartNICs.

illustrated in Figure 2.3b, places the processing unit off the path and establishes communication with the NIC and host via the PCIE. Various options are available for the SmartNIC's processing unit, including off-the-shelf CPUs, FPGAs, and ASICs. When it comes to general-purpose CPUs, the majority of them are ARM-based or MIPS-based, which are comparatively less powerful than the x86 CPUs found in hosts.

2.4 Roofline Benchmark

To reveal the computational capabilities of our SmartNIC, we utilized a tool called *roofline* [51]). This tool evaluates the performance of the embedded CPU in comparison to the CPUs of the host by running a kernel module on each device. By employing *roofline* tool, we can assess the CPU performance at various levels within the memory hierarchy.

The results presented in Table 2.1 indicate the maximum number of floating-point operations per second executed by each CPU, measured in Giga Floating-point OPeration per second (GFLOP/s). It is observed that the host performs five times as many floating-point operations compared to the SmartNIC's CPU. In other words, the SmartNIC has the potential to contribute an additional 20% of computational power to the system. Based on these findings, we propose that *P4Mite*, *P4Hauler*, and *P4Wise* can effectively utilize a SmartNIC in situations where the associated host is overwhelmeed. In short, we can enhance the performance by offloading a portion of the workload to the SmartNIC.

Table 2.1: Roofline's Results.

Device	**CPU Specifications**	**GFLOP/s**
Host (x86)	Intel(R) Xeon(R) Silver 4210R with 10 cores @ 2.4GHz	91.0
SmartNIC (ARM)	Armv8 A72 CPU with 16 cores @ 2.0GHz	17.6

2.5 Load Balancing

In today's digital landscape, applications stand as the central driving force behind the success of businesses and services, underlining the pivotal role of their performance. As applications increase in complexity and usage, ensuring consistent and reliable performance becomes a significant challenge. This is where load balancing steps in as a vital strategy. Load balancing is a method utilized to distribute network or application traffic across numerous servers, resources, or pathways [52, 53]. Its core purpose is to wisely utilize the resources, enhance fault tolerance, and, most significantly, elevate the overall performance and responsiveness of applications.

To achieve these objectives, both academia and industry have explored various types of load balancers. Load balancers can function at different layers of the network stack, such as L4 and L7 [54, 55], and they can be implemented as either software or hardware solutions [56].

With the recent advancement of programmable switches, a growing trend is emerging, aiming to delegate load balancing tasks to the data plane. In the context of P4 and load balancing, an in-network load balancer functions directly within the network infrastructure, efficiently distributing incoming traffic across multiple servers or pathways. It possesses the capability to make load balancing decisions based on various factors, including source IP address, destination IP address, or application type. An in-network load balancer implemented on P4 switch offers distinct advantages, such as optimizing resource utilization and processing packets in real-time at the line rate.

2.6 Reinforcement Learning

Unlike supervised or unsupervised learning, which rely on the collection of labeled or unlabeled data, respectively, Reinforcement Learning (RL) is a subfield of artificial intelligence that an agent interacts with the environment. Its primary goal is to achieve the maximum cumulative reward in an ever-changing environment[57].

RL excels in adapting to environmental dynamics, maintaining optimal strategies through a series of interactions and trial-and-error processes. RL algorithms can be usually categorized into two main approaches, model-based and model-free [58]. 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. Probabilistic [59] and neural network-based [60] are two examples of model-based RL. In contrast, model-free reinforcement learning comes into play when the environment cannot be entirely modeled or predicted, as is the case with intricate and unpredictable scenarios like load balancing. A few notable examples for model-free RL are Q-Learning [61], Deep Q-Network DQN [62], and SARSA [63].

In the realm of Reinforcement Learning (RL), an agent engages with its environment, perceiving the current state and choosing actions based on a policy. Subsequent to each action, the agent receives a reward, aiming to maximize its cumulative rewards over time. In our specific context of load balancing tasks, the environment constitutes the entire network system comprising interconnected elements. The learning agent interacts with this system, dynamically evolving to optimize load balancing among computing resources. The agent's objective is to continually refine its decisions, strategically adapting to the changing demands of the system, all with the ultimate goal of achieving optimal load distribution.

Chapter 3

Related work

We divide this chapter into two sections. In the Section 3.1, we elaborate in-network load balancers with and without using AI/ML, whereas Section 3.2 discusses task offloading into programmable hardware.

3.1 In-network Load Balancing

Load balancing on programmable network devices is a crucial mechanism that enhances network performance and optimizes resource utilization. These devices, such as programmable switches (e.g., Tofino switch) and SmartNICs (e.g., Bluefield and Stingray) equipped with programming capabilities and allow for efficient distribution of incoming traffic across multiple network paths or splitting the load among multiple servers. By distributing load and traffic, load balancing prevents network congestion and bottlenecks, ensuring that the system operates optimally. Programmable network devices enable the flexibility to customize load-balancing algorithms according to specific network requirements and adapt to changing conditions in real time. This empowers network administrators to achieve efficient utilization of network resources, maximize throughput, and provide a seamless experience for end-users.

For connection consistency on layer-4 load balancers, a hash derived from a 5-tuple is stored in the load balancer as shown in Figure 3.1. The 5-tuple consists of the source IP, source port, destination IP, destination port, and protocol type [64]. The hash also identifies an address revealing if a connection is using it for storing a destination IP.

3.1.1 In-network Load Balancing without Machine Learning

In the last decade, various load balancers within the network have emerged as alternatives to traditional load balancers. Table 3.1 lists some well-known in-network load balancers. In the table, RA and AA stand for **R**esource **A**wareness and **A**cceleraor

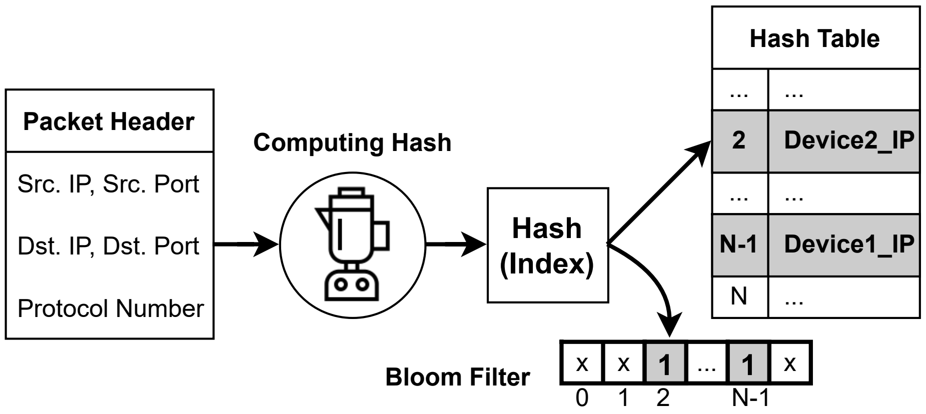


Figure 3.1: Using Hash for Indexing.

Awareness, respectively. To address memory limitations posed by the 5-tuple, SilkRoad [6] adopts a programmable switch and calculates a hash function to support plenty of connections. Loom [65] employs a similar methodology, focusing on scaling per-connection consistency with the aid of a programmable switch. It utilizes multiple bloom filters to compress the connection states. Alternatively, Cheetah [10], a different in-network load balancer, stores information in the packet headers instead of switches. It is important to note that these load balancers do not take accelerators into account, resulting in load distribution at a per-server level.

Hsu *et al.* [66] propose an adaptive weighted traffic splitting mechanism that dynamically adjusts the distribution of incoming traffic across multiple paths based on network conditions. A programmed data plane monitors network metrics such as link utilization and latency, and it calculates appropriate weights for each path to optimize traffic distribution. The authors also introduce an adaptive algorithm that continuously adapts the weights based on real-time measurements, ensuring efficient utilization of available network resources.

In addressing the challenge of supporting a significantly large number of connections, Tiara [9] adopted an innovative approach. To offload the load balancing process, Tiara introduced a 3-tier method. In the first tier, an active switch known as T-Switch utilizes Equal-Cost Multipath (ECMP) to distribute the load across Tiara SmartNICs (T-NIC) located in the second tier. Tiara leverages the programmability of SmartNICs for target server selection. In cases where a SmartNIC is not responsible for a particular connection, the packet is forwarded to the Tiara Server. The Tiara

Server, equipped with the ability to select the appropriate target server, then employs the SmartNIC and switch to forward the packet accordingly. Although these load balancers can assign tasks at a finer granularity, such as per CPU core, they still rely on the accelerators themselves, such as SmartNICs, to perform load-balancing decisions. This dependence on accelerators for load-balancing decisions can result in the inefficient utilization of valuable computing resources. Therefore, an alternative approach is necessary to optimize resource utilization and enhance overall performance.

Several resource-aware load balancers have been developed to optimize performance and resource utilization. One example is CrossRSS [67], which incorporates CPU-awareness into its stateful load balancing mechanism. In CrossRSS, the SmartNIC is responsible for computing the hash and selecting the appropriate core. Another resource-aware load balancer is Charon [7], which adopts a passive approach by receiving updates from agents running on the server. Charon utilizes an FPGA-based SmartNIC to select the most suitable server for load balancing. Like the last two mentioned load balancers, SHELL [68] is another resource-aware load balancer that employs the accelerators (i.e., SmartNICs) for load balancing instead of running the application (or a part of them) on accelerators for performance enhancement

R2P2 [69] proposes a load balancer, which is deployable in our design, applying join-bounded-shortest-queue (JBSQ) policy for Remote Procedure Calls (RPC) by employing a Tofino switch. RachSched [5] also introduces a load balancer applying two-layer schedulers on a programmable switch. After selecting the server in the rack, the second layer picks the appropriate intra-server worker. Although we employ a similar hierarchical design for load balancing, the second layer of our load balancer offers reconfigurable policies for accelerator-aware decisions.

In the study conducted by Cui *et al.* [54], they evaluate the performance of an ARM-based SoC SmartNIC in running a lightweight load balancer. The authors provide insights into which load balancing tasks should be executed on the server and which ones should be offloaded to SmartNICs. Notably, their findings demonstrate the benefits of performing asymmetric and symmetric cryptography on the server and SmartNIC, respectively, highlighting the optimal distribution of cryptographic tasks between the server and the SmartNIC.

Table 3.1: In network load balancers without AI/ML

Load Balancer	**RA**	**AA**	**Deployment**
SilkRoad [6]	✗	✗	In a Programmable Switch
Loom [65]	✗	✗	In a Programmable Switch
Cheetah [10]	✗	✗	In a Programmable Switch
WCMP-based [66]	✗	✗	In a Programmable Switch
Tiara [9]	✗	✗	Switch+SmartNIC+Server
CrossRSS [67]	✓	✗	In a SmartNIC
Charon [7]	✓	✗	In a SmartNIC
SHELL [68]	✓	✗	In a SmartNIC
R2P2 [69]	✓	✗	In a Programmable Switch
RachSched [5]	✓	✗	In a Programmable Switch

3.1.2 In-network Load Balancing with Machine Learning

Here, we study state-of-the-art regarding utilizing AI/ML for load balancing. Offloading applications on network devices, including AI/ML, is a challenging problem, and we illustrate this difficulty and conducted research in this area in Section 3.2.

Table 3.2 lists a few load balancers that employed AI/ML for load balancing. Learned Load Balancing (LLB) [70] is a technique that 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. By analyzing factors such as network traffic, server capacity, and latency, the machine-learning model learns patterns and makes intelligent decisions regarding load distribution. This 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. Chang *et al.* adopt a SmartNIC to generate the weights as not only can they execute inference, but due to hardware performance predictability, it is feasible to estimate an upper bound for the latency for the load balancing process.

LBAS [71] presents a fast switch-based load balancer that considers the states of application servers. Unlike 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.

Load balancing is further compounded by the dynamic nature of network traffic and the growing diversity of workloads traversing the network. QCMP [72] 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 [73] 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.* [74] addressed the challenge of efficiently distributing traffic in data center networks with multi-rooted topologies to optimize bisection bandwidth. They introduced 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.

Reinforcement learning (RL) models present a robust alternative to manually designed heuristics within networked systems. However, they can exhibit unpredictable behavior, raising safety concerns. In [75], Mao *et al.* tackled this issue by proposing an approach that facilitates the safe deployment of RL models, enabling them to adapt to dynamic conditions. In a practical application, they assessed this approach within a real-world load balancing scenario. As a result, the load balancer was able to efficiently respond to abrupt workload changes while maintaining response time objectives.

Given that allocating resources across routes for cloud requests is a complex NP-hard problem [52], machine learning methods offer an efficient approach to address this issue. However, using supervised or unsupervised learning introduces a challenge since they rely on pre-defined datasets that may not encompass all scenarios and can be challenging to acquire. Therefore, reinforcement learning emerges as a more suitable solution because it learns within the environment through trial-and-error processes. However, none of the previous studies that utilize machine learning have taken accelerator awareness into account for 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.

Table 3.2: In-network load balancers utilizing AI/ML

Load Balancer	**RA**	**AA**	**Deployment**
LLB [70]	✓	✗	In a SmartNIC
LBAS [71]	✓	✗	In a Programmable Switch
QCMP [72]	✓	✗	In a Programmable Switch
CrossBal [73]	✓	✗	In a Programmable Switch
RWCMP [74]	✓	✗	Simulation

3.2 Task offloading to Programmbale Devices

Rather than load-balancing, researchers have explored the potential of offloading various applications onto programmable devices. In the subsequent subsections, we have organized the projects into two categories. In the first one, we elaborate on task offloading to accelerators; then, we present task offloading to programmable switches.

3.2.1 Task Offloading to Accelerators

Numerous projects have studied the capabilities of SmartNICs to drive various applications. These endeavours encompass a wide range of domains. For instance, authors in Xenic [76], and LineFS [77], have leveraged SmartNICs to enhance the performance of distributed applications. Real-time analytics projects like iPipe [45] have also benefited from SmartNICs. Furthermore, system designers have empowered micro-services architectures by SmartNICs, exemplified by the work done in E3

[46]. SmartNICs have proven instrumental in accelerating machine learning and deep learning applications as well, demonstrated by projects such as N3IC [78], Brainwave [40], and SpikeOffload [79]. Similarly, GPUs have been widely employed to bolster the performance of learning-based applications [80]. In addition, TPUs and programmable SSDs have found utility in the field of AutoML [39] and [81].

Although it may not be feasible in every instance, all the research projects mentioned have made modifications to a particular application to enhance its performance on their SmartNICs. In some cases, there is a compromise between various parameters, such as in the case of N3IC [78], where reasonable latency is achieved at the expense of accuracy.

Zhao *et al.* explored the potential of emerging SmartNICs for running security applications, particularly those relying on cryptographic operations [82]. It highlights that SmartNICs can significantly benefit from architectural enhancements like cryptographic instructions and hardware accelerators to match server performance in crypto-workloads. However, data movement between the SmartNIC and crypto-hardware cores can introduce overhead, especially for short-lived tasks. SmartNICs, positioned closer to client devices than server CPUs, can accelerate crypto-based functions, but the advantages may be diminished by data-intensiveness or the presence of multiple no-ncrypto tasks in the application.

Xing *et al.* developed a framework that highlights the utilization of the Linux network stack on both host and SmartNIC CPUs. Their research demonstrates how the use of SmartNICs can impact critical performance metrics, including E2E latency, throughput, and multi-core scalability. These effects are attributed to factors such as the architecture of SmartNICs and their computational capabilities [47].

3.2.2 Task Offloading to Programmbale Switches

Lastly, programmable switches have served as versatile platforms facilitating a wide array of network services and applications. Notable examples for the application include NetCache [83] and NetGVT [84] supporting distributed applications, Net-Pixel [85] enabling ML/DL tasks, PoirIoT [86, 87] fortifying security, and RedPlane [88] ensuring fault tolerance. Focusing on machine learning, Parizotto *et al.* [89] reviewed the completed projects in this domain in a systematic review. Collectively, these projects highlight the diverse spectrum of applications and services that have harnessed the capabilities of programmable switches.

Chapter 4

P4Mite

In this chapter, we introduce a cutting-edge load balancing system called *P4Mite* that considers the presence of accelerators. To our knowledge, *P4Mite* is the first load balancer that works at a per-accelerator granularity. In short, *P4Mite* effectively combines mechanisms for load balancing between servers (referred to as *inter-server balancing*) such as ECMP, connection ID hashing [64], and power-of-k-choices [90], with a task distribution at the accelerator level within each server (known as *intra-server balancing*). The latter involves intelligently distributing connections to the CPU or available server accelerators like SmartNICs and GPUs, based on accurate estimations of their respective loads.

An outstanding feature of *P4Mite* is its full deployability on programmable switches, enabling it to handle massive connections with exceptional throughput and minimal latency. This deployment flexibility takes advantage of the programmability inherent in the switches, allowing *P4Mite* to adapt and scale effortlessly across diverse network environments.

Furthermore, *P4Mite*, like related work such as SildRoad [6], and Loop [65], places significant emphasis on maintaining per-connection consistency (PCC) by carefully managing a highly optimized connection table. This table keeps track of existing connections, ensuring that each connection receives consistent treatment throughout the load-balancing process.

In the following subsections, we will study the challenges that *P4Mite* faces in Section 4.1. Next, we dig into *P4Mite*'s design, its data plane, and implementation in Sections 4.2, 4.3, and 4.4, respectively. Afterward, we comprehensively assess our prototype and report quantitative results in Section 4.5. Finally, we conclude and summarize this chapter in Section 4.6.

4.1 Challanges in ***P4Mite***'s Design

There are two challenges in designing *P4Mite*.

4.1.1 Load Balancing in a Diverse Environment

Typically, servers within a pool are expected to exhibit uniform characteristics [91, 92]. However, the accelerators present in these servers introduce heterogeneity into the environment. In other words, CPUs within hosts and their accelerators like SmartNICs and GPUs possess distinct architectures. Even two SmartNICs can yield disparate outcomes due to their differing designs [47, 93]. Unfortunately, policies that perform well in homogeneous setups often demonstrate subpar performance when confronted with heterogeneity. To overcome this challenge, *P4Mite* takes a proactive approach by monitoring all devices, including servers and accelerators, and gathering relevant statistics such as CPU utilization and request processing latency. By analyzing this information, *P4Mite* develops a comprehensive understanding of the status of each device, enabling it to make informed decisions about the optimal destination for dispatching requests. Furthermore, our policies can prioritize powerful resources (e.g., beefy host CPU), allowing them to handle a greater number of requests compared to less capable resources (e.g., smartNIC CPU).

4.1.2 Processing Large Number of Concurrent Flows

Modern programmable switches face limitations in supporting numerous concurrent flows due to their restricted memory capacity, typically around 50-100 MB SRAM [6]. Consequently, prior research has proposed alternative solutions such as hybrid load balancer architectures, which involve hardware/software co-designs [9], or effective compression techniques. These compression approaches aim to reduce memory usage by storing connection hashes instead of the complete 5-tuple (source MAC, destination MAC, source IP, destination IP, protocol) or utilizing indirect VIP-to-DIP mappings [6]. In load balancing context, the VIP and DIP refer to Virtual IP and Direct IP, respectively. These terms are commonly employed in load balancing solutions aimed at distributing incoming packets or requests among multiple end hosts. To be more precise, the VIP represents the IP address utilized by clients to send their requests, whereas the DIP pertains to the address within the back-end network. It is the responsibility of the load balancer to map the VIP to the appropriate DIP.

In the case of *P4Mite*, an additional level of complexity arises in mapping connections to computing units rather than servers, requiring a more detailed and fine-grained process. While *P4Mite* still relies on a connection table to maintain the state of each connection, ensuring per-connection consistency (PCC), we minimize the memory footprint through a combination of data compression techniques for efficient policy representation. Specifically, our system employs hashing and bitmaps to store connection and accelerator states, respectively. Additionally, we utilize a two-step process for VIP-to-DIP mapping, breaking it down into mapping a VIP to a server code and then mapping the server code to a DIP. This indirect mapping approach complements the technique described in [6], which also utilizes indirect mapping to reduce the size of the connection table. The performance of our compression approach is thoroughly evaluated in Section 4.5.5, where we assess its effectiveness and resource utilization.

4.2 *P4Mite*'s Overview

Figure 4.1 provides an overview of the architecture of *P4Mite*. The system comprises a programmable switch, a controller, and a set of agents running on servers and accelerators. In *P4Mite*, the connection state is maintained at the programmable switch, enabling *stateful* load-balancing. Furthermore, requests are forwarded exclusively within the data plane, allowing advanced load-balancing policies while minimizing any impact on flow performance.

Additionally, we assume that each server executes a specific application, which aligns with the usual practices observed in contemporary data centers [94]. Hence, we expect both processing unit, including the server CPU and the accelerator, run a single application such as data analytics, VPN tunnelling, or machine learning inference. It is worth noting that although the rack can accommodate multiple distinct services, our focus remains on the dedicated ones.

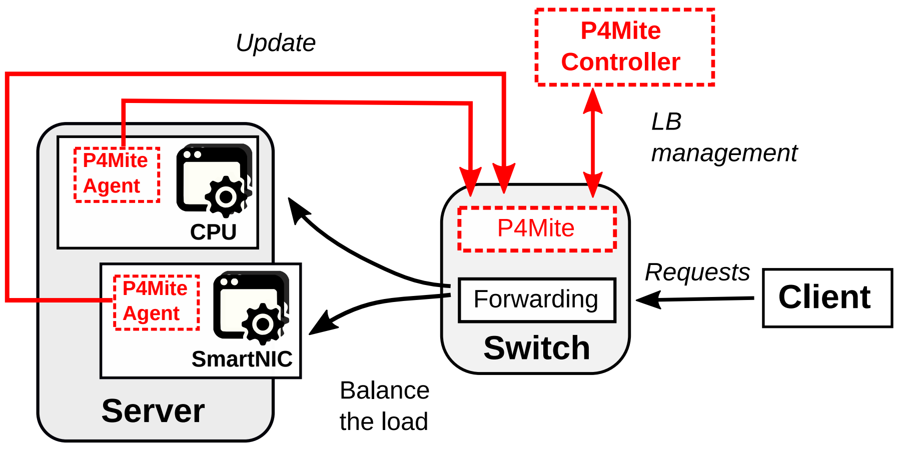


Figure 4.1: *P4Mite* overview.

4.2.1 Prgrammable Switch

The core component of *P4Mite* is the programmable switch in data plane, which seamlessly integrates with the TCP/IP network stack. It performs load balancing at the transport layer, eliminating the need for any modifications on client-server applications. It is important to note that while certain accelerators like FPGAs may require the re-implementation of x86 applications (e.g., in VHDL) to run on their specific hardware, this is not necessary for *P4Mite*.

Furthermore, in our system, each accelerator is assumed to have its network address, and packet delivery to the appropriate accelerators is facilitated through PCIe switching [95, 96, 97]. In certain scenarios, the accelerators may operate their own operating systems [41, 42], allowing resources to be accessed via their respective IP addresses over the network.

4.2.2 Controller

Rather than general controllers' functionality in SDN, the role of the *P4Mite* controller is to effectively enforce the intended load balancing policy on the switch by installing the necessary forwarding rules. An example is to prioritize the CPU over the SmartNIC when both have available resources. Additionally, the controller is responsible for updating the switch configuration whenever changes occur within the server pool, such as the addition/removal of servers or accelerators.

By entrusting all connection handling to the switch, we ensure that the controller does not become a bottleneck in the system. Moreover, the controller operates in a non-intrusive manner, allowing other protocols and network functions to function without interference. Its sole focus lies in managing its state, and the load-balancing structures within the data plane.

4.2.3 Server Agents

To facilitate its operations, *P4Mite* deploys an agent on each processing unit, whether it be a CPU or an accelerator. These agents are responsible for measuring various metrics at both the system and service levels, such as CPU utilization and request processing latency. By collecting these statistics, the agents transmit the updates to the programmable switch using a dedicated L4 port. Instead of embedding the updates within response packets, we opt to use separate packets for updating due two reasons:

1. If we use separate packets for updates, the servers and accelerators send the response packets directly to clients without traversing the load balancer. This bypassing of the load balancer leads to more efficient and streamlined communication.

2. Using separate packets allows the agents to update the switch more accurately. In scenarios where numerous requests are overwhelming a particular resource, the agents can immediately update the switch while the resource is still actively processing operations. Otherwise, the switch must wait until finishing the tasks.

The agents send updates to the switch based on a threshold mechanism to minimize the overhead. Put differently, the agents send an update packet only when the threshold condition is triggered. Additionally, for the sake of simplicity, the agents currently transmit a binary value (e.g., indicating busy or available) to the switch.

4.3 ***P4Mite*'s Data Plane Design**

Figure 4.2 illustrates the data plane layout of *P4Mite* at the programmable switch. When a packet is received, the switch examines whether it originates from an agent for updating accelerators' statuses or a client has sent the packet. In the case of an agent packet, it carries a key and a value indicating which entry in `AccelState` table must be updated. The key represents a unique identifier for a server (`S-code`), while the value is a bitmap that indicates the states of the server's accelerators and CPUs (shown by red dotted arrows). For example, a bitmap of "10" signals that the CPU is busy, but the first accelerator (e.g., SmartNIC) is available to handle new requests. The length of the bitmap corresponds to the number of accelerators in a server plus one, while the size of the register array is proportional to the number of servers in the pool. By utilizing bitmaps and identifiers instead of actual IP addresses, we can reduce memory usage in the switch. Additionally, both the server code and status information are encapsulated within an update packet by a *P4Mite* agent.

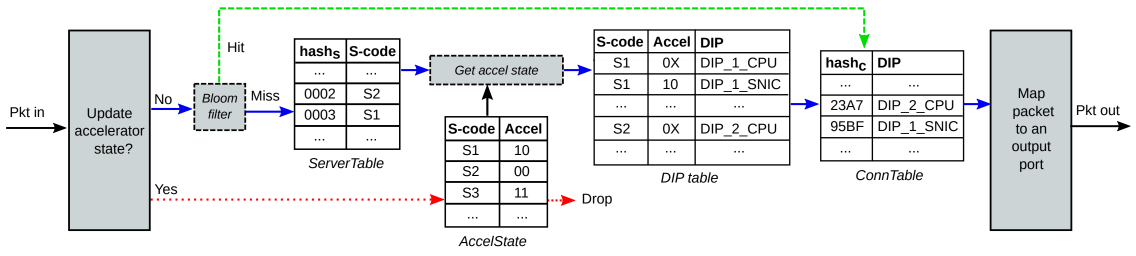


Figure 4.2: *P4Mite*'s data plane layout.

On the other hand, when clients' packets traverse the network, they pass through a bloom filter that determines whether they belong to a new or already registered connections. This bloom filter enables *P4Mite* to read and update the connection status without involving the controller. While the possibility of false positives exists due to hash collisions [98], their impact is negligible as long as the filter size is sufficiently large. Alternative structures such as Cuckoo filters [99] can be implemented for the future work.

If a packet originates from a new connection (bloom filter `Miss`), *P4Mite* computes a hash using the five tuples extracted from the packet's header to select the destination server (`ServerTable`). It is important to note that any desired load balancing policies, such as ECMP [100] or WCMP [101], can be implemented using the `ServerTable`. Once the server decision is made, *P4Mite* consults the `AccelState` table to determine the appropriate CPU or accelerator within the selected server.

After selecting a server, *P4Mite* retrieves the server's and accelerators' load state and uses this information to make the final determination of the packet's destination

(`DIPTable`). Each entry in the `DIPTable` can direct packets to a different accelerator, allowing network operators to deploy various intra-server load balancing policies. We explore the performance of different intra-server load-balancing policies in Section 4.5.3. To maintain per-connection consistency [6], *P4Mite* stores its load balancing decisions, which include the connection state, in a connection table (`ConnTable`). This ensures that all packets from a given connection are consistently delivered to the same destination, even if the server pool or load balancing policy changes.

For subsequent packets from existing connections, the connection state table (bloom filter `Hit`) can be directly matched, saving processing cycles in the network device. Finally, when the switch sends the packets from servers to clients, it could replace the associated DIP with the corresponding VIP. We have not shown it in Figure 4.2 for the sake of simplicity.

4.4 *P4Mite* Implementation

In this section, we present our prototype implementation details for *P4Mite*. Our source code is publicly available at [16] and can be used by other researchers.

4.4.1 *P4Mite* Controller and Switch

We have developed the *P4Mite* switch and its controller using a combination of P4-16 and Python. The total codebase for both modules consists of approximately 350 lines. We employ an array of 50,000 registers, each containing a 16-bit index for the `ConnTable`. To compute hashes, we utilize CRC16. Similarly, an array of registers builds the `ServerTable`, with its size determined by the desired number of servers. The server state, represented by `AccelState`, is retrieved as packet metadata and used for matching in the `DIPTable`, which is a *match+action* table based on exact matching. The switch utilizes standard IPv4 tables for packet forwarding. Lastly, our code is targeted at the Tofino Native Architecture (TNA) model.

4.4.2 *P4Mite* Agents

We developed our agents using Python, comprising approximately 150 lines of code. These agents utilize `tcpdump` to monitor the packets received and sent by the host and measure the processing time as the decision metric. To ensure efficient processing, we run `tcpdump` in "immediate-mode", enabling the agents to analyze mirrored packets in real time.

The structure of our agents is designed to be modular, allowing for easy extension to monitor additional metrics such as CPU utilization (e.g., using `top`) or network statistics (e.g., using `iftop`) in future work. Finally, our agents encapsulate all the relevant information into UDP packets, which are then sent to the switch using a predefined port for communication.

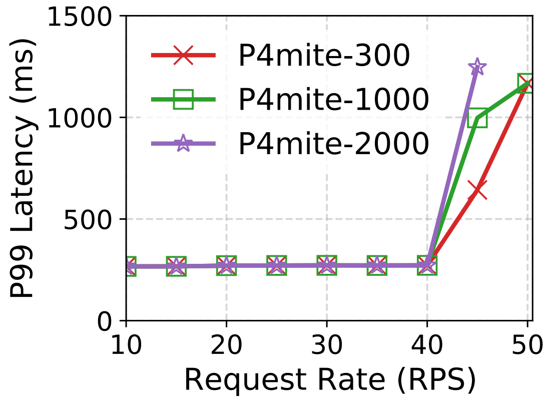
4.5 *P4Mite* Evaluation

Firstly, we provide an overview of the experimental setup in Section 4.5.1. Next, we detail the experiments conducted to evaluate the performance of *P4Mite* and present the corresponding results in Sections 4.5.2 and 4.5.3. Additionally, we compare our solution to alternative load balancer designs in Section 4.5.4. Finally, we assess the resource consumption of *P4Mite* in a real-world data center scenario in 4.5.5.

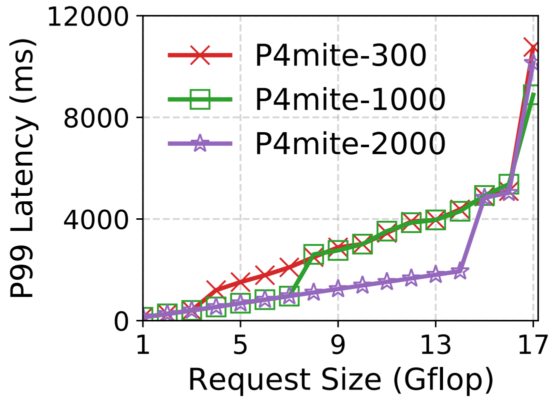
4.5.1 Experimental Setup

We conducted the experiments on a testbed comprising two hosts connected by a Wedge 100BF-32X 32-port programmable switch equipped with a 3.2Tbps Tofino ASIC [14]. Both hosts have identical specifications, featuring an Intel(R) Xeon(R) Silver 4210R CPU @ 2.4GHz, with 10 cores and 32GB memory. One of the hosts serves as the client, while the other runs server applications. The server host mounts a dual-port SFP28, PCIe Gen3.0/4.0 x8, BlueField(R) G-Series SmartNIC, which includes 16 cores, 16GB on-board DDR4 RAM, and enabled crypto accelerators.

In our experiments, we prioritized the host CPU for packet balancing and configured its agent to provide reports based on a combination of thresholds and time intervals, specifically designed for highly computation-intensive applications. This configuration was necessary to address the significant performance gap between the CPU and the SmartNIC in these scenarios, where a single request could overwhelm the SmartNIC. In such cases, we configured the CPU agent to immediately send a second load status report (within 5 milliseconds) upon triggering a threshold, to avoid



(a) Evaluating request rate



(b) Evaluating request size

Figure 4.3: Microbenchmarking results

delays in receiving reports from the SmartNIC. It is important to note that the agent adds approximately 5% load on the server's CPU.

4.5.2 Microbenchmark

We conducted experiments using synthetic workloads to analyze the impact of request rate and request size on the performance of *P4Mite*. For this purpose, we developed an application capable of performing varying amounts of floating-point operations and a client application capable of generating different numbers of requests per second (RPS). Each request triggers a specific level of computation on the server, intended to keep the server busy and overload its CPU usage. Figure 4.3 illustrates the results obtained from the synthetic workloads. We performed two types of experiments using micro-benchmarks. Firstly, we evaluated the behavior of our solution by varying the request rate. Secondly, we assessed the behavior of our solution by varying the request sizes. In both sets of experiments, we tested three different thresholds for the agents to send a load report. We will now delve into a detailed discussion of these experiments.

Request Rate

Figure 4.3a depicts the results of the request rate evaluation for *P4Mite*. In this experiment, we configured the client to trigger 2 GFlop on the server with each request, while varying the request rate. On the server side, the agent used three different thresholds to determine when to trigger the update in the switch to utilize the accelerator. We set three thresholds at 300ms, 1000ms, and 2000ms of processing time. The corresponding results for each threshold are labeled as *P4Mite*-300, *P4Mite*-1000, and *P4Mite*-2000, respectively. We gradually increased the request rate until the server started dropping requests. In our experiments, the client considered a packet lost if either the latency of a request was 10x higher than when the system was not overloaded or if the server dropped the packet.

We observe that all scenarios exhibit similar behaviour up to a rate of 40 RPS, considering the latency is smaller than the smallest threshold. However, at 45 RPS, *P4Mite*-300 demonstrates the best performance since its agent is triggered earlier compared to the other scenarios, allowing the switch to start sending requests to the SmartNIC sooner. Similarly, *P4Mite*-1000 achieves lower latency than *P4Mite*-2000 due to the earlier triggering of the agent. In fact, for *P4Mite*-2000, the agent is not triggered at all, resulting in a latency of approximately 1200ms. The maximum increase in latency is observed in *P4Mite*-2000, as the server becomes overloaded, causing multiple packets to wait in the queue for processing. The results highlight the importance of selecting a suitable threshold for the agent to achieve more satisfactory performance. It also provides evidence that when setting thresholds, it is crucial to consider values that are lower than the latency of the server when it is overloaded.

Request Size

Figure 4.3b describes the behaviour of *P4Mite* as we vary the request size. In these experiments, we fix the request rate at 5 RPS, and the request sizes range from 1 to 17 GFlop, maintaining the same configurations as in the experiments discussed in previous paragraphs regarding request rate assessment.

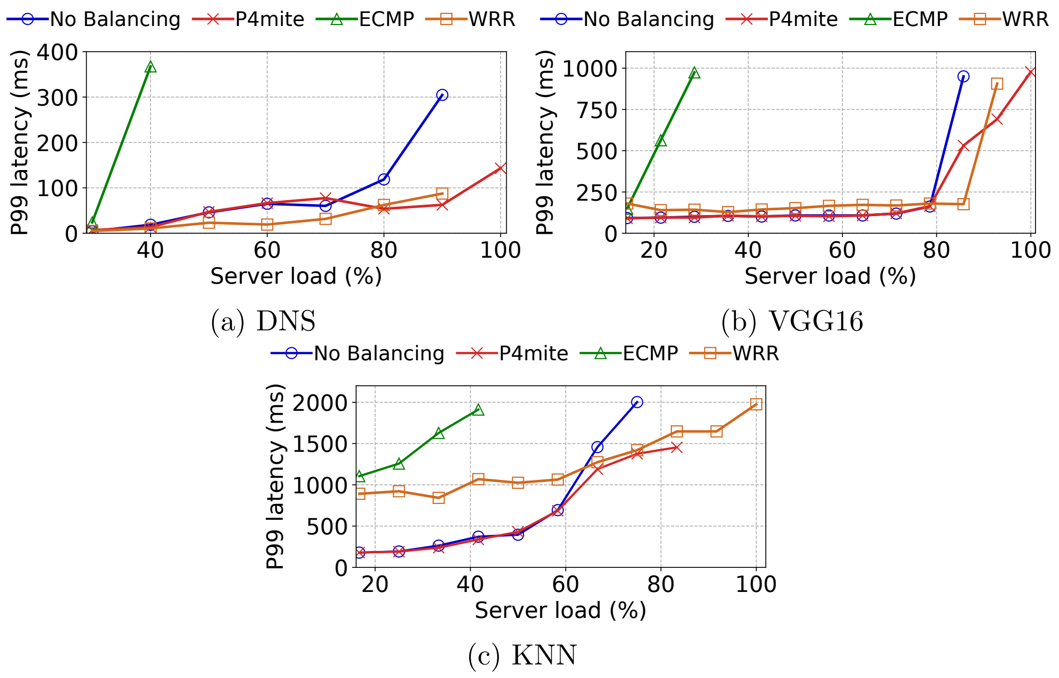
We observed that as the amount of computation increases, the latency also increases for all cases. For small request sizes, the differences between using various thresholds in *P4Mite* are negligible. For example, when each request involves 2 Gflop of computation, the latency is approximately 200ms across all scenarios. As we move to larger request sizes, such as 4 Gflop, the server's latency exceeds 300ms, triggering the agent in *P4Mite*-300. Similarly, at a request size of 8 Gflop, the agent in *P4Mite*-1000 is triggered. Once the agent in *P4Mite*-1000 starts sending reports to the switch, its behaviour becomes similar to *P4Mite*-300. However, since the server is not overloaded at these sizes, these two scenarios perform poorly because some requests are forwarded to the SmartNIC, even though the server could process them faster.

Considering a server capacity of approximately 90 Gflops, a request rate of 5 RPS, and a request size of 15 Gflop, the server's usage reaches 75 Gflops. The remaining capacity is allocated to operating system tasks and connection management. At this point, the system starts to become overloaded, resulting in a latency of 1800ms. In such circumstances, *P4Mite*-2000 outperforms the other scenarios significantly for request sizes larger than 15 Gflop, as it leverages the SmartNIC when the server is overloaded. Finally, we observe that with heavier requests, *P4Mite* also starts dropping packets, since both the server and the SmartNIC become overloaded. However, this behaviour is acceptable, as *P4Mite* can handle requests up to 18 GFlop before dropping packets, while a server-only solution drops packets at 16 Gflop.

4.5.3 Applications

To conduct a more comprehensive evaluation, we analyzed *P4Mite* to achieve load balancing in three real-world client-server applications. Two of these applications involve the server executing inferences for machine learning tasks, specifically VGG16 and KNN. To implement VGG16, we utilized TensorFlow and TensorFlow Lite on the CPU and SmartNIC, respectively. SmartNIC's resource limitations prevented us from running the TensorFlow framework on it. Additionally, we employed the *scikit-learn* framework for K-nearest neighbours. As for the third application, the server performed DNS resolutions using the DNSlib library for Python. For simplification, all three applications operate on the UDP protocol, although *P4Mite* has the potential to handle the TCP protocol.

We deployed each application independently using *P4Mite*. Additionally, we deployed each application using Weighted Round Robin (WRR) and Equal Cost Multi-Path (ECMP) to compare our approach with existing methods. By examining the Roofline results, we observed that the server CPU has a capacity of $5x$ more than the SmartNIC. As a result, we configured WRR to direct one-sixth of the requests to the SmartNIC and five-sixths to the server. Considering the computational demands of each application, we adjusted the request rate to reach the maximum system capacity. The combined maximum rates that the host and SmartNIC can handle are 2750, 70, and 24 RPS for DNS, VGG16, and KNN, respectively. For DNS servers, which are not CPU-intensive, we utilized only one core from each resource, while for the other applications, all cores were utilized. Furthermore, we set the thresholds for DNS, VGG16, and KNN at 100ms, 150ms, and 300ms, respectively. These thresholds were chosen to be 20-30% higher than the typical application delays. To evaluate the performance, we conducted tests for a duration of 30 seconds and measured both request loss and latency for each request. Figure 4.4 illustrates the 99th percentile latency for each application. Similar to the approach used in the microbenchmarks, we terminated the execution immediately upon detecting any packet loss.



(a) DNS (b) VGG16

(c) KNN

Figure 4.4: The 99th percentile latency for specific applications

DNS Evaluation

Figure 4.4a presents the results for the DNS application. Given that DNS is not CPU intensive, both server and SmartNIC have the same delay (around 3ms) when their CPU is not under stress. The blue curve (no balancing) shows that the server can handle requests up to 80% load. At higher loads, the server delay increases drastically. Using *P4Mite*, the agent sends reports to the switch when the server starts to get overloaded. Consequently, a portion of the load is forwarded to the SmartNIC, increasing the maximum rate by approximately 20%. *P4Mite* not only avoids overloading the server but also the offload of requests to the SmartNIC allows more injunctions to be processed.

As such, one core of the SmartNIC's wimpy processor can handle 20% more DNS queries. Regarding ECMP, since it tries to split the requests between the server and SmartNIC evenly, and considering that the SmartNIC's computation power is one-sixth of the server, the SmartNIC becomes quickly overloaded. Conversely, in the weighted round-robin (RR), the switch dispatched 5/6 of the requests to the server and the remaining to the SmartNIC, enabling it to run faster than both the baseline and *P4Mite* for lower rates. However, as we reach 90%, WRR drops packets. *P4Mite* beats WRR because it wisely balances the load, while WRR proactively distributes the load. Nevertheless, WRR has better performance for less load because *P4Mite* needs to wait for the server to get overloaded before switching is triggered.

VGG16 Evaluation

Figure 4.4b delivers VGG16's results. Without stress, the hosts and the SmartNIC's delays are 80ms and 120ms, respectively. Again, in the no balancing (baseline) scenario, the host is capable of serving 80% of the load, and the latency rises exponentially for higher loads. In *P4Mite*, the latency increases if the load becomes higher than 80%; however, it is 50% less than baseline delay, and it can handle approximately 16% more load than the baseline.

Figure 4.4b also indicates that if we use ECMP, the system can handle only 30% load. We investigated these results and noticed that the SmartNIC becomes fully utilized, causing packets to be dropped. Finally, considering WRR as the load balance approach, we observe that it increases the latency compared to the baseline when the load is less than 80%. This occurs because WRR always sends 1/6 of traffic to the SmartNIC, which performs slower than the server. However, WRR enables the application to go up to 90% load because the requests sent to the SmartNIC alleviate the server.

KNN evaluation

Figure 4.4c presents the results for KNN, the most CPU-intensive application in our evaluation. In this scenario, the baseline can handle 75% load until packets start to be dropped. *P4Mite* improves the latency and maximum rate by 25% and 11%, respectively, compared to the baseline. We could not get better results for KNN because our SmartNIC is not powerful enough to process heavier requests within the time constraints. We observe that ECMP handles at most 40% load, where 20% load is forwarded to the SmartNIC. Also, the latency for ECMP is higher than other approaches because the SmartNIC executes KNN requests far slower than the server. Finally, while WRR works poorly for loads below 70%, it can handle more load than *P4Mite*. Our investigation shows that for a load of 90% and higher, requests are lost for *P4Mite* because of request time expiration in the server, not because of packet drop.

4.5.4 *P4Mite* vs. SmartNIC-based load balancer

As we mentioned before, we adopted P4-16 for *P4Mite* development. Using such implementation leads *P4Mite* to operate at the line rate. For a better assessment, we designed another experiment to compare *P4Mite* with other types of implementation. From this perspective, we deployed another load balancer conducting on our Smart-NIC. The new L4 load balancer executes on one core of the SmartNIC, as the other 15 cores should be available for request processing. The rest of the configuration for the SmartNIC-based load balancer is similar to *P4Mite*. We indicate the comparison between *P4Mite* and the SmartNIC-based load balancer in Figure 4.5. Since the requests must go through the SmartNIC, the new load balancer is slower than *P4Mite*. More importantly, the SmartNIC-based load balancer takes up to 700 RPS, and then it starts dropping packets as one core can not manage more connections. On the other side, *P4Mite* can handle up to 8k RPS without dropping packets.

4.5.5 *P4Mite* Resource Usage

At the final step of evaluation, we measured *P4Mite*'s resource usage when it was running on the Tofino Switch. Our switch supports up to 256 servers with two

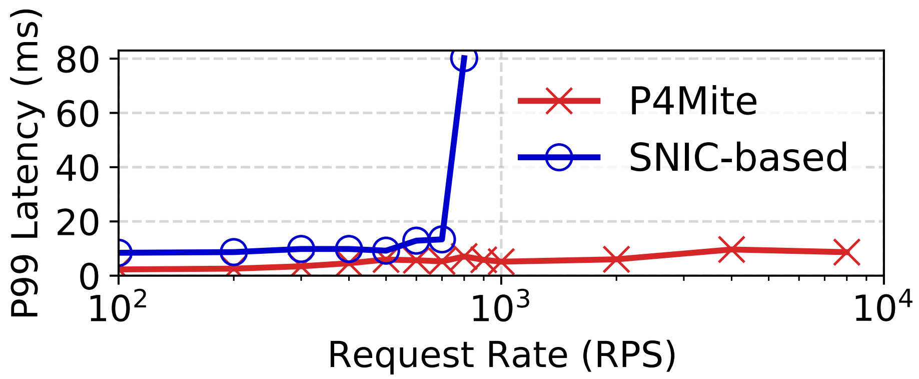


Figure 4.5: P4Mite vs. SmartNIC-based Load balance

Table 4.1: Amount of resources used by *P4Mite*.

Resource	**Usage %**
SRAM	5.1%
TCAM	0.0%
VLIW Instructions	2.6%
Hash Bit	4.0%
Stats ALU	2.1%
Map RAM	5.6%
Exact Match Input Xbar	2.9%

accelerators on each. Table 4.1 lists the resource usage of *P4Mite* if it is holding 50k concurrent connections. This number is close to the limit of *P4Mite*, which it supports by a single pipeline. *P4Mite* can serve more connections by using multiple pipelines [6].

We observe that for every switch resource, *P4Mite*'s overhead is always less than 6%. *P4Mite* uses 5.1% of SRAM and 5.6% of Map RAM mainly due to the implementation of the bloom filters. More specifically, we see that the `ConnTable` dominates the usage of these resources to store connection statuses. Because we use hash functions to map packets to both the `ServerTable` and the `ConnTable`, the Hash Bit usage is 4.0%. The VLIW is used for writing values into packets; since *P4Mite* only writes into packets their destination IPs, VLIW usage is only 2.6%. We are also using only 2.9% exact match input xbar to perform the exact match to get the accelerator state and match the DIP table. The bloom filters also use 2.1% of stats ALU to update the accelerator's state or to store state about a new connection. Finally, *P4Mite* does not use any TCAM, which is an expensive resource.

4.6 Major Conclusions of *P4Mite*

We have introduced *P4Mite*, an in-network tailored for the accelerators load balancer. Unlike previous research that either neglects resource status or performs load balancing at the per-server level, our proposed design incorporates resource monitoring agents that dynamically update the switch based on various metrics. Furthermore, *P4Mite* offers the flexibility to implement a wide range of load-balancing policies.

In our prototype, we implemented the switch on a Tofino switch using P4-16. It effectively balances the load between two resources with differing capacities: a server and an SoC SmartNIC. Our experiments demonstrate that *P4Mite* wisely forwards the requests to the more powerful resource, the server. However, if the server becomes overloaded, the switch redirects a portion of the load to the less powerful resource, the SoC SmartNIC. By effectively utilizing both resources, *P4Mite* significantly enhances the performance of applications.

By considering three applications and utilizing synthetic workloads, we have demonstrated that *P4Mite* can handle $10 - 20\%$ more load and process requests up to 50% faster compared to the baseline. A key advantage of *P4Mite* is its ability to prevent performance degradation even for low and medium loads, as it avoids relying solely on less powerful resources when more capable ones are available. Furthermore, our results indicate that *P4Mite* offers scalability without imposing significant resource requirements. This was highlighted through a comparison of *P4Mite* with a SmartNIC-based load balancer, while also measuring the resource consumption within the switch.

Chapter 5

P4Hauler

In this chapter, we present *P4Hauler*, an accelerator-aware *policy-agnostic generalized* load balancer deployed on a programmable switch (e.g., Intel Tofino), where the network administrator can update policies on-the-fly without rebooting the programmable switch to adapt its operation to different network and application conditions. Furthermore, similar to *P4Mite*, *P4Hauler* aids in reducing the load of the servers' CPU resources in two manners: first, accelerators can handle a portion of the application load; secondly, the servers and accelerators are not involved in LB operations, thereby relieving computing resources to execute application requests.

P4Hauler, like *P4Mite*, relies on agents running on servers and accelerators, which collect statistics and send the monitored resource statuses of these devices to the *P4Hauler* switch. However, in contrast to the *P4Mite*, which stores bitmaps, *P4Hauler* keeps a measurement corresponding to the actual level of each resource usage and uses these measurements to choose a destination for incoming requests.

Note *P4Hauler* can mimic *P4Mite*, where agents only reported whether a target computing resource is available, i.e., *P4Mite* is an instance of *P4Hauler*. But keeping meaningful resource values in the switch for more complex policies, which collect multiple resources or perform the computation in the switch, requires storing all possible outcomes and would exceed the switch memory capacity. Therefore, we introduce a new generalized switch that can implement and compute different policies on-the-fly, while efficiently utilizing constrained resources.

5.1 Challenges in *P4Hauler*'s Design

To summarize, *P4Mite*'s challenges are also present in *P4Hauler* due to its fine-grained load distribution functionality. In addition to the issues discussed in Section 4.1, there are further challenges in the design of *P4Hauler*, which are explained below:

5.1.1 Awareness of Resources

Each distributed application utilizes specific system resources such as CPU, memory, or bandwidth to achieve the desired performance. Therefore, accurate information about the status of these resources is crucial for load balancing in various policies. To address this, *P4Hauler* adopts a provider-collector paradigm where servers provide their resource utilization status to the switch (collector), enabling efficient decision-making for load balancing.

5.1.2 Hierarchical Design for Complex Policies

To support a wide range of policies, such as selecting the least utilized computing resource from available server/accelerator pools, *P4Hauler* requires various operations. However, switches have limited pipeline stages with a restricted number of Arithmetic Logic Unit ALU operations per stage. Additionally, the switch programming model lacks support for loops, necessitating the unrolling of computations and recirculations/resubmissions for iterations, which affects throughput. To overcome these challenges, *P4Hauler* employs a hierarchical design that carefully unrolls computations in two phases to fit within the available stages and computational capabilities. Like *P4Mite*, *P4Hauler* first selects a server using a lightweight policy and then chooses among the CPU or accelerators from the selected server.

5.1.3 Memory Management

Ensuring per-connection consistency in *P4Hauler* requires storing connection statuses, which becomes more demanding due to the granularity of computing resources. However, programmable switches have limited memory capacity. To optimize memory usage, similar to *P4Mite*, *P4Hauler* utilizes hashing and bitmap techniques to track connection statuses. Additionally, *P4Hauler* implements a *quantization* scheme to reduce memory requirements by storing statistics in a compact format within the switch.

5.2 *P4Hauler*'s Overview

Figure 5.1 presents an overview of the architecture of *P4Hauler*. It comprises three main components: (1) an *SDN controller* that enables network operators to specify application-specific policies using a high-level API, (2) agents hosted on servers and accelerators that monitor the resources of these devices and periodically transmit this information to programmable switches (If accelerators cannot independently run the agent, such as GPUs, the server's agent monitors the accelerator.), and (3) a *programmable switch* that incorporates the load balancer responsible for distributing the workload among computing resources and maintaining connection statuses.

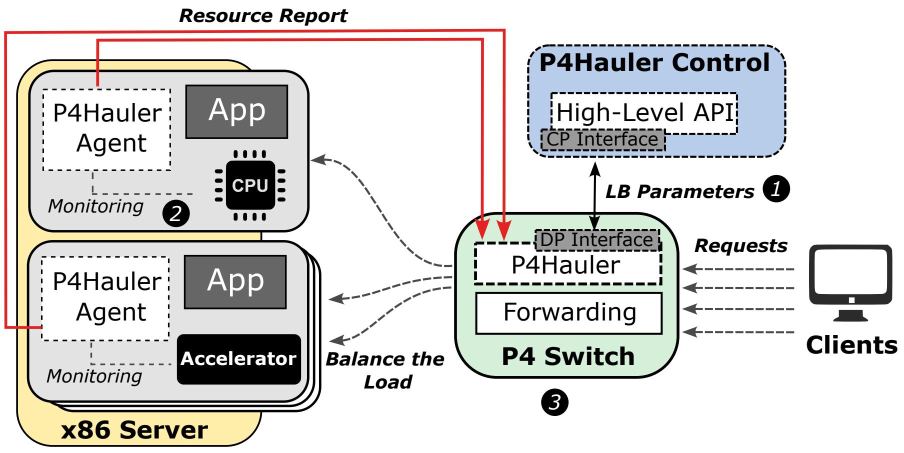


Figure 5.1: *P4Hauler* overview.

In *P4Hauler*, network administrators can define the desired policy through a high-level API, which the controller utilizes to configure the switch accordingly. Additionally, agents collect resource utilization metrics associated with the selected policy and share them with the *P4Hauler*'s switch. These agents employ a customized protocol header to transmit compressed resource statuses, minimizing the memory requirements at the switch. By leveraging the collected information and the configured policy, the switch efficiently distributes the workload among computing resources within a chosen server. Moreover, the switch manages connection consistency using compact data structures. We provide detailed explanations of each component's design in the subsequent subsections.

5.3 *P4Hauler*'s Agents

It is crucial to carefully monitor accelerators' resource utilizations due to their varied architectures and capabilities. In this section, we present schemes for monitoring and reporting resource utilization from agents, along with design optimizations to meet the switch's resource constraints, such as limited memory.

Each computing device capable of running applications in *P4Hauler* hosts an agent responsible for monitoring metrics such as CPU load, CPU utilization, memory usage, and disk usage. For instance, memory-intensive applications like key-value store databases and CPU-intensive tasks like neural network inferences require different resource allocations. Therefore, the agents monitor various resource usage statuses and periodically transmit this information to the switch for informed load balancing. If an accelerator cannot independently run the agent (e.g., GPUs), the host's agent monitors the accelerator's status.

One approach for reporting resource usage to the switch is using a predefined threshold to indicate whether a resource is available or not [12]. However, this method lacks granularity and fails to effectively utilize the target resource. Conversely, providing detailed usage information may overwhelm the switch's limited memory and communication channel. To address this, agents compress the usage measurements using a quantization technique and encapsulate the information in a custom packet header, which is appended after the Ethernet header (as shown in Figure 5.2). We apply quantitative by removing the decimal part. This estimation will not affect the decision making based on the quantized values since a minor change in the load changes the utilization more than 1% in most cases. The status, on the other hand, is reported to the switch periodically, and the reporting rate can be adjusted based on the capabilities of each device. The custom packet header includes the following fields: `DID` (unique identifier of an accelerator, such as GPU, SmartNIC, or server CPU), `CMP` (value corresponding to computing resources, e.g., CPU or GPU load), `MEM` (value corresponding to memory resources), and `NET` (value corresponding to networking resources, such as queue size). It is feasible to extend the header fields to consider different resources as required by load-balancing policies.

Optimizing resource usage at the switch. Ideally, switches should collect exact utilization measurements as 32-bit integers or 64-bit floating-point numbers

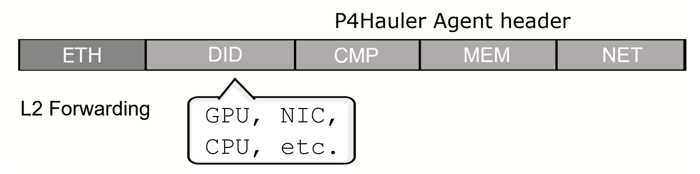


Figure 5.2: *P4Hauler* packet format.

measured by the agents and store them for further processing. However, it is impractical due to limited memory in switches and the incapability of floating-point operations in programmable switches.

In the design of *P4Hauler*, we address this issue by converting floating-point numbers to integers that the switch can process. Additionally, we store resource statuses using 8-bit integers instead of the 32-bit integers. Specifically, we can present a resource status as a number ranging from 0 to 100, and by using 8 bits, it fits the size of the registers in the switch. This process of reducing the number of bits to represent an integer is known as *quantization* [48], which reduces memory requirements at the expense of accuracy. Quantization is widely used for data compression [48, 102], making it suitable for the design of *P4Hauler*.

P4Hauler can perform quantization in two ways: either the agent collects statistics and quantizes them, or the switch receives the exact measurements and performs quantization. While both approaches are practical, the former is preferred as it is easier to implement quantization on agents than the switch. Additionally, quantizing at the agent level reduces the amount of data transmitted over the network and alleviates computing burdens on the switch.

5.4 *P4Hauler*'s Infrastructure

This section first discusses how the resource usage statuses from agents are stored and updated in the switch data plane. Then, we explain how the switch uses that status and different policy building blocks for load-balancing decisions. Finally, we discuss how to support per-connection consistency.

5.4.1 Handling Resources at the Switch

The switch is at the heart of the *P4Hauler* system that can be dynamically configured to support various policies. Briefly, the network administrator assigns on-the-fly configurations to the switch's functionality, and the switch splits the workload according to this configuration. On the other hand, the switch saves the measurements, computes the policy according to the resource updates, and simultaneously routes the data connection packets. However, we face the following challenges in implementing the policy computation.

Multiple arrays for servers and accelerators. The switch data plane executes only one action on an array of registers per packet. Consequently, manipulating the statuses of servers and accelerators in a single array upon receiving an update is impossible. However, some policies, such as selecting the least utilized resource, necessitate accessing and comparing multiple statuses. To address this, we store the device statuses across D different register arrays. The switch dedicates the first array to store the servers' statuses while allocating the subsequent arrays for the accelerators'. This division enables operating on each array within a single pipeline, facilitating decision-making based on the desired server and its associated accelerators. More specifically, the server's state is read from the first array, and the status of the first accelerator is retrieved from the following array, continuing this process for subsequent accelerators.

However, if the value of D becomes large, it introduces a new challenge. The switch programming model has limited stages (e.g., 12 in a Tofino 1 switch) and the available ALU operations per stage. In some scenarios, load balancing may need to be performed across numerous servers, potentially exceeding $1,000$, with each server equipped with approximately 7 accelerators [103]. Consequently, reading the resource status of each device would surpass the available ALU operations. To overcome this constraint, we propose a hierarchical solution capable of handling a few thousand servers (e.g., $1k - 10k$), thereby minimizing the number of resource accesses within the switch.

Our load balancer employs a computationally efficient policy to select a server within the data center. Subsequently, another policy, which may involve more intricate operations, determines the end destination among the CPU and available accelerators on the chosen server, based on their respective resource availability. Considering the presence of 7 accelerators per server, the second policy requires accessing 8 arrays in total ($D = 8$), fitting within a single pipeline. However, implementing multiple resource-aware policies, such as those considering CPU and memory, may necessitate more ALU operations than can be accommodated within a single pipeline. In such cases, packet resubmission would work, although it introduces additional processing latency [104]. We intend to explore these design challenges further and investigate potential mitigation strategies in our future work.

5.4.2 In-Network Load Balancing Policy Support

Implementing new policies by modifying the source code of monolithic P4 programs can be challenging. In contrast, *P4Hauler* proposes a flexible switch architecture comprising various building blocks that can be combined and configured to support multiple policies. The key idea behind our approach is to identify common design patterns and requirements among different policies, allowing us to reuse the building blocks across multiple scenarios. For instance, policies such as Least Utilized Resource (LUR), Power-of-Two (Po2), and Join-the-Shortest-Queue (JSQ) follow a similar procedure: (1) reading a set of monitored values (e.g., resource utilization, number of connections, or queue size) and (2) selecting a device based on a computation performed using the read values. On the other hand, policies like Weighted Round Robin (WRR) and Equal-Cost Multipath (ECMP) distribute the load based on a specific distribution strategy.

Based on the above observations, we have identified two fundamental building blocks, namely the *minimum finder* and the *round-robin scheduler*, which can be employed to construct a wide range of policies. Some examples are provided in Table 5.1. The *minimum finder* manages a set of monitored variables obtained from the devices and determines their minimum value. On the other hand, the *round-robin scheduler* selects a destination based on a specified distribution function. In the subsequent sections, we provide a detailed description of these building blocks.

- ***Minimum finder:*** The *minimum finder* block employs a series of *if-else*

Table 5.1: Example policies and computation requirements

Example Policy	Requirement
Least utilized resource (LUR)	Resource Info and Compute Min
Least connections and weighted least connections	Connections Info and Compute min
Join the shortest queue (JSQ)	Queue Info and Compute Min
Weighted Round Robin (WRR) and Dynamic WRR	Resource Info and Round Robin Scheduler
ECMP	Scheduler

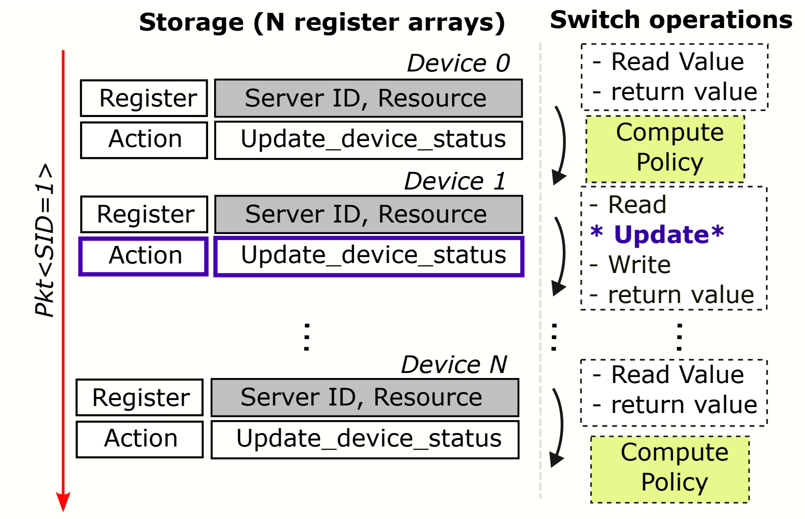


Figure 5.3: Unrolling device state for policy computation.

statements unrolled across the pipeline to perform comparisons. It reads the state of each device and selects the one with the lowest utilization of a specific resource. The number of variables within the *minimum finder* block corresponds to the total number of devices (server + #accelerators). Figure 5.3 illustrates an example of this block. If required by the policy, multiple *minimum finder* blocks can be utilized, each operating on a distinct performance metric.

- ***Round-robin scheduler:*** The *round-robin scheduler* employs a ring buffer, as depicted in Figure 5.4, to evenly distribute the workload. Each slot in the buffer represents a destination device, and a pointer indicates the device to which the request should be directed. The slots in the buffer can be configured statically or dynamically to implement various policies. It is important to note that the number of times a device appears in the ring buffer determines its weight in the load-balancing process. For example, in Figure 5.4, the GPU's weight is 4, while the CPU and NIC each have a weight of 2. Consequently, an 8-sized ring buffer is needed. In our implementation, we utilize a large array (e.g., 1024 or 2048) and allocate a portion of it based on the weight requirements.

It should be noted that the described components may not cover every conceivable policies. Nonetheless, the *P4Hauler* design is flexible and allows for the inclusion of additional blocks. For instance, to accommodate Po2 choices, a new component is required to randomly select two resources and opt for the less utilized one.

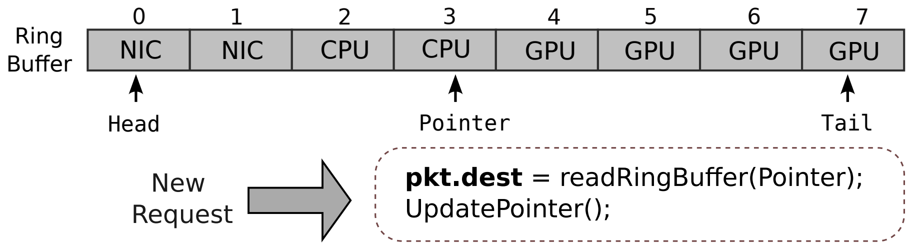


Figure 5.4: An example of the round robin scheduler.

The policy in the switch's data plane may involve one or multiple building blocks, which can be utilized based on the requirements of the application. The controller is responsible for initializing and dynamically adjusting the parameters associated with these blocks without rebooting the switch. Details regarding the configuration of different blocks to support a range of policies are discussed in Section 5.5.

5.4.3 *P4Hauler*'s Data Plane Layout

This section provides the data plane structure of *P4Hauler*, illustrated in Figure 5.5. The data plane implementation of *P4Hauler* involves the utilization of one or more building blocks to implement policies while maintaining connection status. Upon receiving a packet, an ingress element in the data plane checks whether it intends to update a resource usage in tables or establish a new connection.

For update packets, the switch modifies the identified accelerator's resource value in either the *minimum finder* block or the *round-robin scheduler*. The active policy is then computed based on the application requirements, considering the active

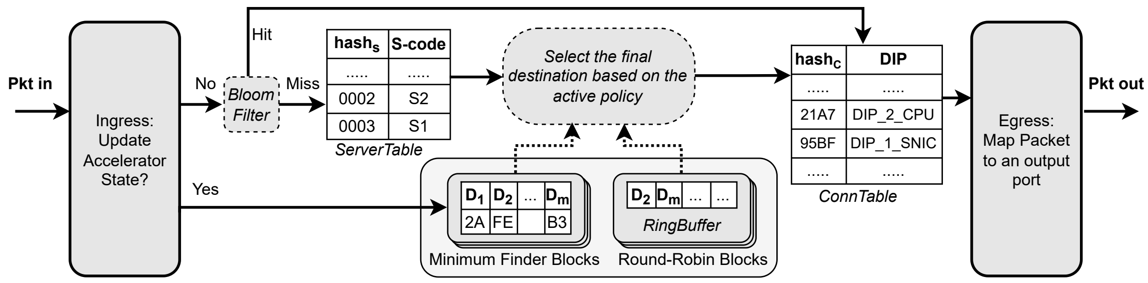


Figure 5.5: *P4Hauler*'s data plane layout.

policy configured in the switch. The computation can involve a *minimum finder* or a *round-robin scheduler*, with the calculation unrolled across the pipeline. Once the switch performs the computation for the policy, it stores the resulting destination in persistent memory for future access by new requests.

For connection packets, however, the previously computed policy update value is read from persistent memory. The switch loads the resulting destination as metadata and forwards it to the connection table (`ConnTable`). The switch matches the packet using a computed hash and reserves a cell in the connection table for the connection destination. New connections add a new entry to the connection table while existing connections are directed to the previously chosen destination, disregarding the policy metadata. Finally, the switch removes the cell entry when the last packet of a connection is processed.

5.5 *P4Hauler* Management

Network operators can leverage *P4Hauler* to define load-balancing policies by utilizing a few data plane building blocks that align with their desired policies. Furthermore, operators are required to specify the target applications when the system caters to multiple applications. To facilitate this process, *P4Hauler* offers a high-level control plane API that can generate configuration files tailored to the target switch. It is important to note that the format of the generated configuration files may vary depending on the architecture of the programmable switch. Furthermore, we assume the presence of a reliable controller and a stable connection between the controller and the dataplane. This implies that in the event of a service interruption, the administrator has the capability to modify the configuration in the dataplane.

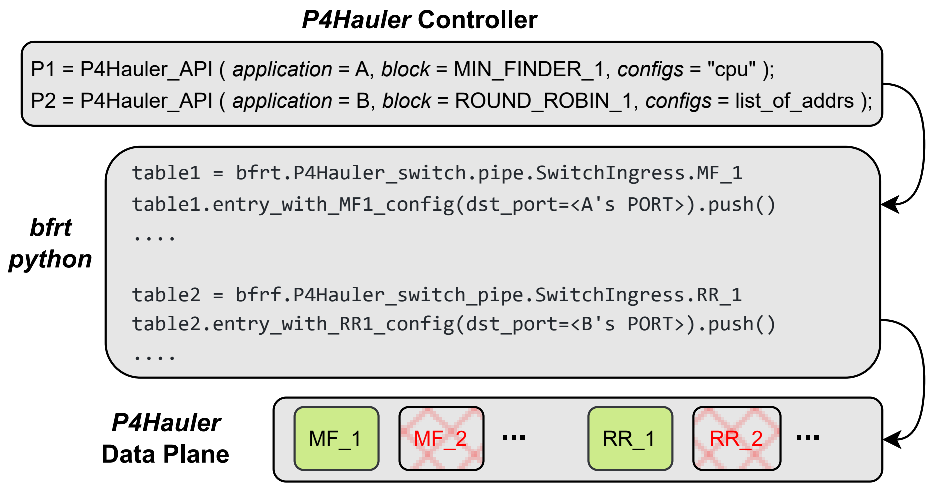
```
def P4Hauler_API(application, block, configs):
```

P4Hauler's controller analyzes the input parameters given to the API and generates the requisite directives or functions for configuring the switch. Among these parameters, the `application` and `block` options dictate the chosen application and the building blocks (either the *minimum finder* or the *round-robin scheduler*) that the switch will utilize to enforce the policy. To illustrate, in order to activate the `least utilized CPU` policy, the operator would supply a *minimum finder* and specify the monitoring of CPU utilization. Subsequently, the controller processes these inputs and configures the switch with the appropriate functions. In cases where a *round-robin scheduler* is designated, the `configs` parameter specifies the weights for the scheduler.

Figure 5.6 provides two examples of policy distribution. The first example involves application A, which utilizes the *minimum finder* block (identified as MF_1 in Figure 5.6). The policy mandates monitoring and recording CPU utilization. Similarly, the second example pertains to application B, which requires a *round-robin scheduler*. The `configs` argument includes a list of weights that the *P4Hauler*'s switch stores in the *RingBuffer* (designated as RR_1 in Figure 5.6). *P4Hauler* employs interfaces like *Python3* and the Barefoot runtime (*bfrt*) for Tofino switch to deploy various components in the data plane. There is no necessity to create a new building block for monitoring additional resources like memory or bandwidth. Instead, we can utilize and enable multiple of the required building blocks, configuring each one to monitor a specific resource for individual applications, as needed. Thus, in cases where we need policies involve multiple resources (e.g., CPU and memory), we can use multiple of the aforementioned building blocks while selecting a computing resource.

The operator can reconfigure the arrangement of building blocks and adjust their configurations while the switch is operational. As a result, policies can be modified on-the-fly in *P4Hauler* without the need to reboot the switch. In the following, we demonstrate how the operator can configure the parameters of the building blocks in the switch to implement the policies outlined in Table 5.1. For illustrative purposes, we consider a scenario with a single server equipped with two SmartNICs.

Activating ECMP: To implement the ECMP policy, we can utilize a *round-robin scheduler* block and store equal weights in all cells in the *RingBuffer*. In this



```
P1 = P4Hauler_API ( application = A, block = MIN_FINDER_1, configs = "cpu" );
P2 = P4Hauler_API ( application = B, block = ROUND_ROBIN_1, configs = list_of_addrs );
```

```
table1 = bfrt.P4Hauler_switch.pipe.SwitchIngress.MF_1
table1.entry_with_MF1_config(dst_port=<A's PORT>).push()
....

table2 = bfrf.P4Hauler_switch_pipe.SwitchIngress.RR_1
table2.entry_with_RR1_config(dst_port=<B's PORT>).push()
....
```

Figure 5.6: *P4Hauler*'s API examples.

case, all three computing resources, namely the server, SmartNIC1, and SmartNIC2, will have equal weights, with $W_{server} = W_{SmartNIC1} = W_{SmartNIC2} = 1$.

Activating WRR: Similar to ECMP, we can employ a *round-robin scheduler* and assign weights based on the computational capacity of each computing resource. In our initial investigation, we found that the server is 5x more powerful than the SmartNIC. Therefore, we can assign a weight of $W_{server} = 5$ and $W_{SmartNIC1} = W_{SmartNIC2} = 1$.

Activating Dynamic WRR: Initially, we can begin with the weights we explained in Weighted Round Robin (WRR) policy. Subsequently, the switch can update the weights of the monitored resources based on an `update` function. For instance, we can set the weight to be inversely proportional to the CPU utilization by using the following update function.

$$Weight_{d,t} = Weight_{d,0} \times (1 - CPU_{d,t}) \tag{5.1}$$

Here $Weight_{d,t}$ is the weight assigned to the link connecting the switch to resource d at time t and $CPU_{d,t}$ is the CPU utilization of resource d at the same time.

Activating Least Utilized Resource (LUR): The Least Utilized Resource policy leverages the *minimum finder* block to determine a destination with the highest availability of the selected resource (e.g., CPU, memory, or network usage). When an update packet is received and processed in the switch pipeline, the designated building block calculates the minimum value among the currently utilized resources and retains it for new connections. For instance, if the server's CPU is found to have the least utilization, the load balancer forwards the new connections to it instead of more utilized SmartNICs.

Activating Join the Shortest Queue (JSQ): The join-shortest queue policy is a variant of the LUR policy, where we can use a *minimum finder* to keep the queue sizes. Then, the policy picks the destination with the shortest queue.

Activating Prioritization (PRT): This policy prioritizes the fastest destination in terms of processing the requests, and then proceeds to others in descending order of their capabilities. With this mind, the fastest resource responds to the request if it is available. For instance, PRT picks the CPU due to its superior computing power compared to the SmartNICs. When the CPU is fully utilized, PRT forward the requests to the SmartNIC, however reverts to the CPU as soon as it can handle additional workloads.

5.6 *P4Hauler* Implementation

In this section, we will delve into the implementation details of *P4Hauler*. All versions of *P4Hauler* are accessible at [15]. Our development efforts include creating a prototype using the Tofino switch and building a simulator to assess its correctness at data center scales.

5.6.1 Prototype Implementation

The switch and controller implementations of *P4Hauler* are based on P4-16 and Python3, respectively. We have mapped our P4 application to the Tofino Native Architecture (TAN). The *ConnTable*, *ServerTable*, *minimum finder*, and *round-robin scheduler* blocks utilize the available registers on the switch. The controller employs Tofino's API to configure the blocks in the data plane. Within the switch, the active policy calculates the destination and stores it in a register, which the switch retrieves when it receives a new connection. Additionally, the controller pushes the necessary match-action rules to the data plane routing table.

The agent implementation of *P4Hauler* consists of approximately 250 lines of Python3 code. It makes use of the *psutil* library to measure resource usage and the *scapy* library to create the customized header displayed in Figure 5.2. The agents measure and quantize the resource usage that the switch stores in registers.

Testbed setup. The testbed consists of a Wedge 100BF-32X programmable switch with a 3.2Tbps Tofino ASIC [14], connecting two x86 machines. One of the machines acts as the client, sending requests to the other machine, which functions as the server. The server mounts an Intel(R) Xeon(R) Silver 4210R CPU running at 2.4GHz, with 10 cores and 32GB of memory. Additionally, the server is equipped with an accelerator, a dual-port SFP28, PCIe Gen3.0/4.0 x8 BlueField(R) G-Series, which features 16 cores, 16GB of on-board DDR4 RAM, and enabled crypto accelerators. We used the same configuration for all experiments unless specified otherwise.

Policies and configuration. We have selected Weighted Round Robin (WRR) with weights set to 5:1 and Least Utilized (LUR) with the least utilized CPU configuration from Table 5.1. WRR and LUR employ the *round-robin scheduler* and *minimum finder* as building blocks for their policies, and the assigned weight are based on each resource capacity shown in Section 2.4. Additionally, we have considered Prioritization (PRT), the policy that we showed its potential for boosting the performance in *P4Mite* (Section 4). In this proposed policy, when the CPU reaches full utilization, available accelerators (e.g., a SmartNIC) are prioritized for load forwarding.

In all three policies, agents from servers and accelerators share measurement updates at a rate of 100 updates per second (the rationale behind this rate selection is further explained in Section 5.7.3). Lastly, we have implemented the baseline policy where the server solely processes all requests without utilizing any accelerators. In scenarios involving multiple servers, the baseline can utilize an ECMP policy, which does not incorporate any accelerators.

Table 5.2: List of Applications for *P4Hauler* assessment.

Application	**Method / Algorithm**	**Dataset**	**Framwork**
Anomaly Detection	MLP NN [48].	UNSW-NB15 [105]	TensorFlow (TensorFlow lite)
Natural Language Processing	BM25 [106]	SciFact [107]	Python3's Library
Image Classification (Supervised)	VGG16 [108]	PetImages [109]	TensorFlow (TensorFlow lite)
Image Classification (Unsupervised)	KNN [110]	MNIST [111]	Scikit-Learn

Applications. For the evaluations of *P4Hauler*, we have utilized four ML/DL applications, as listed in Table 5.2. The first application is an Anomaly Detection

(AD) system [48], which employs a multilayer perceptron neural network (MLP-NN) to identify abnormal activities in networks. We have implemented the AD system using network traffic from the UNSW-NB15 dataset [105] with the TensorFlow and TensorFlow Lite frameworks. TensorFlow Lite is a version of TensorFlow designed for resource-constrained devices such as the SmartNIC.

The second application is a natural language processing (NLP) system called BM25 [106]. It consists of a collection of text-retrieval algorithms used for ranking functions in search engines. We have employed the existing BM25 library in Python3 and applied the model on the SciFact dataset [107].

The third and fourth applications focus on image classification (IC) using supervised VGG16 [108] and unsupervised KNN [110] algorithms, respectively. VGG16 utilizes the PetImages dataset [109] with TensorFlow on the server and TensorFlow Lite on the accelerator. On the other hand, KNN operates on the MNIST dataset [111] using the Scikit-learn framework, both on the server and the accelerator.

5.6.2 Simulation

The simulation study aims to show the performance of *P4Hauler* on a large scale. However, there is currently no available simulator that provides the necessary configuration for a set of servers with accelerators. As a result, we have developed our own simulator based on the specifications outlined in our testbed data. *Each network element is selected and configured in alignment with the testbed configuration.* For a comprehensive assessment of the simulator's outcomes, please refer to Appendix A, where we present evaluation results to verify the accuracy of the simulator. In particular, the network switch employs Equal-Cost Multipath (ECMP) for server selection and then applies one of the policies extensively examined during our testbed evaluation to distribute the workload between the CPU and accelerators. Our simulation methodology heavily relies on the insights derived from the testbed's findings, particularly in terms of the performance capabilities of servers and accelerators, which are measured in requests per second (RPS). By meticulously accounting for the capacities of servers equipped with accelerators, we can evaluate the degree of performance improvement achievable through the implementation of *P4Hauler*'s policies on a data center scale.

We assess three scenarios, each comprising 64, 128, and 256 servers, all equipped with two SmartNICs, while using KNN as the application. This choice is based on KNN's ability to generate the most computationally demanding workload, representing the worst-case load scenario. Nevertheless, our simulator can accommodate various load requirements as needed.

5.7 *P4Hauler* Evaluation

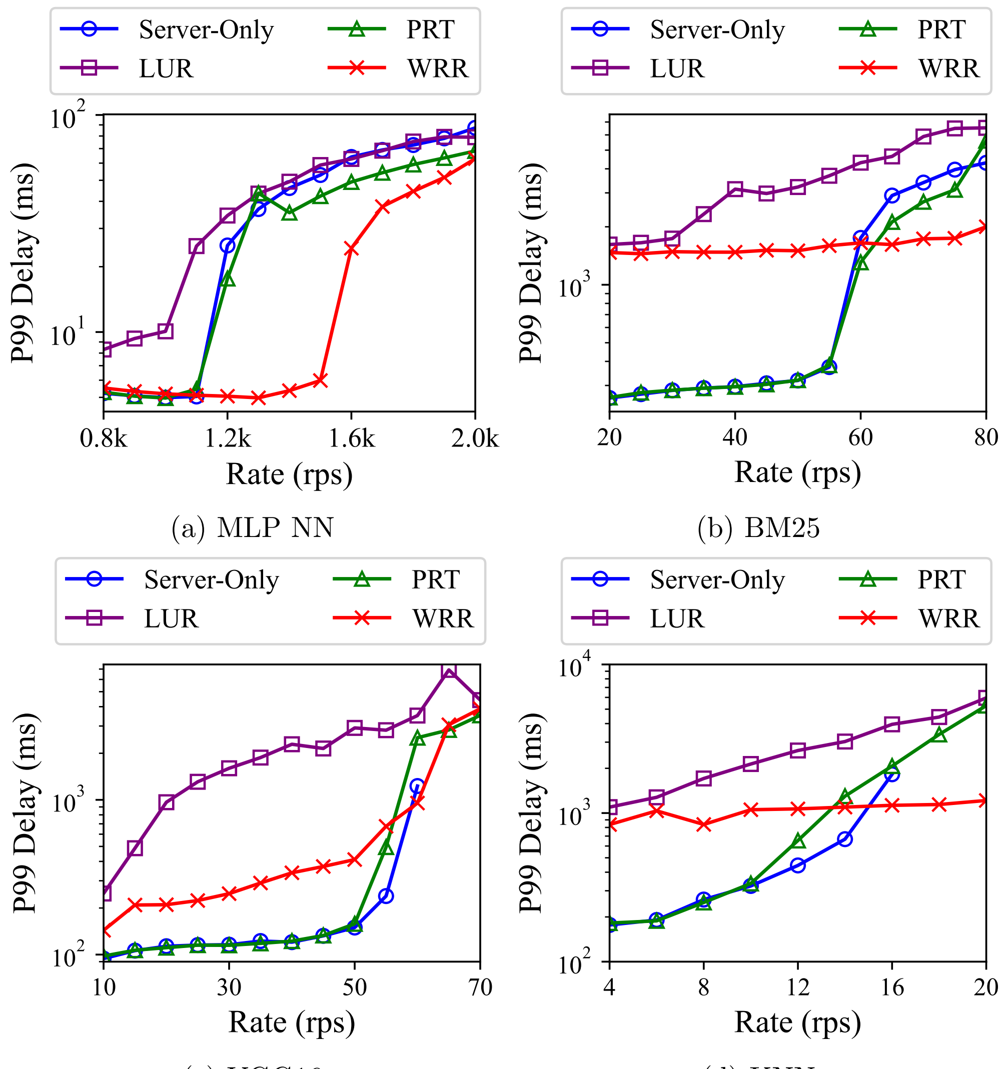
We conduct the evaluations of *P4Hauler* using the explained testbed and simulator. The prototype implementation and assessment on a real testbed aim to showcase its performance and feasibility for deployment. On the other hand, the simulation results provide insights into the performance improvements achieved by *P4Hauler* at scale.

5.7.1 End-To-End Delay

In this section, we conduct a comparison of the 99th percentile (P99) end-to-end (E2E) delay of applications using various policies supported by *P4Hauler* on our testbed. For this assessment, the client transmit requests following a Poisson distribution at a defined rate, and either the target server or the SmartNIC responds to these requests. We increase the request rate for each application until packet drops are observed at the end host, which can be either the server or SmartNIC. We also conduct each experiment 20 times, each time for a duration of 30 seconds at each request rate. Note that in our testbed, packets start dropping at the end host after 30 seconds due to the request overloading the CPU and filling up the NIC's buffer. We evaluate and compare the performance of four chosen policies.

Anomaly Detection (AD)

AD, being the lightest application in terms of CPU demand, was restricted to utilizing only one core from either the server or the SmartNIC to showcase the impact on load balancing. This measure ensured that even the lightest application would overload the computational resources during the experiment. Figure 5.7a illustrates the 99th end-to-end (E2E) delay for AD. It observes that the baseline can handle up to $1.1k$ requests per second (RPS) without experiencing performance degradation.



(a) MLP NN

(b) BM25

(c) VGG16

(d) KNN

Figure 5.7: The 99th percentile E2E delay for different applications.

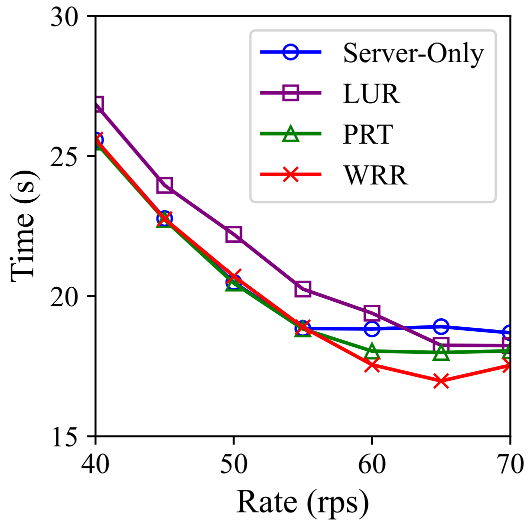
However, beyond that threshold, the delay increases. Regarding LUR, the switch directs requests to one computing target until the other target experiences reduced utilization. Due to SmartNIC's weaker processor compared to the server, the overall end-to-end (E2E) performance is affected. Nevertheless, the workload distribution between the server and accelerator compensates for the processing delay caused by the NIC. Overall, LUR performs similarly or slightly better than the baseline for higher rates. PRT assumes that the server possesses a powerful processor and prioritizes this resource over the NIC, utilizing it only when the server's CPU is fully utilized. Consequently, PRT outperforms the other two policies, particularly at high loads. Lastly, WRR surpasses the other policies by considering the capacity of both targets while distributing loads. For example, the server can process five times more than the NIC. However, as anticipated, all policies suffer significant delays at high loads.

Natural Language Processing (NLP)

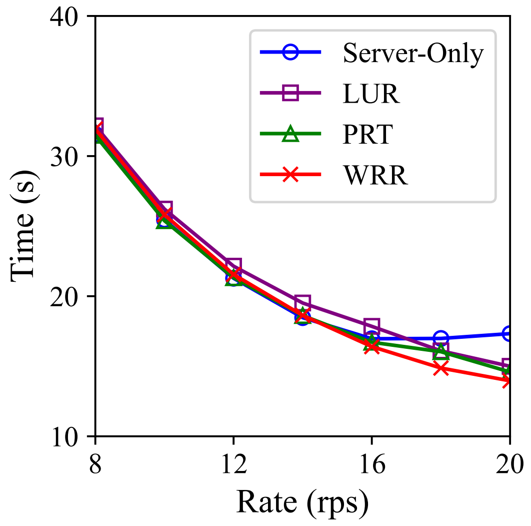
Figure 5.7b displays the end-to-end (E2E) delay for processing the NLP model BM25. The baseline's delay exhibits a linear increase of up to 55 requests per second (RPS), beyond which the increase becomes exponential, indicating that the server becomes overwhelmed. Compared to AD, the BM25 model requires more processing; thus both LUR and WRR experience slightly higher delays at rates higher than 55 RPS. However, the load distributions compensate for this delay. More specifically, WRR outperforms LUR by dividing the load based on the capacity of each target. Lastly, PRT performs similarly to the baseline up to a rate of 60 RPS, as it gives priority to the server over the SmartNIC until the server becomes overwhelmed. Therefore, at rates exceeding 60 RPS, we observe better performance from PRT compared to the baseline.

Image classification using VGG16

Figure 5.7c depicts the end-to-end (E2E) delay for VGG16. Among the different policies, LUR exhibits the poorest performance. This can be attributed to the weaker CPU of the SmartNIC compared to the server. By sending requests to the SmartNIC even when the server CPU can still handle packets, the overall system delay increases compared to the policies that allocate accelerator resources more effectively. Nevertheless, LUR can handle requests up to a rate of 70 RPS, offering an improvement of approximately 27% over the baseline. For lower rates (up to 55 RPS), the baseline and PRT are preferable options as they do not route requests to the accelerator, thereby avoiding any decrease in performance. However, at higher rates, the baseline's delay increases $10x$ compared to the delay at 55 RPS, which is deemed unacceptable. Finally, WRR demonstrates superior performance at high rates due to its load-balancing mechanism. By actively distributing the load, more resources become available for the application, leading to improved performance.



(a) VGG16



(b) KNN

Figure 5.8: Flow completion time for ML applications.

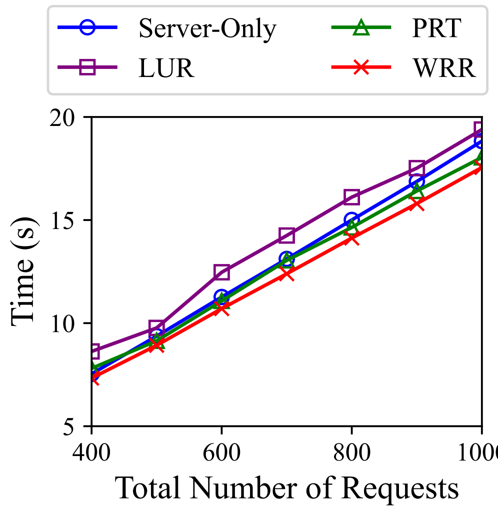
Image classification using KNN

KNN stands out as the most computationally demanding application. We observe that the baseline can handle KNN requests only up to a rate of 16 requests per second (RPS). In contrast, the other policies available in *P4Hauler* demonstrate the ability to handle up to 20 RPS, representing a 25% higher tolerance compared to the baseline. Similar to the results observed for VGG16, LUR exhibits the poorest end-to-end (E2E) delay for KNN. On the other hand, PRT performs comparably to the baseline for low rates (less than 12 RPS) and outperforms WRR. However, as the rate surpasses 16 RPS, WRR emerges as the most suitable policy.

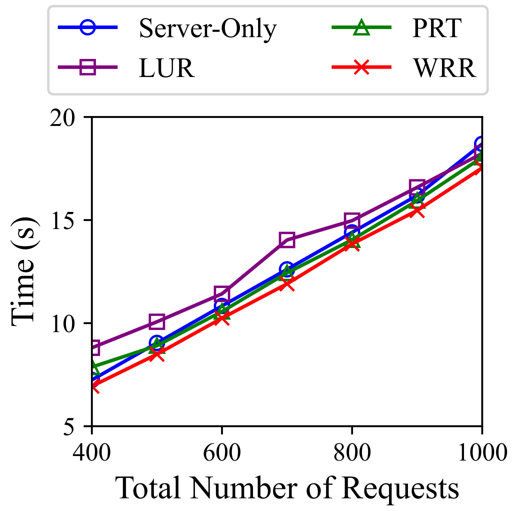
5.7.2 Flow Completion Time (FCT)

This section presents the flow completion time of the above policies with the most computing-intensive models, VGG16 and KNN, at various rates and loads.

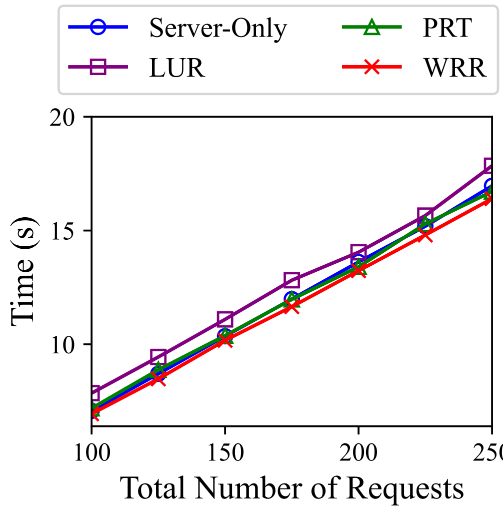
Figure 5.8 displays the flow completion time (FCT) for VGG16 and KNN with a total of 1000 and 250 requests, respectively. At low load, both WRR and PRT exhibit comparable performance to the baseline. The reason is SmartNIC's speed in the case of WRR and the exclusion of SmartNIC usage in the case of PRT. Conversely, LUR demonstrates a higher flow completion time under the same conditions. At high load, specifically at rates of 65 RPS for VGG16 and 18 RPS for KNN, LUR surpasses the



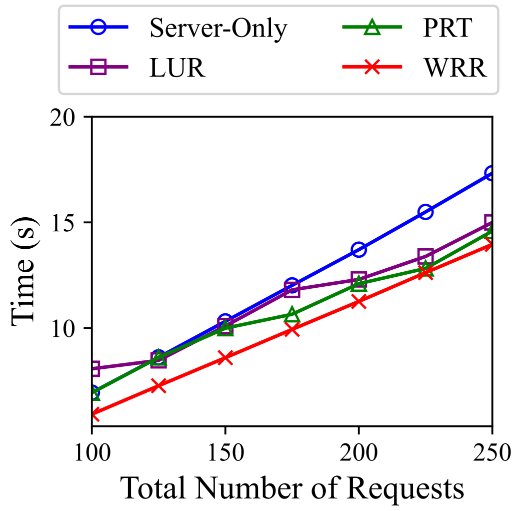
(a) VGG16, Rate: 60 RPS



(b) VGG16, Rate: 70 RPS



(c) KNN, Rate: 16 RPS



(d) KNN, Rate: 20 RPS

Figure 5.9: Flow completion time of different batch sizes.

baseline and achieves performance similar to PRT. Finally, WRR exhibits the shortest completion time across all scenarios.

We conducted additional experiments to compare the flow completion time (FCT) of different policies in scenarios before and after the server CPU becomes overloaded in the baseline solution. Taking VGG16 as an example, we observed in Figure 5.8a that for 1000 requests, the baseline outperformed LUR at a rate of 60 RPS, while LUR performed better than the baseline at 70 RPS. Based on this observation, we selected these two rates and varied the total number of requests from 400 to 1000, presenting the results in Figure 5.9a and Figure 5.9b. Figure 5.9a shows that LUR exhibited a higher flow completion time compared to the baseline. This trend continued in Figure 5.9b up to a batch size of 700. However, for batch sizes of 800 and 900, LUR performed similarly to the baseline. Finally, with a batch size of 1000, LUR became slightly faster than the baseline.

The same trend was observed for KNN, as depicted in Figure 5.9c and Figure 5.9d. We conducted experiments at rates of 16 and 20 RPS, varying the batch size from 100 to 250 requests. Both figures indicate that WRR performed the best, followed by PRT, in terms of flow completion time. Figure 5.9c shows that the baseline outperformed LUR at a rate of 16 RPS. However, Figure 5.9d illustrates that LUR only performed worse than the baseline for a batch size of 100 requests. For batch sizes of 125 and 150, LUR handled the requests with a completion time similar to the baseline. For larger batches, LUR outperformed the baseline, reducing the flow completion time from 17.32 seconds to less than 15 seconds (approximately a 13.2% decrease).

5.7.3 ***P4Hauler*'s Prototype Overheads**

Table 5.3 presents the resource utilization percentages of our prototype on the Tofino switch while handling up to 50,000 connections, with five accelerators at each server. Bloom filter's implementation consumes 5.5% of SRAM, while the connection table utilizes 7.6% of Map RAM. Notably, *P4Hauler* does not employ the costly TCAM memory resource. Additionally, we assessed the overhead introduced by the agents. We observed that the agents incur more overhead when transmitting a greater number of update packets to the switch. However, the observed overhead on the server's CPU was consistently less than 5%. To estimate the overhead on our accelerator, specifically, the SmartNIC using VGG16 and the LUR policy, we deliberately selected a computation-intensive application and the worst policy to represent the worst-case scenario for the overhead measurement.

Table 5.3: Resources usage of *P4Hauler*'s switch

Logical Table ID	**Map RAM**	**Meter ALU**	**SRAM**
9.4%	7.6%	16.7%	5.5%

Figure 5.10 displays the delay and overhead on the SmartNIC's CPU when the

client executes 50 RPS. It is evident that for a low update rate, the overhead is negligible. However, the system response is slower due to the lack of granular statuses of the computing resources, resulting in increased end-to-end (E2E) delay. As the number of updates per second (Update Rate) increases, the E2E delay decreases exponentially initially and then linearly. However, the overhead escalates for high update rates, specifically when sending more than 20 updates per second. In summary, higher update rates lead to linear improvement in the delay but a sharp increase in CPU overhead. Therefore, the operator should determine an optimal updating rate for a given application. Based on these evaluations, we selected an update rate of 100 for our assessment, introducing a maximum overhead of 10%. For higher rates, the overhead becomes excessively high (approximately 30%) without any improvement in the delay.

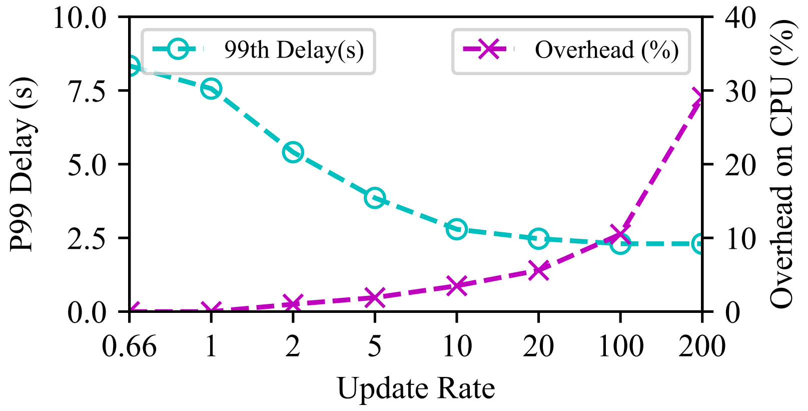


Figure 5.10: Agent interval evaluation.

5.7.4 ***P4Hauler*** **Comparison with the State-of-the-Art**

In this subsection, we assess and contrast the effectiveness of *P4Hauler* in relation to existing solutions, taking into account two separate viewpoints: (1) deploying a load balancer on various hardware platforms, and (2) comparing with the state-of-the-art in-network solutions. In this experiment, we modify the testbed's environment by employing a Tofino switch to distribute the workload across two servers, each equipped with a Bluefield SmartNIC.

Impact of target hardware

We conduct a comparative analysis of *P4Hauler* against a load balancer operating on either a SmartNIC or on a server. We implement a WRR policy, as it has demonstrated optimal performance in prior evaluations at the rate of 20 RPS. To perform load balancing, we employ `Nginx`[1][112] and allocate a dedicated CPU core for this purpose.

Table 5.4 displays the average and 99th percentile End-to-End (E2E) delays for KNN requests, our most CPU-intensive workload, at a rate of 20 requests per second (RPS). Our findings reveal a substantial performance advantage for *P4Hauler* as it diminishes the 99th percentile E2E delays by approximately a factor of three.

This improvement arises from the fact that both the NIC and the server have one fewer core available for processing incoming requests, with that core being tasked with managing concurrent connections. In contrast, in *P4Hauler*, the switch consistently processes traffic at the line rate while maintaining connection states within efficient data structures. Simultaneously, all cores on the SmartNIC and the server are available and engaged in request processing.

Table 5.4: The performance comparison of different hardware targets.

E2E Delay (ms)	**P4Hauler**	**Server LB**	**SmartNIC LB**
Average	446.35	1871.58	1868.78
99th Percentile	1215.29	3707.79	3807.06

Comparison with in-network load balancers

In this evaluation, we incorporate existing in-network switch-based Load Balancers such as Cheetah [10] and SilkRoad [6]. These systems deploy their LBs on a programmable Tofino switch to distribute the load among servers, without taking into account the status of accelerators. However, our evaluation demonstrates that such accelerator-agnostic solutions perform less effectively than *P4Hauler*, a LB that is aware of accelerators. Specifically, we use KNN application with request rate ranging from 20 to 40 for both *P4Hauler* and Cheetah.

[1]`Nginx` can efficiently distribute incoming network traffic or requests across multiple backend servers.

Figure 5.11 presents a comparison of the 99th percentile E2E latency between Cheetah and *P4Hauler*. As expected, Cheetah delivers comparable performance to *P4Hauler* at low rates, which is below 30 RPS. However, Cheetah's performance degrades significantly at higher rates. At low rates, its server CPU can handle the requests effectively, but as the load increases, the CPU becomes overwhelmed, leading to performance degradation.

In contrast, *P4Hauler*, which is aware of the usage status of accelerators and CPUs, efficiently distributes the load across these computing resources. However, we do observe a slight decrease in *P4Hauler*'s performance compared to Cheetah at low rates, which can be attributed to the use of less powerful NIC processors in our testbed. This trend significantly reverses at higher rates due to judicious utilization of the computing resources according to their capacity. Specifically, at high loads, the delay gap between *P4Hauler* and Cheetah is 2.5x, making the former a more favorable solution.

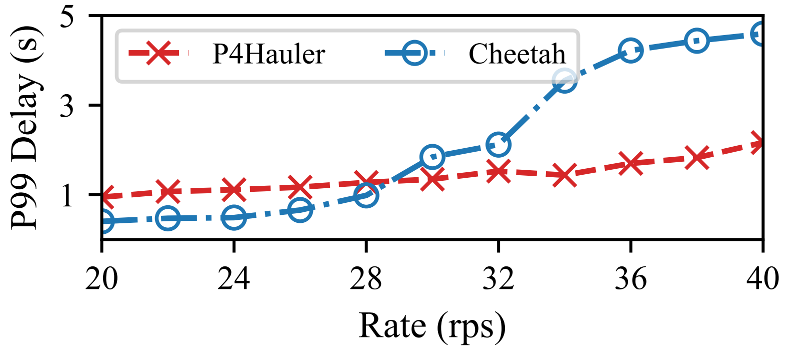


Figure 5.11: *P4Hauler* vs. Cheetah

5.7.5 Simulation Results

In this section, we examine the scalability of *P4Hauler* by evaluating its performance under high server quantities, specifically with 64, 128, and 256 servers, each equipped with two SmartNICs. The reported results are based on an average of 1000 test runs and align with the performance patterns observed in the testbed. This means that the baseline policy is the least effective while the WRR policy proves to be the most optimal, and this difference in performance becomes especially noticeable under heavy workloads.

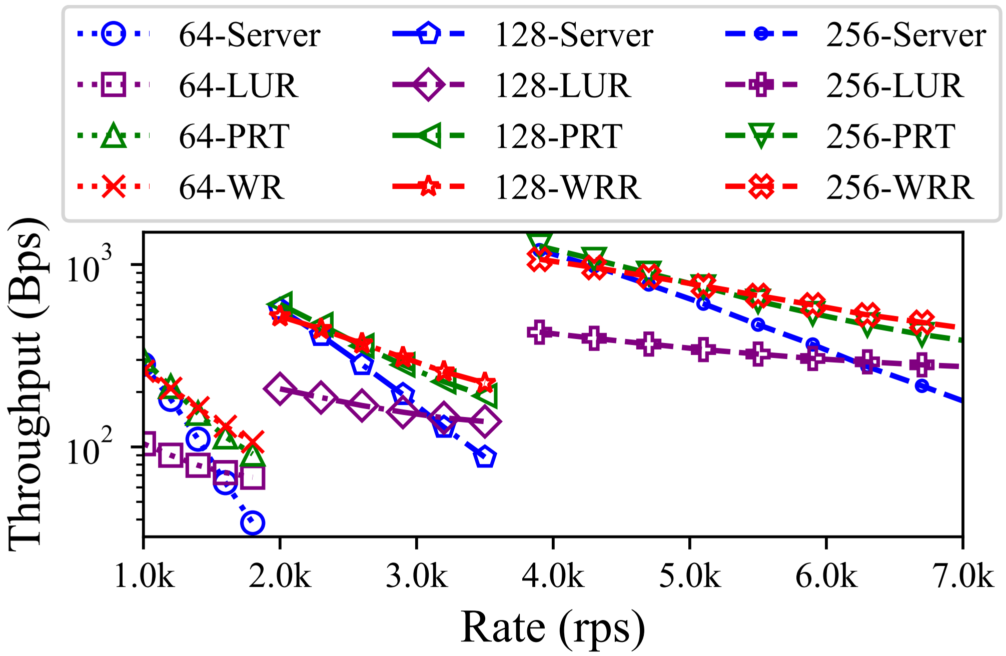


Figure 5.12: Average throughput of different policies.

With the exception of LUR when employed with 64 servers, all policies demonstrate similar levels of throughput performance up to a rate of 1.0k RPS. However, as the workload surpasses this threshold, the baseline policy's throughput starts to decrease. This reduction can be attributed to the fact that, on average, servers begin to receive requests at a rate close to their capacity, which is approximately 16 RPS in line with our configuration (1.0k divided by 64 equals roughly 16 RPS).

In contrast, the LUR policy with 64servers not only surpasses the baseline but also maintains its superior performance up to a rate of 1.6k RPS, after which its throughput also begins to decline. As for the remaining two policies, PRT and WRR, they consistently exhibit higher throughput performance across all tested rates, with WRR consistently outperforming PRT.

A consistent performance trend is evident when considering 128 and 256 servers, as illustrated in Figure 5.12. In these scenarios, LUR initially exhibits suboptimal performance up to rates of 3.3k RPS and 6.6k RPS for 128 and 256 servers, respectively. However, it gradually outperforms the baseline as the workload rates increase. Simultaneously, the PRT and WRR policies prove to be the most efficient options for managing heavier workloads, with WRR showing a slight performance advantage.

In Figure 5.13, we analyze the 99$^{\text{th}}$ percentile E2E delay of different policies, and

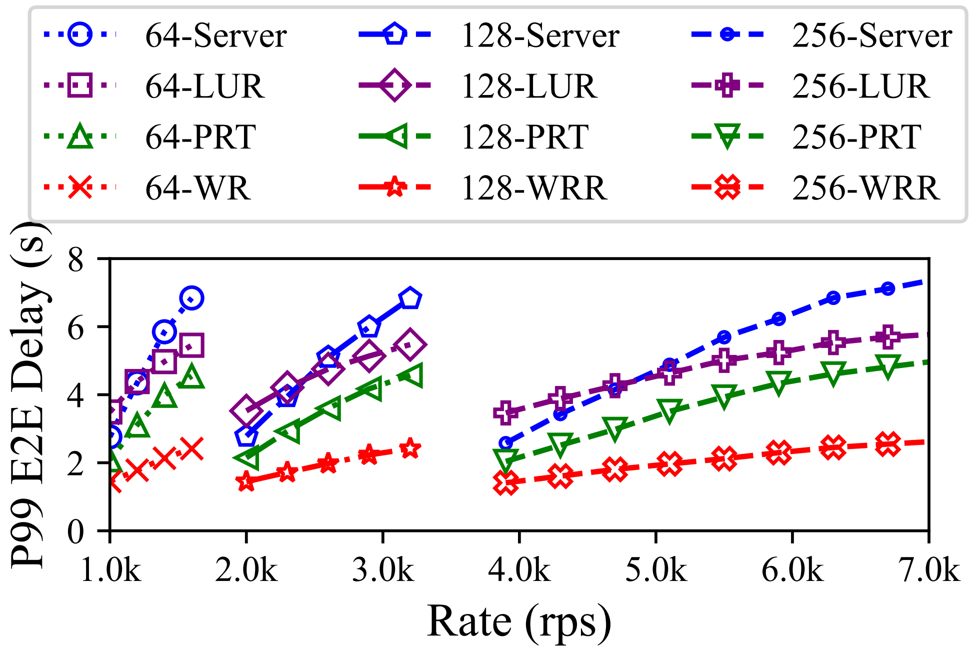


Figure 5.13: Average throughput of different policies.

consistently, *P4Hauler*'s policies surpass the baseline. When considering a workload of 1.0k RPS with 64 servers, *P4Hauler* impressively achieves swift responses, with the WRR policy providing responses in under 1.5 seconds and the PRT policy within 2 seconds. In contrast, the baseline struggles, with request processing times averaging around 2.7 seconds. As we increase the workload to 1.3k RPS and beyond, all of *P4Hauler*'s policies clearly outperform the baseline in terms of E2E delay.

This trend holds as we scale up to 128 and 256 servers. For instance, with 128 servers handling 2.0k RPS, the WRR policy exhibits exceptional performance, resulting in an E2E delay of just 1.7 seconds, making it the top choice. The PRT policy follows closely as the second-best option, while the baseline lags, offering the worst E2E delay. Even at 3.3k RPS for 128 servers, WRR demonstrates impressive resilience, underscoring its ability to efficiently handle high workloads. The same pattern continues with 256 servers, where WRR can manage requests at rates of up to 7.0k RPS, while the E2E delays for other policies increase to 5 seconds or more.

By analyzing Figure 5.14, which depicts the variances in CPU utilization across devices, we can observe how evenly each policy distributes the load among the available devices. To evaluate this, we subjected the system to the rate of 1300 RPS for the 64-server scenario and 2600 RPS for the 128-server. Note that we selected these

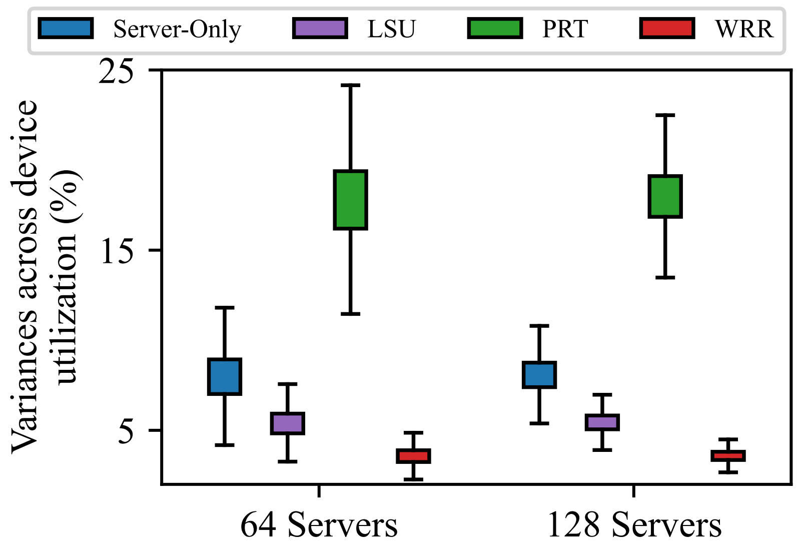


Figure 5.14: Variances of CPU utilization

rates as they overwhelm all servers in the server-only policy. As shown in Figure 5.14, the baseline policy, which utilizes ECMP for request dispatching, exhibits variances of approximately 10%. In contrast, LUR demonstrates a narrower range of variances as it aims to utilize the least utilized device among the selected server and its accelerators. Furthermore, PRT displays the widest range of variances since it avoids using the accelerators until the server reaches full utilization. Lastly, WRR showcases the smallest range of variances, as the policy distributes the load based on the capacity of each device.

5.8 ***P4Hauler* vs. *P4Mite***

In this section, we aim to compare *P4Hauler* and *P4Mite* from various angles. Although both perform load balancing at the granularity of accelerators, there are several distinctions between them.

In terms of flexibility, *P4Mite* is a more lightweight switch-based load balancer that only supports a fixed algorithm for load distribution. In contrast, *P4Hauler* offers a wide range of policies, providing network administrators with a broader selection. This flexibility represents a key advantage of *P4Hauler* over *P4Mite*.

Furthermore, *P4Mite* faces a limitation in executing policies that involve a significant number of arithmetic operations, primarily due to the restricted resources available in the data plane. Typically, intricate operations are mapped to match+action tables for offloading to the programmable switch, but when dealing with a multitude of permutations, this approach can exceed the memory limit. In contrast, *P4Hauler* proposes the use of programmable blocks that neatly integrate into the pipeline. Its improved hierarchical design empowers it to handle complex policies effectively.

The last notable difference is in how these systems operate. In *P4Mite*, the agents modify the switch's status to enforce the desired policy. On the other hand, in *P4Hauler*, the agents are solely responsible for gathering and transmitting measured parameters to the switch. The decision-making process for load balancing is carried out within a programmed block in the switch itself.

In summary, *P4Mite* is a lightweight load balancer that supports only a single accelerator-aware algorithm. On the other hand, *P4Hauler* is a generalized load balancer that can accommodate various policies using the data plane building block, i.e., *P4Hauler is policy agnostic*. Also, it accommodates new policies on-the-fly without interrupting the switch's normal operation. Finally, *P4Hauler* can measure resource usage at a fine granularity, which is missing in *P4Mite*. In fact, it is an instance of the generalized *P4Hauler*. All these benefits of *P4Hauler* come at the price of a slightly more resource demand compared to *P4Mite*. More specifically, we observed that *P4Hauler* utilizes 16.7% of ALU and 7.6% of Map RAM, out of whole switch's capacity, while *P4Mite* needs 2.1% and 5.6% of respective resources at the same circumstance. More details are available in Sections 4.5.5 and 5.7.3.

5.9 Major Conclusions of ***P4Hauler***

In this chapter, we presented a framework called *P4Hauler*, which supports a wide range of policies for load balancing at the per-accelerator level in data centers. Unlike previous research, *P4Hauler* does not hardcode the policies in the switch, allowing network administrators to switch between them dynamically. Similar to the design of *P4Mite*, the agents in *P4Hauler* gather and transmit resource usage information (such as CPU, Memory, and network bandwidth) to the switch, which stores this information in memory. Additionally, the switch contains two kinds of configurable blocks *minimum finder* block and the *round-robin scheduler*. Network administrators can activate different policies by utilizing the information and the configurable blocks.

To implement *P4Hauler*, we created a prototype using P4-16 and Python3. The prototype operates on a testbed that includes a Tofino switch and two servers each with a SmartNIC. We assessed the prototype using various machine-learning applications to demonstrate the benefits of different policies. In other words, our examination revealed that there is no single ideal policy for load balancing. For example, in our setup, PRT demonstrates superior performance at lower rates, whereas WRR proves to be the most effective at higher rates. It should be noted that based on factors such as computing demand, incoming load, and accelerators' capacity, a certain policy outperformed others.

Chapter 6

P4Wise

In this chapter, we introduce *P4Wise*, a load balancer that leverages Reinforcement Learning (RL) to achieve load balancing at the accelerator level. Its goal is to determine the most effective load balancing policies. Our previous research in *P4Hauler* (Chapter 5) led us to a crucial realization: there is no universal policy suitable for all networking conditions, e.g., change in applications, their traffic rates, number of accelerators or servers, and their statuses. Instead, the ideal policy must adapt to the system's current state. To put it differently, any alteration in the system can necessitate an adjustment to the load balancing policy. We observed this phenomenon when traffic rate changes in *P4Hauler*, necessitating the switch to different policies to achieve optimal results. The complexity increases when significant changes occur in the network environment, such as adding or removing devices or upgrading existing ones. In such cases, the configurations, including assigned weights for WRR, must also be updated based on the new device capacity. In summary, *P4Wise* employs RL to tackle this challenge by dynamically monitoring the system's environment and continuously updating its policies and configurations as needed.

Just like *P4Mite* and *P4Hauler*, *P4Wise* depends on agents for gathering information. While *P4Mite* and *P4Hauler* follow the execution of predefined policies, with *P4Mite* using hard-coded policies and *P4Hauler* employing the configurable blocks to activate policies, *P4Wise* takes a different approach. *P4Wise* dynamically selects the optimal policy within the system based on the observed changes to achieve the best performance.

6.1 Motivation

In a broader context, machine learning proves advantageous when confronting problems that entail the processing of vast datasets, making predictions or decisions, and when it is challenging to explicitly program a solution due to the complexity or variability of the data. It is essential to carefully select the appropriate machine learning approach and techniques tailored to the specific problem at hand.

We made some efforts to construct a dataset for load balancing within our simulation, enabling the utilization of supervised learning techniques for load balancing. As detailed in Section 5.7, our evaluation results highlight that either PRT or WRR yield the most promising outcomes. Consequently, when creating the dataset, we ensured an equitable distribution of statistics for each policy to establish a well-balanced dataset. Subsequently, we employed a multi-layer perceptron neural network comprising three layers, each containing ten nodes. This model exhibited an impressive 97% accuracy in predicting the correct weight for 128 servers equipped with one SmartNIC on each.

However, the model's performance diminishes when the rate undergoes changes. For instance, substituting the SmartNIC with a 20% more powerful one led to a drop in model accuracy to 80%. In another test scenario, where a second SmartNIC was added to the system for each server, the accuracy deteriorated even further, reaching 66%. In this case, the model struggled to consistently select appropriate weights to effectively utilize the second SmartNIC.

The above evaluation gives us the following concluding remarks. Both supervised and unsupervised learning necessitate substantial data for model training. However, regrettably, there is no publicly accessible dataset for load balancing. Crafting a comprehensive dataset encompassing all possible combinations, including the number of servers, accelerators, their capacities, and so forth, would be a tedious task. In such cases of lack of appropriate datasets, we require an alternative of supervised and unsupervised learning. Furthermore, if there is a dataset available for retraining the supervised/unsupervised models, its utility comes with the requirement of redeploying the model in the system after training.

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. Moreover, it is feasible to deploy RL-based systems in the data plane as demonstrated in [72], which is not the case for deep neural networks. Integrating neural networks into the data plane poses challenges, primarily stemming from the absence of floating-point operations, necessitating the use of techniques such as binarization.

6.2 Overview of *P4Wise*

Figure 6.1 provides an overview of the *P4Wise* system. To enhance clarity, we have omitted the depiction of accelerators for the servers (hosts) on the left side of the figure. Just like in the case of *P4Mite* and *P4Hauler*, it is important to note that each host can be equipped with one or multiple accelerators. Furthermore, the same agents introduced for *P4Hauler* are responsible for monitoring both the servers and their accelerator usage, as well as transmitting updates to the switch. For the sake of simplicity, we assume that all servers possess uniform computing power. Similarly, they are equipped with accelerators having similar capacities.

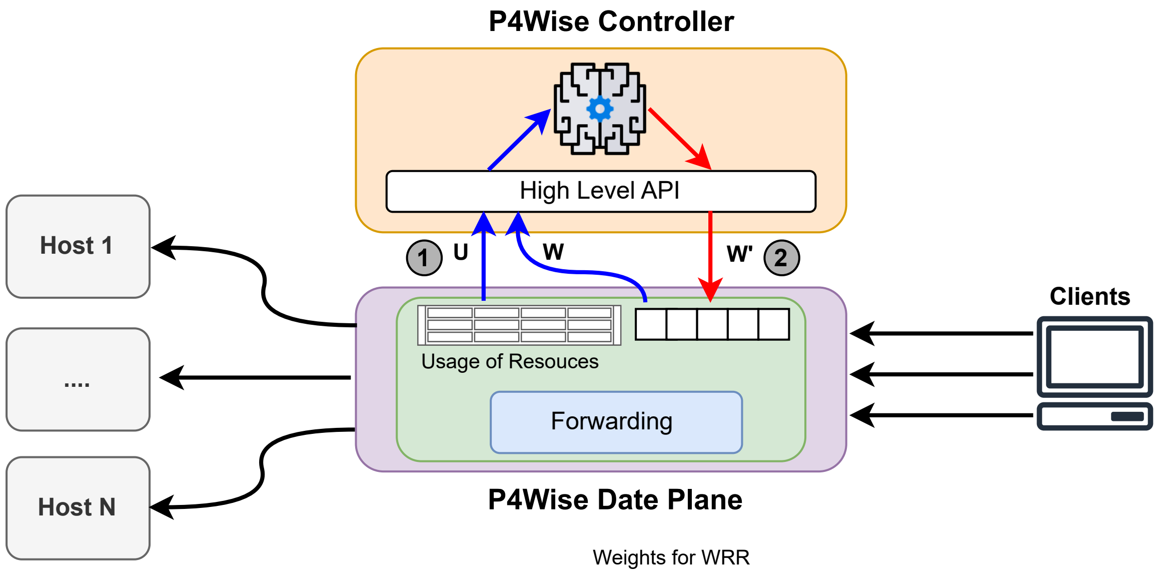


Figure 6.1: *P4Wise* overview.

Two essential components of *P4Hauler*'s data plane are necessary for *P4Wise*: the data structure used for storing statuses and the *round-robin scheduler*. Conversely, *P4Wise*'s controller is responsible for executing the RL agent, which utilizes

Q-Learning approach to decide for load balancing. The RL agent relies on a set of high-level APIs within the controller for communication with the data plane, and the specific APIs used may vary depending on the switches in use. The agent uses these APIs to retrieve information from the data plane, and push the next step action as they are shown in two steps in Figure 6.1. Initially, the RL agent acquires resource usage data and the active weights from the data plane (the solid blue arrows). This data represents the current system status denoted as U and W. Using this information, the RL agent first calculates the reward and then takes an action that generates W', and applies the new weights to *P4Wise*'s switch through the APIs (the solid red arrows). These weights determine the load balancing policy, and it is important to note that varying these weights allows for the emulation of different policies. For example, we can mimic the prioritization policy by assigning a weight of 0 to a particular device and other devices with non-zero weights. Several noteworthy works in the same area, such as [70, 71], have adopted a similar approach to adjust the weights and apply the optimal policy for load balancing in the system.

6.3 *P4Wise*'s Design

In this section, we present our Reinforcement Learning (RL) model designed to efficiently allocate computational load across a network of servers and their respective accelerators. Our primary focus is on formulating a model that strategically determines the optimal weights for the Weighted Round Robin (WRR) within the data plane and changes the weights to emulate different policies (e.g., PRT with $6:0$ or WRR with $6:1$ weights) due to the fact that we can mimic different policies by having a WRR policy that gets updated weights from the RL model to switch between policies, similar to [70, 71]. This weight assignment aims to optimize End-to-End (E2E) latency, particularly for compute-intensive applications, such as Machine Learning (ML), Deep Learning (DL), web servers, and financial analytics, which tend to impose significant computational demands.

Our approach involves enhancing the RL agent's knowledge by incorporating information about various network resources and dynamically updating WRR weights. This allows us to adapt the best weights for the chosen applications, ensuring that the network resources are efficiently utilized. Note that our approach shares similarities with standard RL-based load balancing methods as described in [70, 71, 73]. However, there are two key distinctions to consider. The first distinction is our emphasis on accelerator-aware granularity. Unlike other approaches, which handle accelerators similarly to legacy servers, our approach takes into account the specific characteristics of accelerators. The second significant difference is in the primary objective of previous research. Previous studies employed RL for load balancing to distribute network traffic across multiple paths. In contrast, our focus is on distributing CPU-intensive applications across multiple computing resources, under the assumption that the network is not the limiting factor in our system.

When dealing with a distinct application type that has specific resource requirements, such as memory or network bandwidth, it becomes necessary to make several adjustments. These adjustments encompass revising the reward function, refining the policy, and fine-tuning the model to determine the appropriate resource allocation weights for that particular application.

Finally, it is essential to ensure that *P4Wise*'s data plane load balancer, similar to *P4Mite* and *P4Hauler*, maintains per-connection consistency for forwarded requests. This means that all packets belonging to a specific request must be directed to the same server. Any deviation from this rule can lead to system overloads caused by retransmissions or error resolution. Additionally, *P4Wise* faces the challenge of efficiently managing a substantial number of connections using the available resources, such as programmable switches' registers. To address these critical requirements, we have employed the same techniques and approaches that have been established in the state-of-the-art solutions, including using a bloom filter.

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.

6.3.1 Formal Problem Formulation

State Space (S): States are the elements that encompass the characteristics of the environment observed by the RL agent. Essentially, the agent sees these attributes and based on this information takes actions to earn rewards. States can be presented as an array of numerical value, a data structure, or various other forms. In our context, we represent states of the system consisting of n resources (R) using two vectors of numerical values as given below:

$$S = \{U, W\} \tag{6.1}$$

Where:

$$\begin{aligned} U &= [u_1, u_2, ..., u_n] \\ W &= [w_1, w_2, ..., w_n] \\ n &\in R \\ 0 <= u_i &<= 100 \\ 0 <= w_i &<= w_{max} \end{aligned} \tag{6.2}$$

In the provided equations, resource utilization is represented as a value ranging from 0 to 100, while the weights can take on any positive numerical values. Determining the maximum weight involves a trade-off: a wider range of allowable weights provides greater precision in assigning them, but also introduces increased complexity in the weight assignment process. Consequently, the *P4Wise*'s agent necessitates more iterations for convergence.

Action Space (A): In the context of *P4Wise*, when the RL agent takes an action, it means that it assigns weights 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 by constraining the action space to three actions: 1) incrementing accelerators' weights by 1, 2) decrementing accelerators' weights by 1, and 3) maintaining the current weights. With this limited set of actions, the agent not only explores fewer options during training but also retains the ability to apply a diverse range of weights to the system.

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 $-1$
```

Reward Function (R): Algorithm 1 outlines the process for determining the reward. It begins by reading the current state at line 1. Afterward, having the utilization of resources and their weights, the agent calculates the 99th latency [1] and average utilization of the hosts, at lines 2 and 3, respectively.

Following the described calculations, three conditions are checked to determine whether a terminal state has been reached. In a terminal state, the agent has either selected an incorrect set of weights resulting in system failure, or identified the optimal weights that demonstrate the best performance. In non-terminal states, however, the allocated weights do not excessively utilize resources, even though the E2E latency is not optimized. The first `if` statement at line 4, assessing whether utilization has exceeded 100%, signifies the identification of an incorrect set of weights, indicating a potential system failure. On the other hand, the subsequent two `if` statements

[1]We use this approach in our simulator only. In a real implementation, the agents running on the hosts and accelerators can measure the 99th latency, and send it along with other measurements to the switch.

at lines 7 and 10 are examining the selection of optimal weights. The `if` statement at line 7 assesses whether optimal latency is achieved by exclusively utilizing the hosts. In the next `if` statement, at line 10, it determines if all resources are used, the more powerful devices are not underutilized, as employing the wimpy devices instead them causes severe performance degradation. This assessment is influenced by our findings in *P4Hauler*, where prioritizing hosts was necessary if they possessed sufficient capacity to manage the load. Finally, at the end of 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.

Policy (π): Algorithm 2 depicts the ϵ-greedy policy employed by the agent which explores the environment with a probability of ϵ. As previously indicated, the agent utilizes a Q-Learning approach for decision-making. Therefore, with high probability at each step, it chooses the action with the maximum Q-Value, while with lower probability, a random action should be selected for exploration. Keeping this in mind, at lines 1 to 3, we initially retrieve the current state of the system, then obtain the Q-Values corresponding to the current state, and finally generate a random number. Subsequently, the `if` statement determines whether the action with the highest Q-Value should be taken or a random one. If the random number is greater than ϵ the algorithm returns the action with the highest Q-Value, otherwise a random action is returned.

Algorithm 2 Policy π, A ϵ-greedy approach

```
1: $U, W \leftarrow read\_current\_state()$
2: $Q\_Values \leftarrow Q\_Table\_read\_row(U, W)$
3: $rand \leftarrow random()$
4: if $rand > \epsilon$ then
5:     return $action\_with\_max\_Q\_Value(Q\_Values)$
6: else
7:     return $random\_action(Q\_Values)$
8: end if
```

6.3.2 Initialization

In the initialization phase of *P4Wise*, two components are initialized. The first component is the vector of weights, where prioritization is given to the servers (hosts) by assigning the maximum weight to all of them and zero to accelerators. In essence, *P4Wise* initiates load balancing by emulating PRT, with the understanding that weights will be updated later.

The second component is the Q-Table, which presents the Q-Values of the three actions at each step. It is crucial to emphasize that the availability of prioritized devices is key for the agent's decision-making process. Based on our assumption that all servers have identical capacities and an equal weight is assigned to all servers, we can understand the state of the system through the average utilization of servers. Another critical factor is whether accelerators have been incorporated. If so, it is imperative to ensure that the servers are not underutilized.

Given this consideration, as illustrated in Table 6.1, we partition the potential range for average utilization into N sub-ranges with intervals of $\frac{100}{N}$. For each interval, the accelerators may or may not be used, thereby doubling the total number of rows to $2 \times N$. A larger value for N provides increased granularity at the expense of expanding the Q-Table. Last but not least, all Q-Values are initialized to 0.

Table 6.1: initialized Q-Table

States		**Q-Values**		
Average Utilization Ranges	**Are the accelerators involved?**	**Action 1**	**Action 2**	**Action 3**
$[0, \frac{100}{N})$	No	0	0	0
$[0, \frac{100}{N})$	Yes	0	0	0
$[\frac{100}{N}, 2 \times \frac{100}{N})$	No	0	0	0
$[\frac{100}{N}, 2 \times \frac{100}{N})$	Yes	0	0	0
...	...	...	...	...
$[(N-1) \times \frac{100}{N}, 100]$	No	0	0	0
$[(N-1) \times \frac{100}{N}, 100]$	Yes	0	0	0

6.3.3 Load Balancing with *P4Wise*

Algorithm 3 outlines the load balancing procedure executed by *P4Wise*'s agent. At lines 1 and 2, the Q-Table (QT) and the weights (W) are initialized. Following the initialization, the agent commences the first loop, indicated by a `while` at line 3, which actively observes the environment. Given the `if` in line 4, we detect the load (i.e., applied rate) changes in the system, by monitoring the utilization, which starts the load-balancing procedure in the system. By calculating the reward, we detect if we need to adjust the weights. If so, the second `while` statement (line 6), is executed by the agent until a terminal state is reached, which is either the optimal weight or an incorrect weight. In this second loop, the agent initially selects an action (shown as a) using the policy π (line 6). Subsequently, the new weights corresponding to the chosen action are calculated (line 7). These new weights are then applied to the system, and the new states, denoted as U', W', are collected from the environment (line 8). The agent proceeds to calculate the reward in the new state (line 9). Finally, the agent updates the Q-Table values using the Bellman Equation [61] to optimize the Q-Values (line 10).

Algorithm 3 Load Balancing with *P4Wise*

```
1: initialize QT
2: initialize W
3: while True do
4:     if change_state() then
5:         while ! final_state(U, W) do
6:             $a = \pi(U,\ W)$
7:             $W = update\_weight(W,\ action)$
8:             $U',\ W' = apply\_the\_weights(W)$
9:             $r = calculate\_reward(U',\ W')$
10:            $QT(U,W,a) = QT(U,W,a) + \alpha[r + \gamma\ max(QT(U',W')) - QT(U,W,a)]$
11:            $U,\ W = U',\ W'$
12:        end while
13:    end if
14: end while
```

6.4 *P4Wise*'s Implementation

We have developed a proof of concept prototype for the *P4Wise* controller in Python, consisting of approximately 100 lines of code. This prototype is accessible to the public via the link provided in [113]. The controller exhibits the capability to collect data from either the testbed shown in Section 5.6.1 or the simulated environment introduced in Section 5.6.2. Subsequently, it can apply weight adjustments to the system. In the following section, we will conduct a performance evaluation of *P4Wise* within our simulator to assess its scalability and effectiveness.

In the following experiments, we consider 64, 128, or 256 servers, and two SmartNICs on each server, unless specified otherwise. The computing capacities of the CPUs and SmartNICs are in line with our testbed's devices.

6.5 *P4Wise*'s Evaluation

We initiate this section by conducting a series of experiments aimed at identifying the optimal parameters for fine-tuning our model in subsection 6.5.1. Certain parameters, such as α and γ, as well as thresholds for latency and utilization, need to be determined. Once the optimal model is identified, we proceed to evaluate *P4Wise*'s trained model at various scales compared to *P4Hauler* in subsection 6.5.2. Finally, in subsection 6.5.3, we compare *P4Wise* with a supervised learning approach.

6.5.1 *P4Wise* Tuning

We divide this subsection into two main parts. In the first part, we concentrate on tuning the parameters in the reward function, while in the second part, we explore the environment to determine the optimal values for α and γ in Bellman equation. Regarding the policy π, we set ϵ to 0.1, indicating that it selects the action with the highest Q-Value in 90%, while in 10% of situations, the agent explores the environment through a randomly chosen action. A high value for ϵ leads to a drop in performance during inference as it generates random actions. However, it is necessary to have such exploration during training for the agent.

Tuning the Reward Function

Two thresholds in the reward function need to be determined: *latency_threshold* and *utilization_threshold*. According to the evaluation of *P4Hauler* in 5.7, we observed that the system's 99^{th} percentile E2E latency is consistently less than 1 second for KNN in the best policy (PRT or WRR). Consequently, we consider this value as 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 6.2. Instead of *P4Wise*'s agent, which utilizes Q-Learning, the agent in this experiment only switches between PRT and WRR policies. The goal is to find out 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 99th 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 utilization, the 99th percentile E2E latency reaches to $1s$ as well. We adopted *utilization_threshold* = 60% for all subsequent experiments in this section.

The final parameter to set in the reward function is *max_reward*, which we have considered as 10 in our experiments. Since we receive a -1 reward in each step, any number greater than 1 will suffice for this parameter. It just needs to show whether we reach a terminal state. Put differently, the agent should receive significant rewards or penalties in terminal states based on the outcomes, while encountering minor punishments in intermediate states.

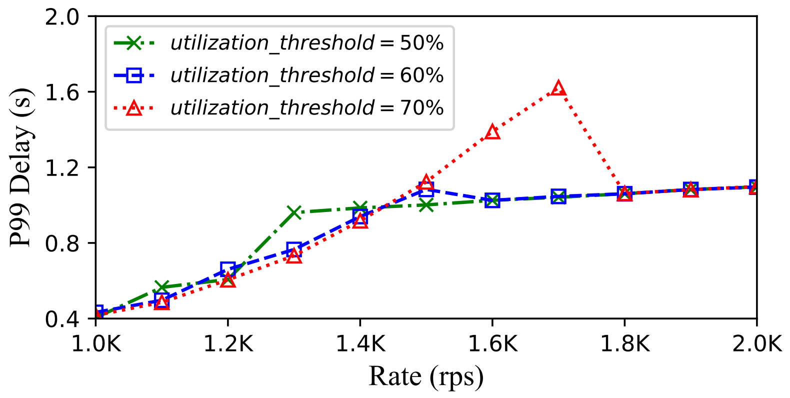


Figure 6.2: Tuning *utilization_threshold* for *P4Wise*.

Tuning Bellman Equation's Parameters

There are two parameters in the Bellman equation, and both can take values between 0 and 1. The first one is α or the `learning rate`, where a smaller value for α adds more granularity at the expense of requiring more time for training. The second parameter is γ or the `discount rate`, indicating the importance of the next state in the learning process.

In our experiments, *P4Wise* converges with different values for the mentioned parameters when setting a coarse value for γ. However, the number of iterations required for convergence may vary. Figure 6.3 illustrates the sum of the 100 most recent actions for different values. Given that we consider a maximum reward of 10, the summation can reach up to $100 \times 10 = 1000$. Based on our observations, *P4Wise* converges gradually with $\alpha = 0.3$, while with $\alpha = 0.5$, the system converges at a faster rate. In the remainder of the manuscript, we consider $\alpha = \gamma = 0.5$.

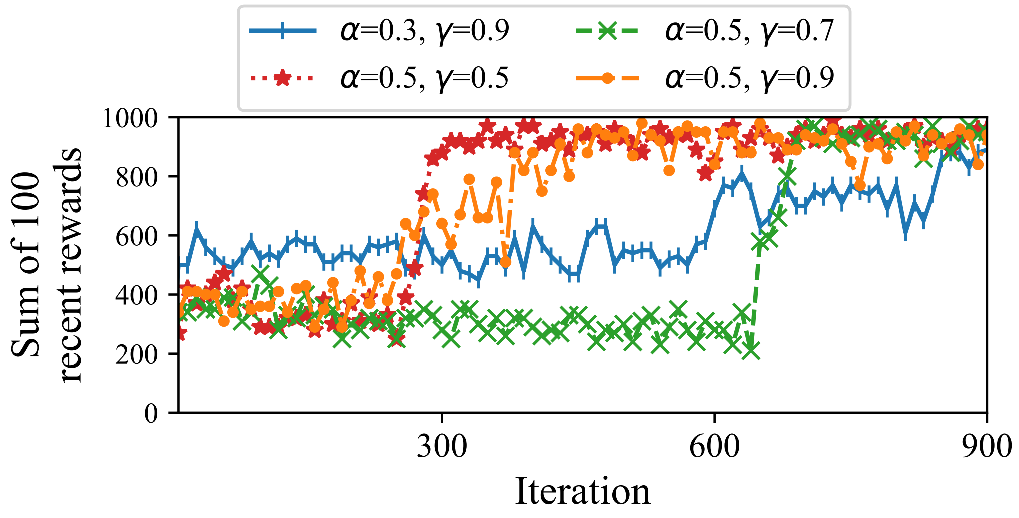


Figure 6.3: Tuning Bellman equation's parameters.

6.5.2 *P4Wise* at Different Scales

In this subsection, we evaluate the trained *P4Wise*'s agent with the specified parameters for 128 servers and two SmartNICs on each, across different scales. We vary the number of servers, and for each evaluation, we compare *P4Wise* with *P4Hauler*'s best policies.

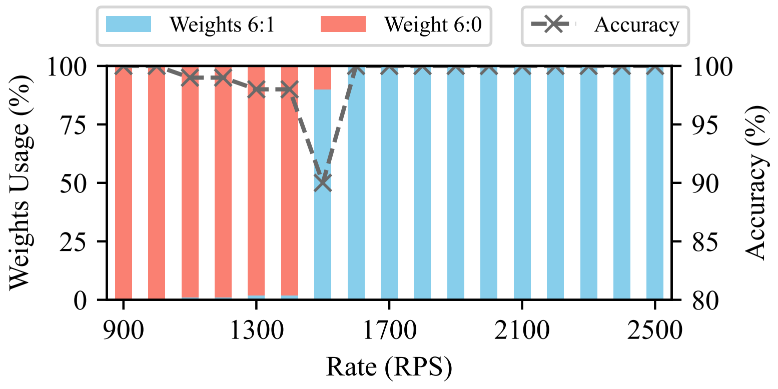


Figure 6.4: *P4Wise*'s outcome for 128 servers after training.

Figure 6.4 depicts the performance of *P4Wise*'s agent with the specified parameters after training. On the plot's weights usage axis, we observe the percentage of weight usage between 6 : 1 and 6 : 0 for various rates, while the secondary axis displays the accuracy of correct weight selection by *P4Wise*. We assume that 6 : 0 is the correct weight for rates less than 1.4 RPS, and 6 : 1 for higher rates. With this consideration, at the 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 various other weights were tested during training; however, none of them yielded promising results.

Figure 6.5 provides the 99th percentile delay and throughput for a data center with 64 servers. As anticipated, *P4Wise* exhibits a 99th percentile delay that closely resembles the optimal policy, PRT, at lower rates (below 700 RPS), and converges toward the behavior of WRR at higher rates (above 700 RPS).

The same pattern is observed at various rates for a system with 256 servers, as depicted in Figure 6.6. 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.

Figure 6.7 presents the chosen weights and the accuracy of weight selection for *P4Wise* in 256-server scenario. The observed weights align with our findings in Figure 6.6. Up to a rate of 3k RPS, the weights align with PRT with an accuracy of 95%. However, at higher rates, 6 : 1 is applied.

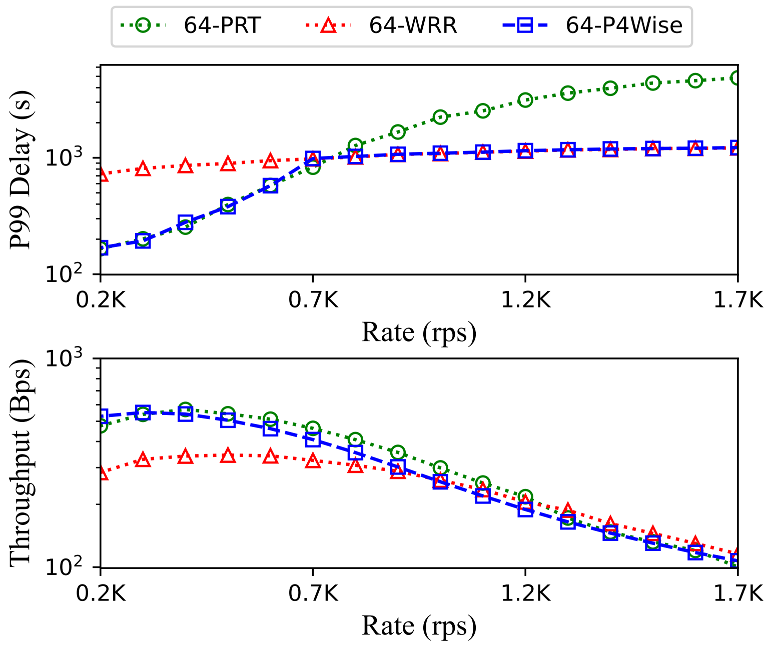


Figure 6.5: Balancing the load among 64 servers with *P4Wise*.

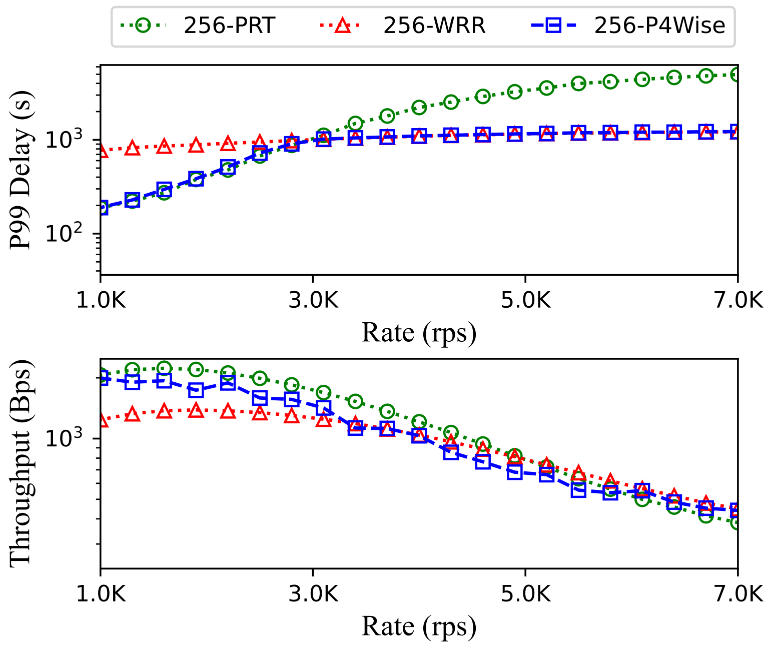


Figure 6.6: Balancing the load among 256 servers with *P4Wise*.

6.5.3 *P4Wise* vs. Supervised Learning

One of the advantages of *P4Wise* mentioned in section 6.1 is its capability to operate in an online manner. In other words, *P4Wise* can actively assess the reward of actions

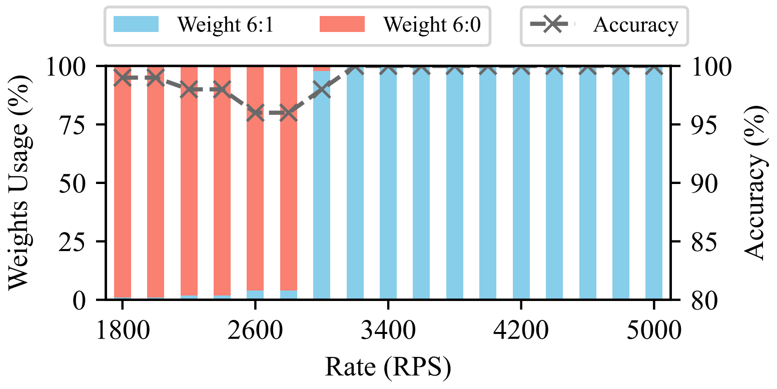


Figure 6.7: *P4Wise*'s outcome for 256 servers.

to verify their correctness. In such a scenario, *P4Wise* can dynamically respond to changes in the environment. In contrast, a supervised learning solution necessitates data collection, training, and model redeployment. In this experiment, as illustrated in Figure 6.8, we consider 128 servers, each equipped with one SmartNIC. After data collection, we trained a neural network with three 10-node layers to choose among a few predefined weights. The accuracy of the neural network in this setup is approximately 97%. On the other hand, we employed *P4Wise* in an online manner, and the accuracy of *P4Wise* with this configuration exceeds 90%. At $time = T$, we added a second SmartNIC to all servers, and we observed that the accuracy of the supervised learning model dropped to 65%. Nevertheless, *P4Wise* requires approximately 250 iterations according to Figure 6.3 when we reset *P4Wise*'s Q-Table and initiate training. After the required time, *P4Wise* reaches to more than 90% accuracy.

6.6 Major Conclusions of *P4Wise*

Based on our observations in the *P4Hauler* section, we recognized that there is no single ideal policy that consistently delivers promising results across all network scenarios. Specifically, we observed that network administrators must adapt and activate different policies based on the prevailing system load. With this in mind, we introduce *P4Wise* in this section, which leverages a Reinforcement Learning approach to determine the optimal weights that mimic the most effective policy for the current system conditions.

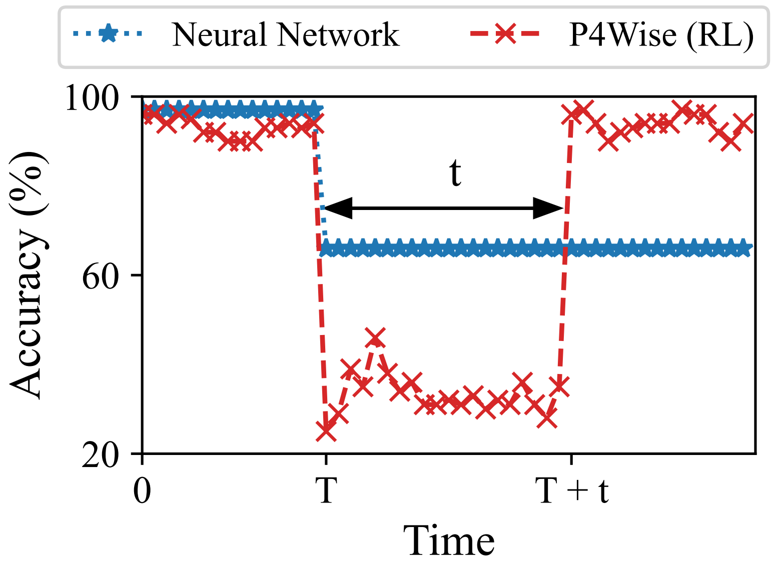


Figure 6.8: *P4Wise* vs. supervised learning approach.

Within *P4Wise*'s data plane, we integrate two components inherited from *P4Hauler*: a *round-robin scheduler* and a data structure that stores information on resource utilization measured by agents on servers and their accelerators. Additionally, a crucial RL agent operates within *P4Wise*, responsible for selecting the best weights. This RL agent utilizes available APIs to retrieve information from the data plane and uses it for training the RL model. The RL model follows a trial-and-error approach to discover the most suitable weights for the system.

To evaluate the performance of *P4Wise*, we have implemented a proof-of-concept implementation in Python, which operates within our simulator. Once *P4Wise* is properly fine-tuned, for different scales, it can intelligently apply the appropriate policy and adapt its weights for the specific system condition without experiencing the same performance degradation as *P4Hauler* under similar conditions. According to our quantitative results, *P4Wise* consistently selects the appropriate weights with an accuracy of at least 90%. Furthermore, not only is *P4Wise* effective at different scales but it can also dynamically respond to changes in the environment by adapting its load balancing approach accordingly.

Chapter 7

Conclusion and Future Work

In the subsequent chapter, we draw our research to a conclusion by considering the findings of *P4Hauler*, *P4Mite*, and *P4Wise*. Following that, we delve into several potential avenues for future work based on the limitations identified in the current study.

7.1 Conclusions

Although network speed and data raised exponentially in the recent decade, CPU enhancement reached a plateau. With these in mind, system designers introduced accelerators to bring more computational power for bearing massive data. Load balancers are a possible solution to split the computation among the CPUs and the accelerators. However, accelerators have different architectures making load balancing challenging.

In our work, first, we evaluated the possibility of using a programmable switch, like Tofino, for per-accelerator load balancing in *P4Mite*. Considering the accelerator capacity, we observe promising results in the initial implementation. For instance, the accelerator's computational power is one-fifth of CPU's in our testbed, and by using *P4Mite*, the system can handle up to 20% load and processes requests 50% faster. *P4Mite* achieves the improvements while using negligible resources in the programmable switch according to our evaluation.

Seeing favourable outcomes, we extended the load balancer and introduced *P4Hauler*, which supports a few well-known policies and is on-the-fly configurable. That is to say that, in contrast to other in-network load balancers, the system admin can switch between the policies without rebooting the switch. The data plane is also extendable to support additional policies by implementing other programmable blocks if required. Through quantitative evaluations of four different applications in terms of their computational demands, we discovered that there is no universally ideal policy for all scenarios. Instead, the choice of policy should be based on the application's demand and the applied load to the system. This observation motivated us to develop *P4Wise*, a reinforcement learning-based hybrid in-network load balancer. *P4Wise* retains all functionality of *P4Hauler* and includes a learning module in the control plane to learn network dynamics and on-the-fly switch between load balancing policies within the lifetime of running applications.

7.2 Future Work

This section summarizes all possible future extensions of the proposed thesis work.

Utilizing other accelerators. Although 20% improvement is satisfactory due to the utilized accelerator's computation capability, we could improve the performance even more by forwarding a portion of the load into other accelerators like the GPU. From the design perspective, *P4Hauler* and *P4Mite* supports other accelerators, assigning a flow or request to them. From the implementation on the testbed, however, the use case applications must be able to operate on GPU as well. By utilizing more accelerators, the overall performance will improve, specifically for adopting GPU for machine learning and deep learning applications.

Policies and applications. The current two building blocks of *P4Hauler* can support a set of load-balancing policies but cannot cover all possible ones, e.g., power-of-two choices. Thus, another possible extension is to explore other applications and suitable policies for them. For example, we plan to accommodate video streaming applications (e.g., Zoom or Teams). Also, if we want to support applications demanding microseconds latency deadline, we have to extend the current design of *P4Hauler*.

Performance metrics and workload. The chosen ML applications are computation-intensive rather than bandwidth-heavy; thus, in future, we can think of such applications. Also, we can measure the performance of load balancers in terms of their operation time while considering mixed or skewed loads for various types of applications.

Exploring rewards functions and RL algorithms. *P4Wise* deploys a simple reward function and policy. Though both work sufficiently for the current load balancing problem, it is worth exploring other reward functions and algorithms to see if we can find a combination of reward function and policy that offers the best performance. Also, we can consider deploying different RL algorithms. We will explore such an avenue in future.

Bibliography

[1] John Shalf. The future of computing beyond Moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[2] Rafael Vidal Aroca and Luiz Marcos Garcia Gonçalves. Towards green data centers: A comparison of x86 and arm architectures power efficiency. *Journal of Parallel and Distributed Computing*, 72(12):1770–1780, 2012.

[3] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on load distribution and the role of programmable switches. *ACM SIGCOMM Computer Communication Review*, 49(1):18–23, 2019.

[4] Xingyu Zhou, Ness Shroff, and Adam Wierman. Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers. *Performance Evaluation*, 145:102146, 2021.

[5] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-scale scheduler for RackSched computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240, 2020.

[6] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan 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*, pages 15–28, 2017.

[7] Carmine Rizzi, Zhiyuan Yao, Yoann Desmouceaux, Mark Townsley, and Thomas Heide Clausen. Charon: Load-aware load-balancing in P4, 2021.

[8] Gal Mendelson, Shay Vargaftik, Dean H Lorenz, Kathy Barabash, Isaac Keslassy, and Ariel Orda. Load balancing with jet: just enough tracking for connection consistency. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 191–204, 2021.

[9] Chaoliang Zeng, Layong Luo, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei 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.

[10] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *17th USENIX Symposium*

on Networked Systems Design and Implementation (NSDI 20), pages 667–683, 2020.

[11] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.

[12] Hesam Tajbakhsh, Ricardo Parizotto, Miguel Neves, Alberto Schaeffer-Filho, and Israat Haque. Accelerator-aware in-network load balancing for improved application performance. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022.

[13] David Hancock and Jacobus Van der Merwe. Hyper4: Using P4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 35–49, 2016.

[14] 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.

[15] Tajbakhsh Hesam. P4Hauler. https://github.com/PINetDalhousie/P4Hauler/, 2022.

[16] Tajbakhsh Hesam. P4Mite. https://github.com/PINetDalhousie/p4mite, 2022.

[17] Nick McKeown. Software-defined networking. *INFOCOM keynote talk*, 17(2):30–32, 2009.

[18] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[19] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[20] Cavium. *OpenFlow Switch Specification*. Open Networking Foundation, 2023 (accessed December 1, 2023). https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[21] Mohamad Darianian, Carey Williamson, and Israat Haque. Experimental evaluation of two openflow controllers. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 2017.

[22] Fangye Tang, Meysam Shojaee, and Israat Haque. Ace: an accurate and cost-effective measurement system in sdn, 2021.

[23] Meysam Shojaee, Miguel C. Neves, and Israat Haque. Safeguard: Congestion and memory-aware failure recovery in SD-WAN. In *16th International Conference on Network and Service Management, CNSM 2020, Izmir, Turkey, November 2-6, 2020*, pages 1–7. IEEE, 2020.

[24] Udaya Lekhala and Israat Haque. Piqos: A programmable and intelligent qos framework. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2019, Paris, France, April 29 - May 2, 2019*, pages 234–239. IEEE, 2019.

[25] Fangye Tang and Israat Haque. Remon: A resilient flow monitoring framework. In *Network Traffic Measurement and Analysis Conference, TMA 2019, Paris, France, June 19-21, 2019*, pages 137–144. IEEE, 2019.

[26] M. A. Moyeen, Fangye Tang, Dipon Saha, and Israat Haque. SD-FAST: A packet rerouting architecture in SDN. In *15th International Conference on Network and Service Management, CNSM 2019, Halifax, NS, Canada, October 21-25, 2019*, pages 1–7. IEEE, 2019.

[27] Israat Haque and M. A. Moyeen. Revive: A reliable software defined data plane failure recovery scheme. In Stefano Salsano, Roberto Riggio, Toufik Ahmed, Taghrid Samak, and Carlos Raniery Paula dos Santos, editors, *14th International Conference on Network and Service Management, CNSM 2018, Rome, Italy, November 5-9, 2018*, pages 268–274. IEEE Computer Society, 2018.

[28] I. Haque and N. Abu-Ghazaleh. Wireless software defined networking: a survey and taxonomy. *IEEE Communications Surveys and Tutorials*, 18(4):2713–2737, May 2016.

[29] Vinay Kolar, Israat T. Haque, Vikram P. Munishwar, and Nael B. Abu-Ghazaleh. Ctcv: Coordinated transport of correlated videos in smart camera networks. In *24th International Conference on Network Protocols (ICNP)*. IEEE, 2016.

[30] Israat Haque, Mohammed Nurujjaman, Janelle Harms, and Nael Abu-ghazaleh. SDSense: An agile and flexible SDN-based framework for wireless sensor networks. *The IEEE Transactions on Vehicular Technology*, 68(2):1866 – 1876, February 2019.

[31] Hossein Ghannadrezaii, Jean-François Bousquet, and Israat Haque. Cross-layer design for software-defined underwater acoustic networking. In *IEEE OCEANS*. IEEE, 2019.

[32] Dipon Saha, Meysam Shojaee, Michael Baddeley, and Israat Haque. An Energy-Aware SDN/NFV architecture for the internet of things. In *IFIP Networking 2020 Conference (IFIP Networking 2020)*, Paris, France, June 2020.

[33] Israat Haque and Dipon Saha. SoftIoT: A resource-aware sdn/nfv-based iot network. *The Elsevier Journal of Network and Computer Applications*, 193, Nov 2021.

[34] M. Kulkarni, M. Baddeley, and I. Haque. Embedded vs. external controllers in software-defined iot networks. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021.

[35] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, page 87–95, 2014.

[36] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Computing Surveys (CSUR)*, pages 1–36, 2021.

[37] Calin Cascaval and Dan Daly. P4 architectures. https://opennetworking.org/wp-content/uploads/2020/12/p4-ws-2017-p4-architectures.pdf, 2022.

[38] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[39] Google. Google AutoML. [n.d.] AutoML: Train high-quality custom machine learning models with minimal effort and machine learning expertise. https://cloud.google.com/automl/.

[40] Microsoft. Microsoft Brainwave. [n.d.]. Brainwave: a deep learning platform for real-time ai serving in the cloud. https://www.microsoft.com/en-us/research/project/project-brainwave/.

[41] Mellanox. *BlueField SmartNIC for Ethernet High Performance Ethernet Network Adapter Cards*. Intel, 2023 (accessed December 1, 2023). https://network.nvidia.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

[42] Broadcom. *Stingray PS225*. Broadcom, 2023 (accessed December 1, 2023). https://docs.broadcom.com/doc/PS225-PB.

[43] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–131, 2020.

[44] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. Fast and concurrent RDF queries using RDMA-assisted GPU graph exploration. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 651–664, 2018.

[45] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[47] Tong Xing, Hesam Tajbakhsh, Israat Haque, Michio Honda, and Antonio Barbalace. Towards portable end-to-end network performance characterization of smartnics. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 46–52, 2022.

[48] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 513–533, 2022.

[49] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[50] Xiao Wang, Haidong Yi, Jia Wang, Zhandong Liu, Yanbin Yin, and Han Zhang. GDASC: a GPU parallel-based web server for detecting hidden batch factors. *Bioinformatics*, 36(14):4211–4213, 2020.

[51] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 129–148, 2015.

[52] Sambit Kumar Mishra, Bibhudatta Sahoo, and Priti Paramita Parida. Load balancing in cloud computing: a big picture. *Journal of King Saud University-Computer and Information Sciences*, 32(2):149–158, 2020.

[53] Jiao Zhang, F Richard Yu, Shuo Wang, Tao Huang, Zengyi Liu, and Yunjie Liu. Load balancing in data center networks: A survey. *IEEE Communications Surveys & Tutorials*, 20(3):2324–2352, 2018.

[54] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. Offloading load balancers onto smartnics. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 56–62, 2021.

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

[56] Nisha 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).

[57] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[58] Elisha 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).

[59] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in neural information processing systems*, 31, 2018.

[60] Chao Ma, Stephan Wojtowytsch, Lei Wu, et al. Towards a mathematical understanding of neural network-based machine learning: what we know and what we don't. *arXiv preprint arXiv:2009.10713*, 2020.

[61] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[62] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning. In *Learning for dynamics and control*, pages 486–489. PMLR, 2020.

[63] Zhi-xiong Xu, Lei Cao, Xi-liang Chen, Chen-xi Li, Yong-liang Zhang, and Jun Lai. Deep reinforcement learning with sarsa and q-learning: A hybrid approach. *IEICE TRANSACTIONS on Information and Systems*, 101(9):2315–2322, 2018.

[64] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[65] Jiao Zhang, Yuxuan Gao, Shubo Wen, Tian Pan, and Tao Huang. Loom: Switch-based cloud load balancer with compressed states. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

[66] Kuo-Feng Hsu, Praveen Tammana, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Adaptive weighted traffic splitting in programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 103–109, 2020.

[67] Tom Barbette, Marco Chiesa, Gerald Q. Maguire, and Dejan Kostić. Stateless cpu-aware datacenter load-balancing. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, page 548–549, 2020.

[68] Benoît Pit-Claudel, Yoann Desmouceaux, Pierre Pfister, Mark Townsley, and Thomas Clausen. Stateless load-aware load balancing in p4. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 418–423. IEEE, 2018.

[69] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPC first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[70] Brian Chang, Aditya Akella, Loris D'Antoni, and Kausik Subramanian. Learned load balancing. In *24th International Conference on Distributed Computing and Networking*, pages 177–187, 2023.

[71] Jiao Zhang, Shubo Wen, Jinsheng Zhang, Hua Chai, Tian Pan, Tao Huang, Linquan Zhang, Yunjie Liu, and F Richard Yu. Fast switch-based load balancer considering application server states. *IEEE/ACM Transactions on Networking*, 28(3):1391–1404, 2020.

[72] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. Qcmp: Load balancing via in-network reinforcement learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, pages 35–40, 2023.

[73] Bruno Coelho and Alberto 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.

[74] Jiyoon Lim, Jae-Hyoung Yoo, and James Won-Ki Hong. Reinforcement learning based load balancing for data center networks. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 151–155. IEEE, 2021.

[75] Hongzi Mao, Malte Schwarzkopf, Hao He, and Mohammad Alizadeh. Towards safe online reinforcement learning in computer systems. In *NeurIPS Machine Learning for Systems Workshop*, 2019.

[76] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 740–755, 2021.

[77] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 756–771, 2021.

[78] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 513–533, Renton, WA, April 2022. USENIX Association.

[79] Diman Zad Tootaghaj, Anu Mercian, Vivek Adarsh, Mehrnaz Sharifian, and Puneet Sharma. SmartNICs at edge for transient compute elasticity. In *Proceedings of the 3rd International Workshop on Distributed Machine Learning*, pages 9–15, 2022.

[80] Chris McClanahan. History and evolution of gpu architecture. *A Survey Paper*, 9:1–7, 2010.

[81] Jaeyoung Do, Ivan Luiz Picoli, David Lomet, and Philippe Bonnet. Better database cost/performance via batched i/o on programmable ssd. *The VLDB Journal*, pages 403–424, 2021.

[82] Jack Zhao, Miguel Neves, and Israat Haque. On the (dis) advantages of programmable nics for network security services. In *2023 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2023.

[83] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[84] Ricardo Parizotto, Braulio Mello, Israat Haque, and Alberto Schaeffer-Filho. NetGVT: offloading global virtual time computation to programmable switches. In *Proceedings of the Symposium on SDN Research*, pages 16–24, 2022.

[85] Hisham Siddique, Miguel Neves, Carson Kuzniar, and Israat Haque. Towards network-accelerated ml-based distributed computer vision systems. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 122–129. IEEE, 2021.

[86] Carson Kuzniar, Miguel Neves, Vladimir Gurevich, and Israat Haque. IoT device fingerprinting on commodity switches. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2022.

[87] Carson Kuzniar, Miguel Neves, and Israat Haque. Poster: Accelerating encrypted data stores using programmable switches. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2020.

[88] Daehyeok Kim, Jacob Nelson, Dan RK Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 223–244, 2021.

[89] Ricardo Parizotto, Bruno Loureiro Coelho, Diego Cardoso Nunes, Israat Haque, and Alberto Schaeffer-Filho. Offloading machine learning to programmable data planes: A systematic survey. *ACM Computing Surveys*, 2023.

[90] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[91] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, pages i–189, 2018.

[92] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532, 2020.

[93] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. λ-nic: Interactive serverless compute on programmable smartnics. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 67–77, 2020.

[94] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[95] Broadcom. Braodcom. [n.d.]. PCI Express switches. https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches.

[96] Arrow. PCIE Switches — Arrow Electronics. https://www.arrow.com/en/categories/electronic-switches/pci/pci-express-switches.

[97] Mellanox. Mellanox. [n.d.].ConnectX®-5 EN Card. https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf.

[98] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252, 2003.

[99] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[100] Christian Hopps et al. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, Internet Engineering Task Force, 2000.

[101] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[102] Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-sheng Hua. Quantization networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7308–7316, 2019.

[103] Debobroto Das Robin and Javed I Khan. CLB: Coarse-grained precision traffic-aware weighted cost multipath load balancing on pisa. *IEEE Transactions on Network and Service Management*, 2022.

[104] Dingming Wu, Ang Chen, TS Eugene Ng, Guohui Wang, and Haiyong Wang. Accelerated service chaining on a single switch ASIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 141–149, 2019.

[105] Nour Moustafa and Jill Slay. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *2015 military communications and information systems conference (MilCIS)*, pages 1–6. IEEE, 2015.

[106] Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

[107] David Wadden, Shanchuan Lin, Kyle Lo, Lucy Lu Wang, Madeleine van Zuylen, Arman Cohan, and Hannaneh Hajishirzi. Fact or fiction: Verifying scientific claims. In *EMNLP*, 2020.

[108] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[109] Jeremy Elson, John R Douceur, Jon Howell, and Jared Saul. Asirra: a captcha that exploits interest-aligned manual image categorization. *CCS*, 7:366–374, 2007.

[110] Gongde Guo, Hui Wang, David Bell, Yaxin Bi, and Kieran Greer. KNN model-based approach in classification. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 986–996. Springer, 2003.

[111] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[112] Nginx. `https://nginx.org/`. Accessed: December 1, 2023.

[113] Tajbakhsh Hesam. P4Wise. `https://github.com/PINetDalhousie/P4Wise`, 2022.

Appendix A

Simulator Validation

This appendix offers a direct comparison of the outcomes derived from our actual test environment and those generated by our simulation. This comparative analysis aims to enhance the authenticity and trustworthiness of our simulation methodology, demonstrating its consistency with real-world performance attributes. In Figures A.1, A.2, A.3, and A.4, we can discern the throughput and E2E delay data from both our physical test setup and the simulator, revealing a striking resemblance with only minor disparities.

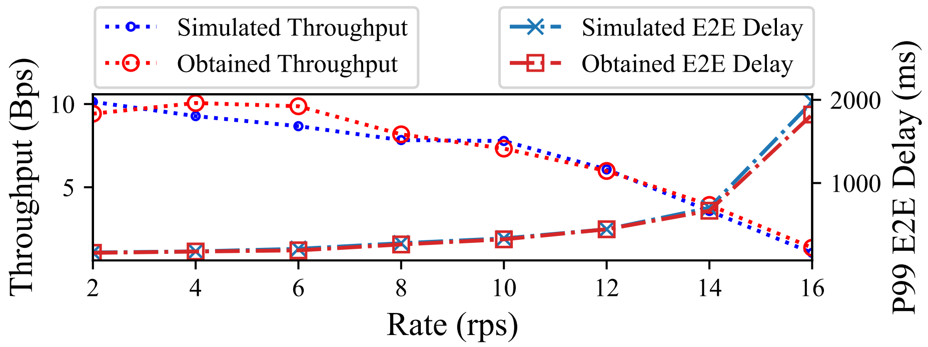


Figure A.1: Server-only's results on the testbed and simulator

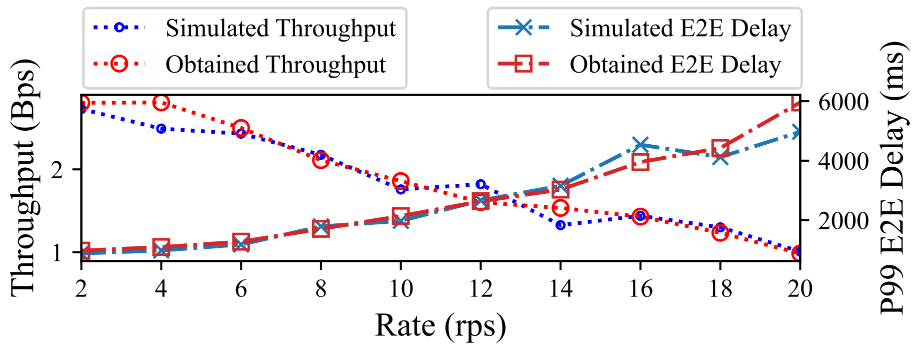


Figure A.2: LUR's results on the testbed and simulator

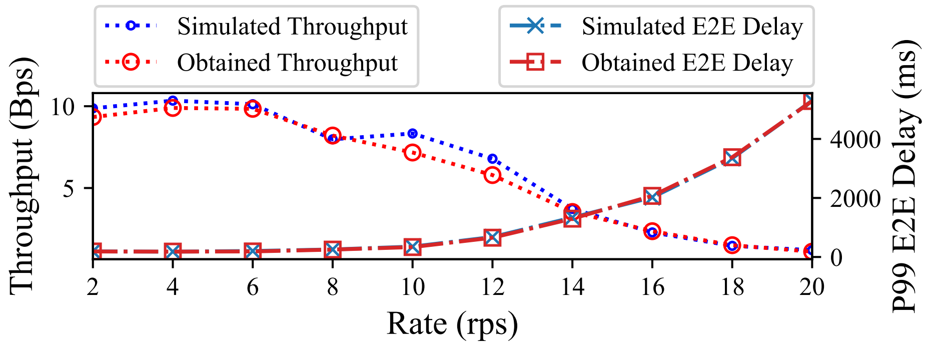


Figure A.3: PRT's results on the testbed and simulator

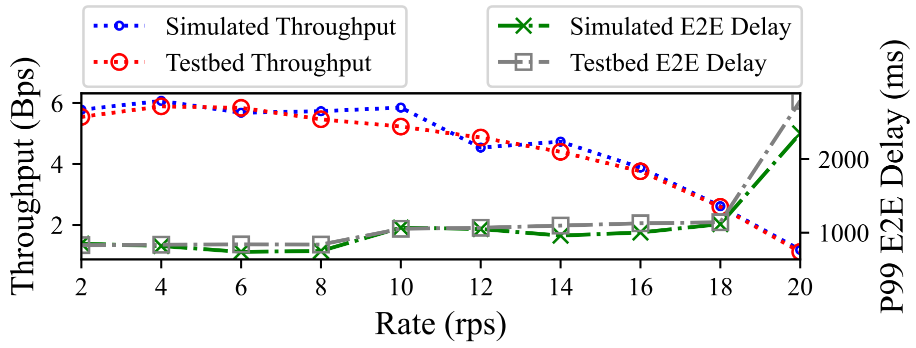


Figure A.4: WRR's results on the testbed and simulator

Appendix B

List of Publications from the Ph.D. thesis

1. Xing T, **Tajbakhsh H**, Haque I, Honda M, Barbalace A. Towards portable end-to-end network performance characterization of smartnics. InProceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems 2022 Aug 23 (pp. 46-52).

2. **Tajbakhsh H**, Parizotto R, Neves M, Schaeffer-Filho A, Haque I. Accelerator-aware in-network load balancing for improved application performance. In2022 IFIP Networking Conference (IFIP Networking) 2022 Jun 13 (pp. 1-9). IEEE.

3. **Tajbakhsh H**, Parizotto R, Schaeffer-Filho A, Haque I. P4Hauler: an Accelerator-aware In-network Load Balancer for Applications Performance Boosting. IEEE Transactions on Cloud Computing. 2023 Dec.