UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

BRUNO LOUREIRO COELHO

Efficient and Scalable Load Balancing Through Cross-Plane Cooperation

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Alberto Egon Schaeffer Filho

Porto Alegre
October 2023

CIP — CATALOGING-IN-PUBLICATION

Coelho, Bruno Loureiro

Efficient and Scalable Load Balancing Through Cross-Plane Cooperation / Bruno Loureiro Coelho. – Porto Alegre: PPGC da UFRGS, 2023.

88 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Alberto Egon Schaeffer Filho.

1. Load Balancing. 2. Traffic Engineering. 3. Deep Reinforcement Learning. 4. Machine Learning. 5. Programmable Data Planes. 6. Elephant Flow Detection. I. Schaeffer Filho, Alberto Egon. II. Título.

ABSTRACT

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

Keywords: Load Balancing. Traffic Engineering. Deep Reinforcement Learning. Machine Learning. Programmable Data Planes. Elephant Flow Detection.

Balanceamento de Carga Eficiente e Escalável Através da Cooperação Entre Planos

RESUMO

Balancear o tráfego de rede por meio de vários caminhos mais curtos tornou-se uma prática comum para obter uma utilização eficiente da infraestrutura de rede. Embora seja frequentemente utilizado, o Equal-Cost Multi-Path (ECMP) oferece um desempenho longe do ideal. Várias soluções de balanceamento de carga utilizam o Weighted-Cost Multi-Path (WCMP), dividindo o tráfego entre links proporcionalmente aos pesos dos links. No entanto, a implementação do WCMP exige que o controlador atualize as regras de encaminhamento match+action sempre que os pesos devem ser alterados, introduzindo um atraso antes que a divisão de tráfego apropriada possa ser aplicada. Além disso, o WCMP divide o tráfego de rede baseando-se apenas nos pesos dos links, sem levar em conta as características ou necessidades de cada fluxo. Neste trabalho, propomos CrossBal, um sistema híbrido de balanceamento de carga baseado em Deep Reinforcement Learning (DRL) que concentra seus esforços em fluxos elefantes de alto impacto. O agente de DRL é modelado para ser capaz de utilizar os links de rede de forma eficiente e, ao mesmo tempo, minimizar o espaço de ação, permitindo que o agente aprenda rapidamente como balancear a carga. Além disso, o CrossBal pode reagir rapidamente às mudanças na rede monitorando e alternando as rotas ativas diretamente no plano de dados. Nossa avaliação mostra que o CrossBal consegue utilizar os recursos da rede de forma eficiente, usando a maioria dos links disponíveis, ao mesmo tempo que reduz o desequilíbrio na utilização dos links. Também avaliamos a detecção de fluxo elefante utilizada pelo CrossBal, mostrando como ele pode identificar rapidamente fluxos elefantes através do uso eficiente dos recursos do *switch* programável.

Palavras-chave: Balanceamento de Carga. Engenharia de Tráfego. Aprendizado por Reforço Profundo. Aprendizado de Máquina. Planos de Dados Programáveis. Detecção de Fluxos Elefantes.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ML	Machine Learning
ECMP	Equal-Cost Multi-Path
WCMP	Weighted-Cost Multi-Path
DRL	Deep Reinforcement Learning
SDN	Software-Defined Networking
ASIC	Application-Specific Integrated Circuit
NIC	Network Interface Card
FPGA	Field Programmable Gate Array
CPU	Central Processing Unit
DPDK	Data Plane Development Kit
XDP	eXpress Data Path
DDoS	Distributed Denial of Service
RL	Reinforcement Learning
NN	Neural Network
DNN	Deep Neural Network
FC	Fully Connected
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
DQN	Deep Q-Learning
DM	Demand Matrix
RF	Random Forest
FIFO	First-in First-out
RR	Round-Robin
PoC	Proof-of-Concept

LoC	Lines-of-Code
VM	Virtual Machine
OS	Operating System
FCT	Flow Completion Time
MLU	Maximum Link Utilization
WAN	Wide Area Network

CONTENTS

1 INTRODUCTION

This work focuses on the efficient utilization of network resources through the judicious load balancing of high-impact flows. More specifically, we propose CrossBal, a load balancing system that combines programmable data planes with Machine Learning (ML) to reduce the imbalance in the utilization of network links. Section 1.1 provides further contextualization into load balancing in computer networks, while Section 1.2 presents the motivation of this work. Next, Section 1.3 lists the goals of this work, while Section 1.4 concludes this chapter with the outline of this document.

1.1 Contextualization

With the accelerated growth of the Internet in the past few decades, network infrastructures have been continuously expanded. Computer networks often implement redundant paths between end-points in order to increase reliability and meet the demands of users and services (HSU et al., 2020b). However, in order to efficiently utilize the available multi-path infrastructure, the incoming traffic must be properly split between the redundant output links, avoiding overwhelming links while others remain underutilized (NOORMOHAMMADPOUR; RAGHAVENDRA, 2018).

In this multi-path scenario, classic shortest-path routing strategies are unable to provide efficient network utilization (REDA et al., 2020). Equal-Cost Multi-Path (ECMP) is a widely deployed load balancing technique due to its simplicity, being readily available in commercial switches (GAVRILUŢ; PRUSKI; BERGER, 2022). However, ECMP may present severe performance issues, being unable to provide an efficient split of the network traffic (ZHANG et al., 2018). Weighted-Cost Multi-Path (WCMP) extends ECMP by adding weights to each hop, increasing performance and resilience to network asymmetry. Researchers have proposed a diverse set of techniques for calculating optimal weights for WCMP. These techniques can be based on heuristics (PERRY et al., 2023), Machine Learning models (XU et al., 2018; VALADARSKY et al., 2017), or other strategies (LE et al., 2021; MAGNOUCHE et al., 2021). However, updating link weights during congestion requires control plane intervention, which introduces considerable delay. On the other hand, load balancing solutions that rely entirely on the data plane are limited to specific topologies (e.g., CONGA (ALIZADEH et al., 2014), HULA (KATTA et al., 2016)) or employ simple heuristics (e.g., LetFlow (VANINI et al., 2017), BurstBalancer (LIU et

al., 2022b)).

In addition to the aforementioned deficiencies, the majority of proposed load balancing systems generally do not consider the characteristics or needs of each network flow. Elephant flows are high-throughput, long-lasting flows that tend to have a large impact on the network (LIU et al., 2022a). While elephant flows may constitute a small portion of total flows, research shows that a few large flows constitute more of the total network traffic than an immense number of small flows (DURNER; KELLERER, 2020). Considering the impact that elephant flows have on the network, intelligent rerouting of these flows can severely improve network utilization (JURKIEWICZ, 2021a). Additionally, as elephant flows are long-lived, we have more chances to reroute them.

Given the importance of elephant flows, a system capable of identifying and rerouting these flows is better equipped to load balance the network. However, the identification of elephant flows requires monitoring up to terabits per second of network traffic. While control plane solutions can enable complex techniques for identifying elephant flows, they are not able to process network traffic at these rates (JURKIEWICZ, 2021b). An alternative is to use the data plane of networking devices to aid in the identification of elephant flows. While emerging programmable switches (BOSSHART et al., 2014) allow us to reconfigure the packet processing pipeline, they are still subject to limitations, as these devices tend to have a few tens of MBs of memory, a restricted set of logical and arithmetic operations, limitations on memory accesses, and a strict time budget to process each packet (SAPIO et al., 2017).

In addition to the challenges in efficiently and accurately detecting elephant flows, load balancing active flows is not a trivial task. For instance, there may be a large number of possible routes for each active flow. Therefore, the strategy employed must be able to identify and select the best route, while also being able to scale up to larger networks. Further, load balancing systems must be able to react to changes in the network state, such as transient congestion. This requires the decision-loop to be fast enough to detect and handle congestion in a short time-scale. In conclusion, an ideal load balancing system has a large number of requirements, several of which seem opposed to each other.

1.2 Motivation

Despite research efforts, current techniques for network load balancing tend to either be too slow to react to network changes or use very simple heuristics. Generally,

approaches in the control plane employ more complex algorithms for splitting the incoming traffic (e.g., Machine Learning models (XU et al., 2018; VALADARSKY et al., 2017)). However, these approaches have a significant delay, as the controller has a slower decision-loop (KATTA et al., 2016). In contrast, in-network load balancing systems tend to use simple heuristics, often acting based on local or limited information, resulting in suboptimal decisions (ZHANG et al., 2017).

Considering the requirements listed previously, current load balancing schemes fall short from providing efficient resource utilization. In order to fully utilize the network infrastructure, we require an approach capable of providing both intelligent and reactive routing, while also taking into account system scalability and the characteristics of flows.

We propose CrossBal, a hybrid load balancing system that combines an intelligent control plane with a reactive data plane. CrossBal employs a Deep Reinforcement Learning (DRL) agent in the control plane, responsible for intelligently selecting routes that maximize the efficiency of the network. CrossBal focuses on optimizing the routing of high-impact flows, such as elephant flows. By prioritizing a small subset of high-impact flows, we can efficiently load balance the network without introducing scalability issues. Further, CrossBal offloads part of the task of identifying elephant flows to the data plane. Through the collaboration of the control and data planes, CrossBal achieves scalable, accurate, and fast detection of elephant flows. Finally, CrossBal exploits programmable data planes to quickly detect and react to congestion in active paths. By utilizing programmable switches to probe and switch between installed paths for each elephant flow, CrossBal complements the slower (but intelligent) control-loop of the DRL agent with a quicker and reactive control-loop in the data plane.

1.3 Goals

The goal of this work is to design, implement, and evaluate a system capable of intelligently and efficiently load balancing elephant flows over the available network infrastructure. Further, we aim to design, implement, and evaluate a Deep Reinforcement Learning (DRL) agent capable of aiding our load balancing system with real-time route selection for active flows.

In order to achieve these goals, we must address the following objectives:

- **Investigate the state-of-the-art**. Several network load balancing systems have

been proposed in the literature. However, current solutions are still unable to deliver optimal performance. By studying the literature in load balancers and ML-aided routing techniques, we can identify the strengths and weaknesses of each approach;

- **Design and implement CrossBal**. This work proposes CrossBal, a hybrid Machine Learning-aided load balancing system capable of identifying and rerouting elephant flows, as well as detecting and reacting to congestion in selected paths. We highlight the main challenges in designing a hybrid load balancing system, as well as a Deep Reinforcement Learning agent for actively rerouting flows;
- **Evaluate a Proof-of-Concept prototype**. In order to validate our design, we implement and evaluate a prototype of CrossBal. The evaluation is based on BMv2[1] switches in an emulated environment with a realistic network topology and workloads.

1.4 Document Outline

The remainder of this document is organized as follows. Chapter 2 provides background knowledge on *programmable networks* (Section 2.1), the technology employed in this work; *load balancing* in computer networks (Section 2.2), the main focus of this work; and *Deep Reinforcement Learning* (Section 2.3), the Machine Learning technique responsible for actively rerouting high-impact flows.

Chapter 3 presents the design of our proposed system, CrossBal. First, Section 3.1 shows an overview of how CrossBal approaches load balancing the network. Then, we detail key aspects of our design: the cross-plane *elephant flow detection* (Section 3.2), the *Deep Reinforcement Learning agent* (Section 3.3), and the data plane mechanism to *monitor and switch paths* (Section 3.4). Finally, Section 3.5 concludes the chapter by detailing the architecture of CrossBal.

Next, Chapter 4 reports our evaluation of a proof-of-concept prototype of CrossBal. We first detail the prototype that was evaluated (Section 4.1), then describe the methodology employed in our evaluation (Section 4.2). We present the results of our evaluation of the performance of CrossBal in Section 4.3, followed by an analysis of proposed optimizations for the detection of elephant flows in the data plane (Section 4.4).

Chapter 5 discusses the related work, presenting the state-of-the-art in network

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

load balancing. For each work, we highlight its main characteristics, as described in Section 5.1. First, Section 5.2 covers data plane load balancers, while Section 5.3 covers approaches in the control plane. Section 5.4 concludes the chapter by covering emerging classes of load balancers: *hybrid* and *end-host* load balancers.

Finally, Chapter 6 concludes this work with our final remarks. Section 6.1 presents a summary of our contributions, while Section 6.2 discusses interesting directions for future work.

2 BACKGROUND AND MOTIVATION

This chapter provides necessary background information on several topics relevant to this work. First, Section 2.1 provides background on programmable networks, the emerging technology we utilized in this work. Next, Section 2.2 discusses load balancing in computer networks, the main focus of our work. Finally, Section 2.3 presents the main concepts of Deep Reinforcement Learning, the Machine Learning model employed in this work to select routes for active flows.

2.1 Programmable Networks

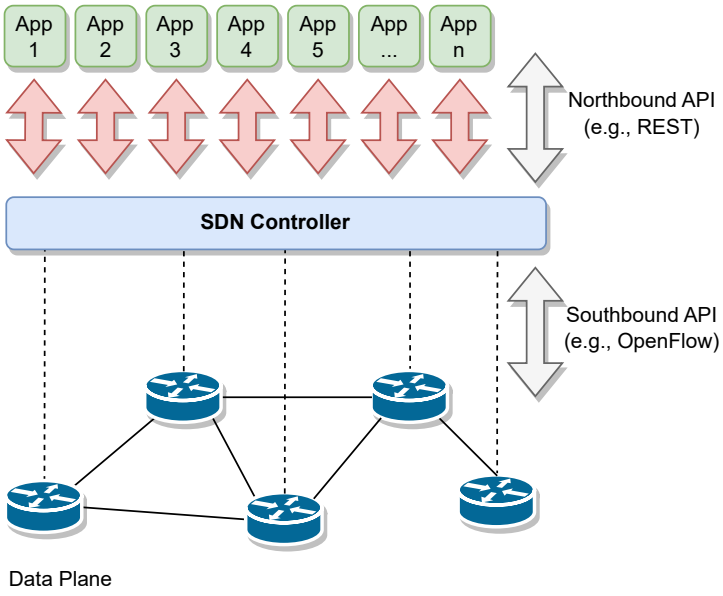
Enabling programmability in computer networks is an effort that dates back to the 1990s. Since then, the recurring motivations for making networks programmable have been to facilitate network management and innovation (FEAMSTER; REXFORD; ZEGURA, 2014). Despite several early attempts, it was not until recently that network operators started adopting programmability. Starting from the late 2000s, both the academia and the industry began using Software-Defined Networking (SDN) to create more manageable and efficient networks (ZHU et al., 2020). Examples of adoption by the industry include Google's private WAN, B4 (JAIN et al., 2013), that has been in use since 2010 (HONG et al., 2018); and Microsoft's decentralized WAN traffic engineering system, BlastShield (KRISHNASWAMY et al., 2022).

2.1.1 Software-Defined Networking

Software-Defined Networking (SDN) is a paradigm that divides functionality between a logically-centralized Control Plane and a distributed Data Plane, as shown in Figure 2.1. By decoupling functionality, SDN allows a centralized controller to have a global view of the network. At the same time, the network data plane can be simplified to basic forwarding devices (ZHU et al., 2020).

In SDN, the logically-centralized controller is responsible for the configuration of the forwarding devices in the data plane (ZHU et al., 2020). This configuration is done through the Southbound API (as shown in Figure 2.1), with protocols such as OpenFlow (MCKEOWN et al., 2008). Using OpenFlow as an example, the controller config-

Figure 2.1 – Plane decoupling in Software-Defined Networks.



Source: Adapted from Kreutz et al. (2015).

ures the OpenFlow-enabled switches with *flow rules*, which tell the device how to process incoming packets.

Table 2.1 shows a simplified example of a flow table, with each entry specifying a list of values for the match fields, a list of (zero or more) counters to update, and a (non-empty) list of instructions to be applied over the packet. For instance, the first line in the table dictates that packets arriving from port 2 of the forwarding device, with a destination IP address matching the prefix $10.0.1.0/24$ (e.g., $10.0.1.1$ or $10.0.1.50$), should be processed according to the instructions in this entry. This includes decrementing the time-to-live field of the packet, and then forwarding it through port 3 of the switch. Additionally, the counter for bytes sent through that output port should be updated with the size of the packet forwarded.

Another example shown in Table 2.1 can be understood when looking at both the second and third entries in the flow table: when a packet arrives at port 1, if its destination TCP port is 4321, it should be forwarded through port 2 (entry #2). However, if the destination TCP port does not match the specified value, the packet should be dropped[1] (entry #3). Finally, entry #4 is a simplified example of a typical default rule. Default rules are applied when a packet does not match any of the other entries. In this example, we want every packet that does not match any of the other entries to be forwarded to the

[1] In practice, flow tables also require each entry to have a priority specified. The priority field is used to select an entry when a packet matches more than one flow entry. This field was omitted in the example to simplify the table.

Table 2.1 – Simplified example of a Flow Table

Match Fields			**Counters**	**Instructions**
Ingress Port	Destination IP Address	Destination TCP Port		
Port 2	10.0.1.0/24	Any	Bytes Sent	decrement TTL output port_3
Port 1	0.0.0.0/0	4321	Bytes Sent Packets Sent	decrement TTL output port_2
Port 1	0.0.0.0/0	Any	Packets Dropped	drop
Any	0.0.0.0/0	Any	-	Forward to Controller

Source: The Authors.

controller (a “packet_in” to the controller).

One of the consequences of having a logically centralized controller is that all the information about the network is gathered in one (logical) place. As shown in Figure 2.1, this allows network applications to communicate with the controller through a Northbound API (e.g., REST). This way, a developer can create a specific application, such as an innovative routing algorithm, that receives topology information from the controller. After computing the new routes, this application can forward it to the controller, which translates these routes into entries of a flow table for OpenFlow-enabled switches to apply.

SDN controllers can combine the functionality of the Southbound API and the Northbound API, allowing it to implement complex logic over the network. As a simple example, a controller can receive the first packet of each flow (a ”packet_in” in OpenFlow) through the Southbound API, then query a routing application (e.g., based on Machine Learning) through the Northbound API. Once the application returns an optimal routing strategy for this flow, the controller can update the flow tables of the switches through the Southbound API, e.g., by sending a “flow_mod” message with the OpenFlow protocol.

2.1.2 Programmable Data Planes

While Software-Defined Networking allows network operators to have more control over their networks, SDN protocols (e.g., OpenFlow) still have to be consistently updated to include new header fields and instructions (BOSSHART et al., 2014). For instance, the first version of the protocol, OpenFlow 1.0, released in December of 2009,

supported 12 header fields for matching in flow table entries (OPEN NETWORKING FOUNDATION, 2009). By version 1.4, released in October 2013, OpenFlow had been extended to support 41 header fields (BOSSHART et al., 2014). Additionally, newer versions of the protocol also extended the list of supported actions (OPEN NETWORKING FOUNDATION, 2015).

The need to constantly update the protocol to support new headers and actions can slow down the pace of innovation (BOSSHART et al., 2014). Seeking to empower network operators with more control over network protocols and how data packets are processed, researchers aimed to add programmability to the *data plane*. Programmable data plane devices include programmable Application-Specific Integrated Circuits (ASICs) (e.g., Intel Tofino), smart Network Interface Cards (NICs), and Field Programmable Gate Arrays (FPGAs). Additionally, general-purpose Central Processing Units (CPUs) (LIATIFIS et al., 2023) empowered with technologies such as Data Plane Development Kit (DPDK) (FOUNDATION, 2015) or eBPF eXpress Data Path (XDP) (HØILAND-JØRGENSEN et al., 2018) can be used to process packets efficiently.

In this work, we focus on programmable switches compatible with the P4 language (BOSSHART et al., 2014). Implementations of P4-enabled switches include software switches, such as BMv2[2], and programmable ASICs, such as the Intel Tofino[3]. While the exact architecture of a programmable switch depends on the target (e.g., BMv2 or Tofino), P4-enabled switches are based on an Abstract Forwarding Model (BOSSHART et al., 2014), shown in Figure 2.2.

Switches following the Abstract Forwarding Model (BOSSHART et al., 2014) are composed of a parser, an ingress, and an egress processing blocks (LIATIFIS et al., 2023). Several concrete architectures (e.g., v1model[4], PSA[5]) also define a deparser block (not present in Figure 2.2). The deparser block specifies the headers to be added to the packet that is going to be emitted in the output port, as well as the relative order of the headers.

- **Parser**: This block is the first step in the pipeline, being responsible for parsing protocol headers. As opposed to SDN protocols like OpenFlow, P4 does not define any protocol headers by default. Instead, network operators have complete control over protocol headers, being responsible for defining the **size** and **order** of each

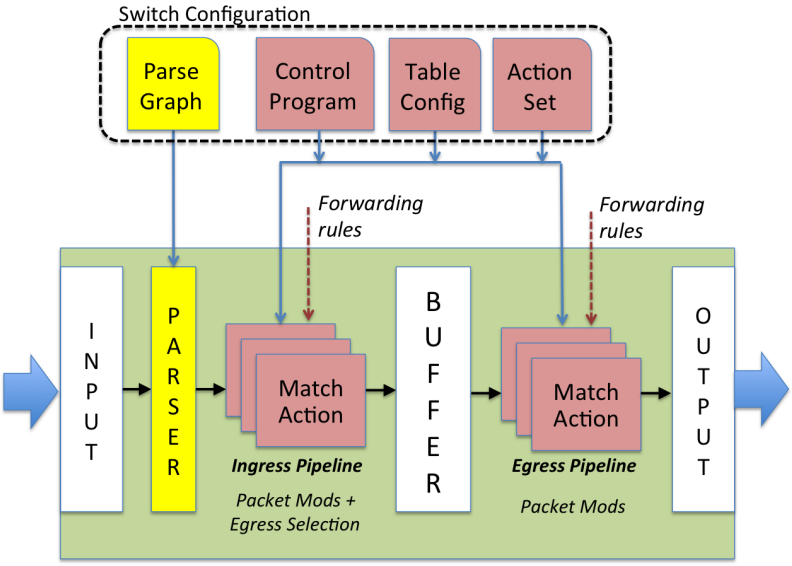
[2]BMv2: <https://github.com/p4lang/behavioral-model>

[3]Intel Tofino: <https://www.intel.com/content/www/us/en/products/details/network-io/programmable-ethernet-switch/tofino-series.html>

[4]v1model: <https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4>

[5]PSA: <https://github.com/p4lang/p4c/blob/main/p4include/bmv2/psa.p4>

Figure 2.2 – Abstract Forwarding Model



Source: Bosshart et al. (2014).

protocol field. This eliminates the issue cited above, where SDN protocols have to be constantly updated. Additionally, it empowers network operators with the ability to design, test, and deploy *custom protocols*. The parser is also responsible for deciding if packets should be accepted for further processing or if they should be dropped. If a packet is accepted by the parser, the P4-enabled switch maintains the value of parsed header fields to be used by subsequent processing blocks;

- **Ingress**: The ingress processing block is responsible for processing packets that have been accepted by the parser. In this block, the network operator can define *match+action* tables, matching on arbitrary keys, such as packet header fields or custom metadata, and invoking actions defined by the operator. Match+action tables are similar to Flow Tables (e.g., Table 2.1), except the keys (match fields) and the actions (instructions) are defined by the network operator. Actions can be created based on simple **arithmetic and logic primitives**, as well as architecture-specific functions. This includes reading and storing data in **registers**, allowing stateful processing. Along with user-defined metadata, each architecture defines a set of intrinsic metadata. For instance, the developer can read timestamp when the packet was enqueued, or when it started being processed by the ingress block;
- **Egress**: The egress processing block is identical to the ingress processing block, except the output port has already been selected. The egress processing block is par-

ticularly useful in scenarios involving *packet cloning*. Several P4-enabled switches support functionality such as packet cloning, packet resubmission, and packet recirculation. It is also worth noting that both blocks have an independent amount of resources, such as SRAM and arithmetic and logic units. Further, each block (ingress and egress) has its own independent set of registers and match+action tables (as defined by the network operator). While the resources of the ingress and egress blocks are independent, they are also **scarce**. State-of-the-art programmable switches have a few tens of MBs of SRAM and a limited amount of pipeline stages (SAPIO et al., 2017).

Similarly to switches in a Software-Defined Networking scenario, P4-enabled switches also communicate with a logically centralized controller. However, this is subject to the limitations of the communication channel and the controller's processing rate. The controller is responsible for configuring each switch with the source code and other configuration files at startup, as shown in Figure 2.2. Additionally, the controller can interact with each switch during runtime, inserting and removing entries from match+action tables; accessing the values of statistics counters and stateful registers; configuring packet cloning and recirculation; and so on. The complete set of operations supported by P4-enabled programmable switches is defined in the P4Runtime Specification[6].

Data plane programmability is being extensively researched in recent years. By exploiting the capabilities of P4-enabled switches, researchers have explored offloading part of applications into the network (LIATIFIS et al., 2023). Aside from enabling new applications, programmable data planes also allow network operators to design, evaluate, and deploy new solutions to old problems.

For instance, in a previous work, we implemented both the computation of features and the evaluation of a Machine Learning (ML) model in the data plane to detect Distributed Denial of Service (DDoS) attacks (COELHO; SCHAEFFER-FILHO, 2022); HULA (KATTA et al., 2016) implements a load balancing system for datacenters entirely in the data plane; and HashPipe (SILVA et al., 2018) uses P4-enabled switches to detect heavy hitters - flows that have large traffic volumes. In conclusion, programmable data planes allow researchers and network operators to propose and evaluate innovative protocols and algorithms. In this work, we use P4-enabled switches to compute features for an ML model and to aid the controller in load balancing the network.

[6]P4Runtime Specification: <https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html>

2.2 Load Balancing in Computer Networks

The expansion of computer networks, along with the increasing complexity and requirements of applications, has made clear the need for efficient utilization of network resources. Classic shortest-path routing strategies are not able to meet these requirements (REDA et al., 2020). Instead, traffic engineering systems started focusing on *load balancing* the network.

2.2.1 Equal-Cost Multi-Path (ECMP)

Equal-Cost Multi-Path (ECMP) evenly splits traffic between multiple equal-cost paths. This is commonly achieved by applying a hash function H over the five-tuple that defines a flow: source IP address, destination IP address, source port, destination port, and protocol. As H is deterministic, applying it over the five-tuple always produces the same result, which can be used to pick among equal-cost paths (ZHANG et al., 2018).

As the implementation of ECMP in forwarding devices is very simple, it is readily available in commercial switches (GAVRILUŢ; PRUSKI; BERGER, 2022). However, ECMP suffers from severe performance drawbacks, often causing high link utilization. One of the reasons why ECMP may lead to link congestion is due to *hash collisions* between large flows (ZHANG et al., 2018). Another drawback of ECMP is that it lacks any knowledge about the current state of the links, that is, it is not aware of any potential congestion along paths (ALIZADEH et al., 2014).

2.2.2 Weighted-Cost Multi-Path (WCMP)

Weighted-Cost Multi-Path (WCMP) improves upon ECMP by adding weights to each hop, increasing performance and resilience to network asymmetry. By carefully setting appropriate weights for each link, WCMP can lead to efficient network utilization even during link failures. These weights can be calculated through heuristics (PERRY et al., 2023; LE et al., 2021; MAGNOUCHE et al., 2021) or Machine Learning models (XU et al., 2018; LI et al., 2020; VALADARSKY et al., 2017).

While WCMP can provide more efficient network utilization than ECMP, updating link weights during congestion requires control plane intervention. This controller in-

volvement introduces considerable delay (ZHANG et al., 2020), which limits the switch's ability to react to transient congestion (ALIZADEH et al., 2014).

Further, WCMP is not as straight-forward as ECMP to implement in commodity switches. While ECMP requires one entry in a hash table or match-action table for each equal-cost hop, standard implementations of WCMP require replicating entries proportionally to the weight of each path (HSU et al., 2020b). This can lead to severe waste of switch resources depending on path weights.

2.2.3 Elephant Flows

Computer applications differ in the way they use the network - some only transmit a small portion of data, while others may require a large amount of bandwidth. Aside from the amount of data transferred, the duration of network flows can also differ based on the application. Finally, the requirements for each flow also depend on the application - some require low latency, while others are mostly interested in high throughput.

While flow classification can be quite complex, two broad types of flows are mice and elephant flows. *Mice flows* are characterized by low-throughput and short duration (SILVA et al., 2018). Due to these characteristics, the performance of these flows is more sensitive to additional latency and packet loss (ALIZADEH et al., 2014). By contrast, *elephant flows* are characterized by high-throughput and long duration (LIATIFIS et al., 2023).

Out of the two, mice flows represent the majority of network flows. Despite this, it is actually a small number of *elephant flows* that carry most of the network traffic (GUO; MATTA, 2001; ALIZADEH et al., 2010; GREENBERG et al., 2009). Due to their characteristics, elephant flows tend to have a large impact on the network (CURTIS et al., 2011). Consequently, optimizing the path taken by these flows can severely improve network utilization (AL-FARES et al., 2010).

Considering the impact of elephant flows, identifying these flows may allow rerouting them in order to avoid network congestion. Elephant flows can be identified through *detection* or *prediction*. Detection-based approaches employ simple thresholds or heuristics for detecting elephant flows (SILVA et al., 2018). However, this typically requires flows to be tracked long enough to be detected, incurring delay. An alternative is the prediction of elephant flows, using heuristics (SILVA; SCHAEFFER-FILHO; GRANVILLE, 2022) or Machine Learning models (DURNER; KELLERER, 2020).

Despite its flexibility, the identification of elephant flows in the control plane is not able to scale up to current network throughput rates, which can reach terabits per second (JURKIEWICZ, 2021b). While sampling can improve scalability, it delays the identification of elephant flows (SILVA et al., 2018). An alternative is to implement the identification of elephant flows in the data plane. However, this poses several challenges due to the limitations of programmable data plane devices (previously explained in Section 2.1). Therefore, there is a trade-off between the scalability provided by simple approaches that can be implemented in the data plane (e.g., IDEAFIX (SILVA et al., 2018), HashCuckoo (SILVA; SCHAEFFER-FILHO; GRANVILLE, 2022)) and the high accuracy of control plane approaches, where it is possible to implement more complex identification techniques, such as Machine Learning models (DURNER; KELLERER, 2020).

2.3 Deep Reinforcement Learning

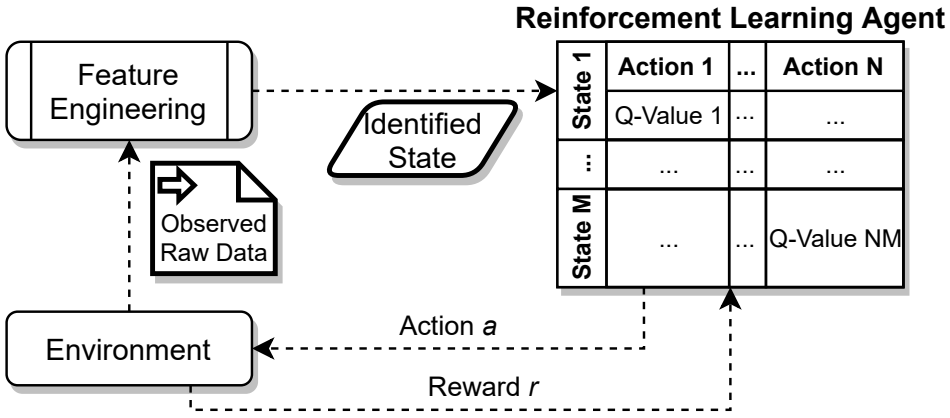
Machine Learning models can be divided into supervised models, unsupervised models, semi-supervised models, and reinforcement learning (PARIZOTTO et al., 2023). While supervised models learn from a labeled dataset, reinforcement learning algorithms learn by interacting with an environment (SUTTON; BARTO, 2018).

2.3.1 Reinforcement Learning

In Reinforcement Learning (RL), an agent interacts with its *environment* in each timestep $t \in 1, 2, ...$ In each iteration, the agent observes a state $s \in S$ and chooses an action $a \in A$ according to its policy π (ZHAN; ZHANG, 2020). Afterwards, the agent receives a reward r according to a *reward function* and transitions to a new state $s' \in S$ according to a *transition function*. RL algorithms can learn an optimal policy π even without any explicit knowledge of the reward or transition functions (RESTUCCIA; MELODIA, 2020).

Q-Learning is one of the most popular RL algorithms due to its simplicity. In Q-Learning (Figure 2.3), the agent attempts to learn to estimate the value of every state-action pair, (s, a). To this end, the agent keeps an entry for each state-action pair in its Q-table, $Q(s, a)$. At the end of each iteration, the agent updates the value of the entry

Figure 2.3 – Main aspects of Reinforcement Learning



Source: The Authors.

$Q(s, a)$ with Equation 2.1, where s is the observed state, a is the action to be taken in this state, and r is the reward expected from taking the action a in state s. Additionally, s' is the state the agent will transition to in the next step and $a' \in A$ is each possible action the agent can take. Lastly, we have γ, the discount factor for future rewards, and α, the learning rate. As this equation involves a future state, s', the update of Q-values are done in the next time-step, where the agent can easily observe s' (SUTTON; BARTO, 2018).

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha(r + \gamma \max_{a'} Q_k(s', a') \tag{2.1}$$

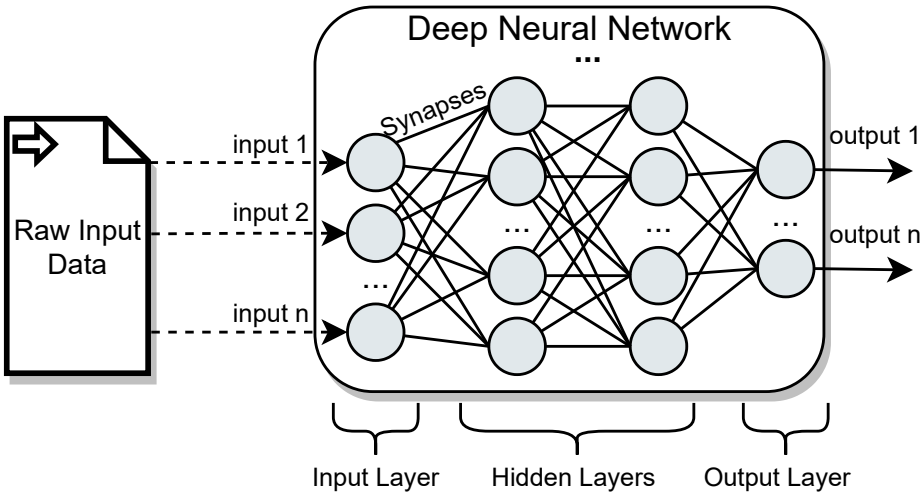
As shown in Figure 2.3, Q-Learning requires transforming the observed raw data into the observed (identified) state. This process is known as *feature engineering*, as the raw data is used to compute "features" that represent the observable state. Due to this requirement, Q-Learning is ill-suited for complex environments, as explicitly representing the entire state space can lead to an effect known as "*state-space explosion*" (RESTUCCIA; MELODIA, 2020). For instance, efficiently representing the state of computer networks is extremely challenging with traditional RL algorithms. In Q-Learning, representing the utilization of each link in the network would require transforming continuous values into discrete values. This process of transforming a continuous space into a discrete one is known as quantization. After quantization, the Q-table would require l^q entries for each possible action, where l is the number of links in the network and q is the number of values used for representing the link utilization. Therefore, in order to store an entry for each possible value, it is necessary to quantize the link utilization u_l into q possible values. These issues with explicitly representing (and storing) each possible observable state are not limited to applying RL in computer networks, also happening in several other

domains (RESTUCCIA; MELODIA, 2020).

2.3.2 Neural Networks

A Neural Network (NN) is a supervised Machine Learning model. A representation of the structure of a (Deep) Neural Network is shown in Figure 2.4. An NN is composed of an input layer, a set of hidden layers, and an output layer. Each layer is composed of a series of artificial neurons (LUONG et al., 2019). A Deep Neural Network (DNN) is a Neural Network with multiple hidden layers.

Figure 2.4 – Overview of Deep Neural Networks.



Source: The Authors.

During inference, an NN has input data fed into neurons in its input layer. The neurons in this layer are connected to neurons in the (first) hidden layer by *synapses*. Each synapse connecting neuron i to neuron j can be represented by a weight $W_{i,j}$ and a bias $B_{i,j}$. While the value of a neuron in the input layer is determined by the input data, the value of a neuron j in the next layer (first hidden layer) is computed by applying an activation function, $G(y_j)$, over the weighted sum of the values of every neuron i in the previous layer (input layer) that connects to neuron j (in the first hidden layer) (LUONG et al., 2019). More formally, if $G(y_j)$ is the activation function, the value of neuron j, x_j, can be defined with Equation 2.2.

$$x_j = G(\sum_i x_i * W_{i,j} + B_{i,j}) \tag{2.2}$$

The computation in Equation 2.2 is applied over every neuron j in layer l. As

this computation is applied over each neuron j in a given layer, subsequent layers can only be computed after the values of the neurons in the previous layers are known. In other words, for each layer $l \in \{1, 2, ..., |L|\}$, we must compute the values in layer $l - 1$ before computing l. Neural Networks similar to the one shown in Figure 2.4 are known as Feedforward Neural Networks, as the synapses are always directed from layer $l - 1$ to layer l (TALBI, 2021). Further, Neural Networks where every neuron i in layer l is connected to every neuron j in layer $l + 1$ are known as Fully Connected (FC) Neural Networks. It can also be said that layer l is fully connected to layer $l + 1$ (LUONG et al., 2019).

There is a wide range of activation functions $G(y_j)$ that can be applied to compute the final value x_j of neuron j. A popular example is the Rectified Linear Unit (ReLU) activation function, as shown in Equation 2.3 (GLOROT; BORDES; BENGIO, 2010).

$$G(y_j) = \begin{cases} y_j, & y_j \geq 0 \\ 0, & y_j < 0 \end{cases} \tag{2.3}$$

The training of a neural network consists of updating the weight (and bias) of each synapse. While there are multiple ways of updating these weights, a very common technique is backwards propagation combined with Stochastic Gradient Descent (SGD). At a very high level view of the technique, the combination of these techniques aims to calculate the error of a prediction during training and adjust the synapses based on that error. The values of the synapses are adjusted starting with the output layer, going back to the hidden layer(s). As we rely on the calculated error of the prediction during training, this means we must know the correct prediction for each training sample. In other words, neural networks are a *supervised* Machine Learning model (TALBI, 2021).

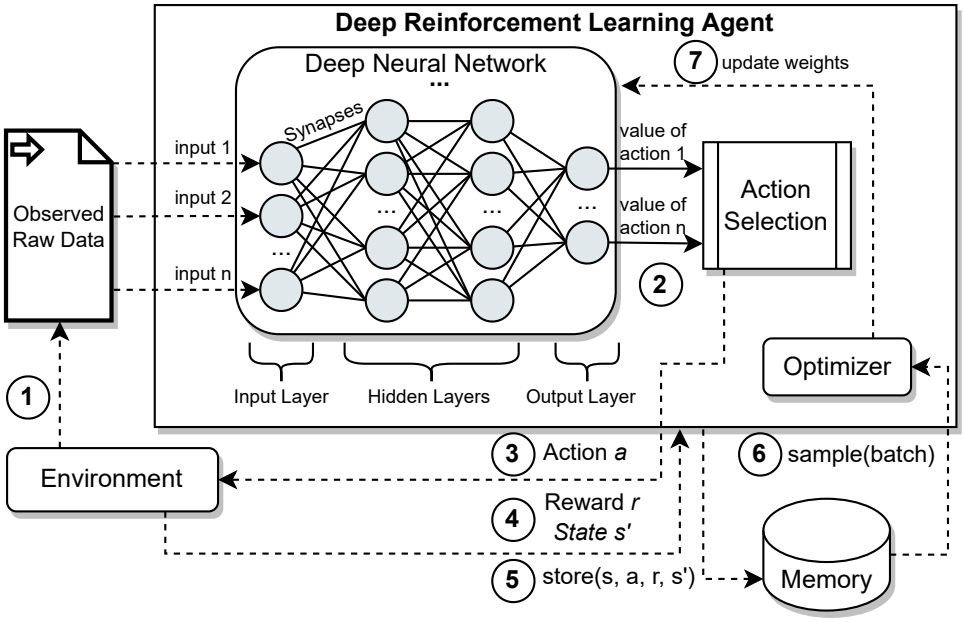
2.3.3 Deep Q-Learning

In order to be able to efficiently utilize Reinforcement Learning algorithms in systems with complex environments, researchers have proposed replacing the state-action pair with a function approximation. In particular, the use of *Deep Neural Networks* as a function approximator for RL (ELIYAHU et al., 2021) has been extensively researched in the past years. As this technique combines Deep Neural Networks with Reinforcement Learning, it is known as Deep Reinforcement Learning (DRL). While Deep Reinforce-

ment Learning (DRL) includes a broad set of algorithms, we focus specifically on Deep Q-Learning (DQN). Therefore, for the remainder of this document, we use the terms Deep Reinforcement Learning (DRL) and Deep Q-Learning (DQN) interchangeably, unless stated otherwise.

Figure 2.5 illustrates the main aspects of a Deep Reinforcement Learning (DRL) agent. First, (1) the DRL agent *observes raw data* which describes its current state s. As opposed to traditional Reinforcement Learning, the agent utilizes a Deep Neural Network (DNN) to process the information describing the state. This alleviates the need to discretize and store each possible state, as the DNN is capable of identifying similar states. Then, (2) the DNN outputs the expected *value of each action* for the observed state as the value of each neuron in the output layer. Next, (3) the agent *selects an action* a based on its strategy. This typically involves choosing between *exploiting*, i.e., choosing the action with the highest expected value, or *exploring* a random action, based on its exploration parameters. After acting, (4) the agent receives a *reward* r and *transitions* to a *new state* s'. State-of-the-art Deep Reinforcement Learning techniques usually (5) *store transitions* as a tuple (s, a, r, s') to be used for periodic training (ELIYAHU et al., 2021). This technique is known as memory replay, which has been shown to greatly stabilize learning (MNIH et al., 2015). With memory replay, (6) the agent periodically *samples a batch of transitions* stored in its memory. Finally, (7) the agent uses this batch to update its *internal weights* based on the rewards received. In order to stabilize the learning of the agent, the internal weights of the neural network being trained remain unchanged between "episodes". For a deeper understanding of Deep Reinforcement Learning techniques, we refer the reader to Mnih et al. (2015) and Silver et al. (2016).

Figure 2.5 – Steps of an iteration of a Deep Learning Agent



Source: The Authors.

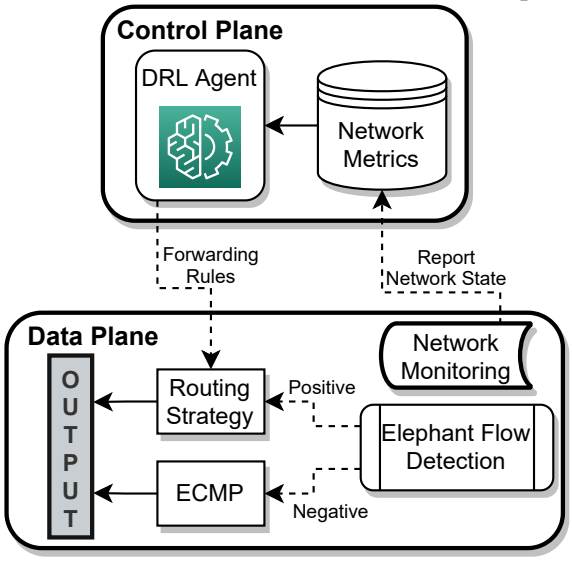
3 CROSSBAL: CROSS-PLANE LOAD BALANCING

CrossBal is a hybrid load balancing system that combines an intelligent control plane with reactive data plane processing. In this chapter, we present CrossBal, starting with an overview of the approach (Section 3.1), while subsequent sections present key design elements. Section 3.2 describes our collaborative mechanism for detecting elephant flows. Next, Section 3.3 discusses the main challenges and design choices in employing a Deep Reinforcement Learning agent for load balancing network flows. Then, Section 3.4 elaborates on how CrossBal utilizes programmable data planes to be able to react to short-lived network congestion. Finally, Section 3.5 concludes this chapter with architectural details of CrossBal.

3.1 Approach Overview

Figure 3.1 presents an overview of our approach. CrossBal relies on two *key principles* to balance network utilization in an efficient and scalable manner: (i) *cross-plane cooperation*, enabling CrossBal to quickly react to transient congestion by relying on mechanisms implemented in the programmable data plane, without having to sacrifice decision-making intelligence at the control plane; and (ii) *scalable traffic rerouting*, by focusing its efforts on flows that have the highest impact on the network (e.g., elephant flows and heavy hitters), as opposed to dealing with every flow in an equal manner.

Figure 3.1 – Overview of the collaboration between control and data plane employed in CrossBal.



Source: The Authors.

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

CrossBal decomposes load balancing into two complementary control-loops that work together to provide both intelligent and reactive decision-making. There is a *slower, but more intelligent, control-loop* at the control plane, which is fed with network monitoring data and employs the DRL agent to compute the optimal routes. However, programmable data plane devices also apply a *faster, but simpler, control-loop* to probe, monitor, and rapidly switch between a subset of active routes selected by the DRL agent. By having the data plane cooperate with the control plane in multiple aspects, CrossBal can achieve both intelligent and reactive load balancing of the network.

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

3.2 Identifying Elephant Flows Efficiently and Accurately

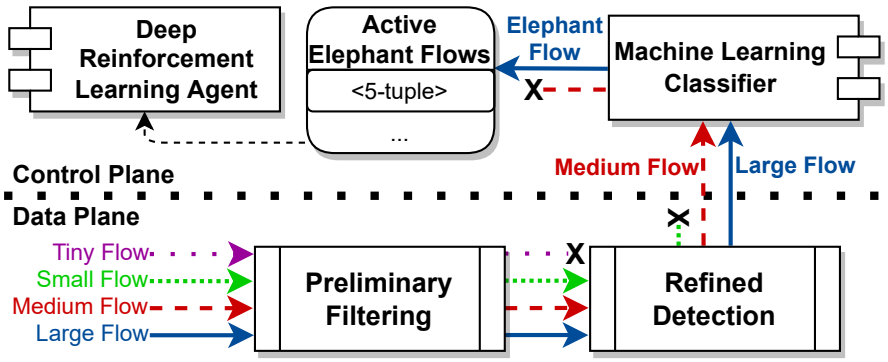
Identifying elephant flows efficiently and accurately is challenging in high-throughput networks, where network traffic rates can reach terabits per second (JURKIEWICZ, 2021b). Despite its flexibility, a centralized controller is incapable of performing per-packet classification without incurring latency overheads (SIVARAMAN et al., 2017).

A natural solution is to utilize programmable data plane devices, such as P4-enabled switches (BOSSHART et al., 2014), to offload part of this task. However, implementing an accurate and efficient identification of elephant flows in the data plane is not straight-forward due to the limitations of these devices (as previously discussed in Section 2.1). In particular, programmable switches impose a strict time budget to process each packet, limiting memory accesses and per-packet processing (SAPIO et al., 2017). Further, the data structures employed in the identification of elephant flows must be compact, as these devices tend to have only a few tens of MBs of memory (HSU et al., 2020a).

With the limitations of programmable data plane hardware in mind, CrossBal *de-*

composes the detection of elephant flows into three levels of complementary mechanisms, balancing the tradeoffs between fast and lightweight detection in the data plane with more accurate and heavyweight detection in the control plane. As illustrated in Figure 3.2, these mechanisms include a `Preliminary Filtering` that is applied over all flows, followed by a step of `Refined Detection` over a subset of flows in the data plane. Finally, the control plane implements a `Machine Learning Classifier` to provide a final, more accurate classification. We detail each of these mechanisms next.

Figure 3.2 – Overview of the cross-plane elephant flow detection.



Source: The Authors.

- ***Preliminary filtering:*** the data plane implements a threshold-based[1] detection that tracks the number of bytes, number of packets, and duration for each active flowlet[2]. The main aspect of this mechanism is that it must handle a large number of flows, consequently limiting the amount of processing and storage available for each flow. Thus, as shown in Figure 3.2, this step acts as an early filtering of low-throughput and short flows in order to save hardware resources. A further optimization is proposed and evaluated (in Section 4.4), where only packets larger than a threshold are accounted for. By utilizing this strategy, the number of packets of a flowlet can be seen as a lower bound of the number of bytes transmitted by the flowlet. This can lower the number of bits utilized to store this information, saving precious on-board memory. The same reasoning can be applied to track the number of flowlet timeouts of a flow rather than the entire duration.
- ***Refined detection:*** While the threshold-based mechanism mentioned above can exclude a large number of small and short flows, it may lead to a high number of false positives if used by itself. To address this, CrossBal employs further mechanisms to

[1]The configuration of thresholds should be done by network administrators based on knowledge of their networks. These thresholds should consider the characteristics of the network load and the hardware of data plane devices.

[2]Flowlets are bursts of packets from the same flow separate by an idle interval.

detect elephant flows. The intuition is that because the preliminary threshold-based filtering already excluded a large number of unimportant flows, it is now possible to implement a slightly *refined detection* mechanism over the remaining flows of interest, as shown in Figure 3.2. In particular, CrossBal implements a Classification Tree[3] in the form of *if-else* statements in the programmable data plane. As there is a smaller number of flows to consider, it is possible to dedicate slightly more onboard memory to track *features* of each flowlet. In our proof-of-concept prototype, we track simple statistics of the inter-arrival-time and packet size of each flowlet. We leave a more thorough feature selection as future work.

- ***Cross-plane detection:*** Although CrossBal implements multiple mechanisms for the identification of elephant flows directly in the data plane, ensuring line-rate processing requires sacrificing accuracy for efficiency. In order to provide a more accurate detection of elephant flows, CrossBal employs cross-plane collaboration, as shown in Figure 3.2. This builds upon the *preliminary filtering* (which reduces the amount of flows of interest) and upon the *refined detection* (which tracks additional features for relevant flows). Therefore, CrossBal implements a classifier in the control plane that receives the information tracked by the data plane. As the controller provides a more flexible programming model, and considering the features extracted by the data plane, we implement a Random Forest[4] in the control plane, providing higher accuracy than a single classification tree.

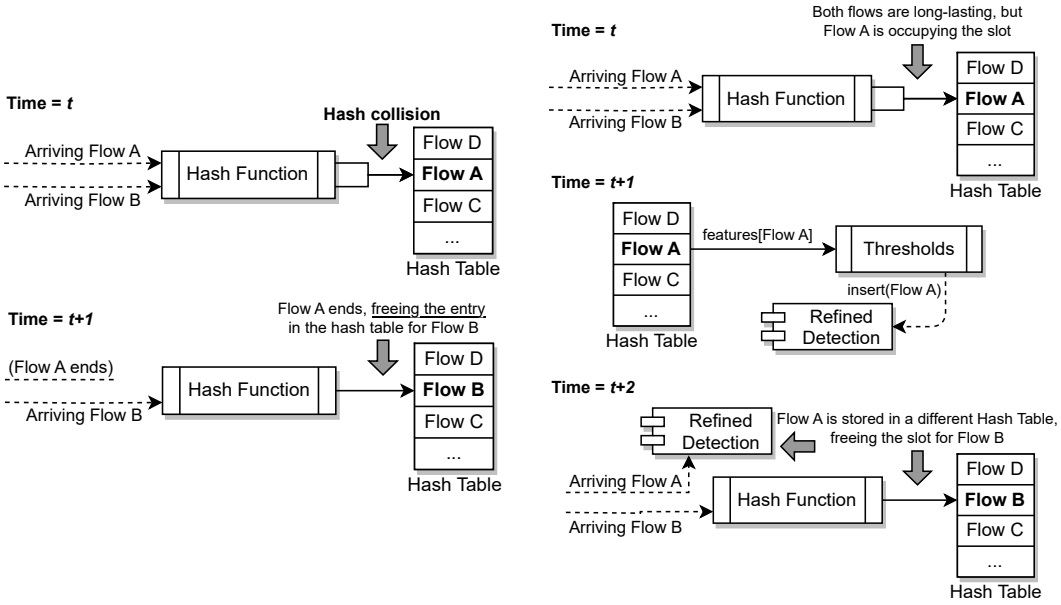
CrossBal utilizes hash tables to implement the *preliminary filtering* and the *refined detection*. Since onboard memory is a scarce resource, collisions in the hash tables are unavoidable. However, due to the multi-stage elephant flow detection spanning both the data and control planes, hash collisions do not cause elephant flows to pass undetected. Figure 3.3 shows examples of hash collisions that may happen during the detection of elephant flows in the data plane. Particularly, when one of the colliding flows is a short-lived flow (Figure 3.3a), this flow tends to complete while the elephant flow is still active. This means that the elephant flow will eventually be able to utilize the hash table entry once the short-lived flow expires. In another scenario (Figure 3.3b), when the hash of two

[3] A Decision Tree (DT) is a Machine Learning model that can learn classification or regression from training data. A Classification Tree (CT) is a subset of DTs that is trained to predict a class for each sample evaluated (QUINLAN, 1987). CTs can be generated from training data with algorithms such as c4.5 (QUINLAN, 1993).

[4] A Random Forest (RF) is an ensemble (POLIKAR, 2006) of classification or decision trees (HO, 1995). By employing several classifiers trained over a different subset of data, RFs provide better accuracy and avoid overfitting the training data (BREIMAN, 2001).

Figure 3.3 – Relevant scenarios where hash collision may happen.

(a) Hash collision of a mice and elephant flows. (b) Hash collision between two elephant flows.



Source: The Authors.

(undetected) elephant flows lead to the same table entry, the first flow (A) will occupy the slot. Eventually, the *preliminary filtering* will recognize flow A as being a potential elephant flow, inserting it in the list of flows tracked by the *refined detection*. This way, flow A will be removed from the first hash table, freeing the slot for the second flow (B). This same reasoning is applicable to hash collisions in the *refined detection*, as flows are eventually exported to the controller, freeing the occupied slot.

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

3.3 Deep Reinforcement Learning Agent

The Deep Reinforcement Learning (DRL) agent is responsible for selecting the routes for active elephant flows that lead to efficient network utilization. In particular, the modeling of the observed state, action space, and reward function impact how well the agent can achieve this goal.

State Space. The state must include all the information the agent needs in order to take an appropriate decision at a given time. However, the state space must be carefully designed in order to avoid communication overheads between the data plane and the controller. Further, excessive or redundant information may negatively impact the learning rate of the agent.

Past research has proposed the utilization of a Demand Matrix (DM) (or Traffic Matrix) as input for a Deep Reinforcement Learning agent (VALADARSKY et al., 2017), and other ML-based frameworks (RUSEK et al., 2019). While the DM for a given iteration (or epoch) could be calculated by polling statistics from switches, this design assumes that there is (at least some) *regularity* in the network traffic. Further, designing an agent based on DMs in a multi-path scenario is challenging, as traffic is not evenly split between shortest-paths. Finally, as our agent *actively* reroutes elephant flows, designing the state space based on past DMs would not provide the agent with enough information about the current state of the network, as the DM would not properly reflect the impact of an action taken by the agent.

Another possible design for the state space is to use the information collected by the programmable switches during the `Preliminary Filtering` and `Refined Detection` steps of the elephant flow detection (previously detailed in Section 3.2). However, exporting this information to the controller would pose scalability challenges, considering the amount of information tracked by the data plane. Further, directly utilizing this information as the state space of the agent would likely lead to slow learning, as most of the flows tracked (especially by the `Preliminary Filtering`) are too small to have a considerable impact in the network.

Our approach: In order to avoid the aforementioned problems, we model the state based on the utilization of each link in the network. More formally, the agent observes LU, a vector of the utilization $U_{i,j}$ of each link $j \in (1, 2, ..., m)$ for every switch[5] $i \in (1, 2, ..., n)$:

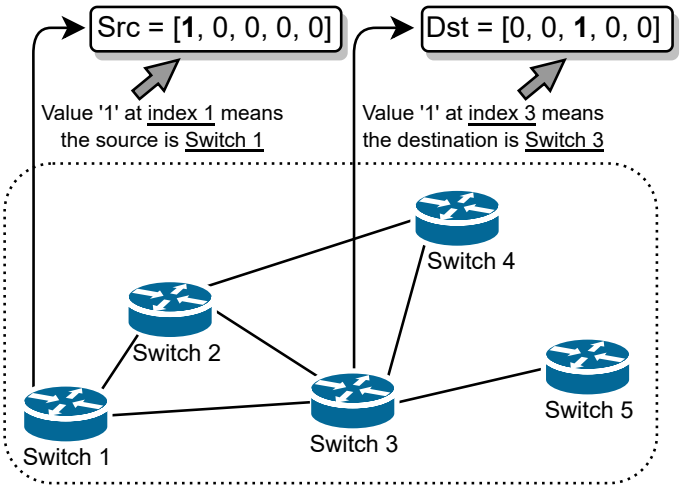
$$LU = (U_{1,1}, U_{1,2}, ..., U_{1,n}, U_{2,1}, ..., U_{2,n}, ..., U_{m,n})$$

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

[5]Each switch can have a different number of links, i.e., m does not have to be the same for every switch i. The notation was simplified to be easier to understand.

Considering that the agent is used to reroute active elephant flows, the state is modeled to also consider the two endpoints of that network flow. However, we find that directly using identifiers such as the 5-tuple of the flow is not the most appropriate solution. This is because requiring the agent to learn the mapping of IP addresses can lead to slower learning. Instead, we map the source and destination IP addresses to the corresponding source and destination edge switches, including numerical identifiers of these switches in the state observed by the agent.

Figure 3.4 – The source and destination switches are encoded in one-hot vectors.



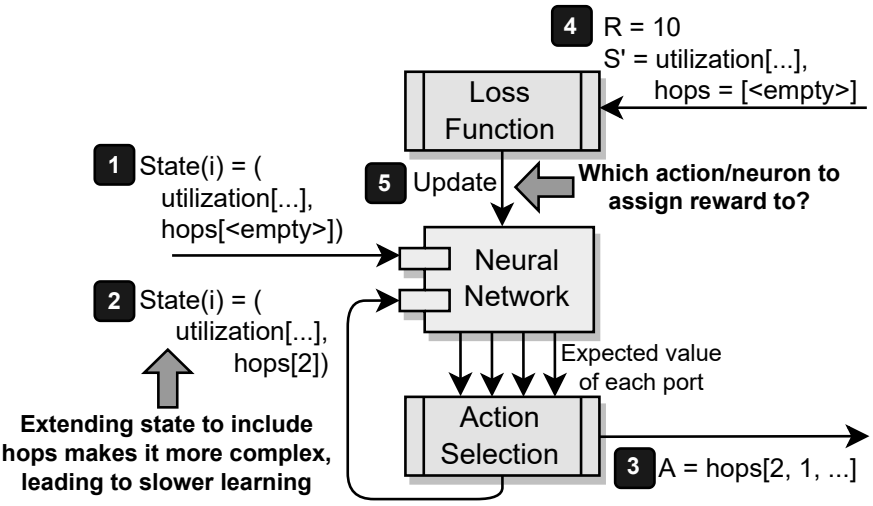
Source: The Authors.

Figure 3.4 shows an example of how we encode the identifiers into one-hot vectors for the state space. Formally, in a topology with n switches $Sw_1, Sw_2, ..., Sw_n$, the source IP address originating from switch Sw_i is mapped into a one-hot vector $Src = (X_1, X_2, ..., X_{n-1}, X_n)$, where $X_i = 1$, and $X_j = 0 \; \forall j \neq i, j \in (1, 2, ..., n)$. The formulation for the one-hot vector Dst for the destination switch is analogous. Therefore, at iteration t, the agent observes the state $S_t = (LU, Src, Dst)$.

Action Space. The action space must allow the agent to make decisions on how to reroute active (elephant) flows. A naïve approach would be to map each possible route to an action. However, this would generate a large number of possible actions, leading to slow learning. An alternative would be to model each action as one hop, as shown in Figure 3.5. In this scenario, (1) the state space would need to include the information of each hop in the end-to-end route. (2) This would increase complexity, possibly slowing down the training of the agent. Further, (3) hop-by-hop routing with Deep Neural Networks can generate paths with persistent loops and blackholes (XIAO et al., 2020). Finally, (4) an end-to-end route would require several hops, i.e., several actions being taken in a sequence. (5) However, the environment would still return a single reward for the composite

route, leading to *reward assignment* issues.

Figure 3.5 – Mapping actions to hops leads to reward assignment issues.



Source: The Authors.

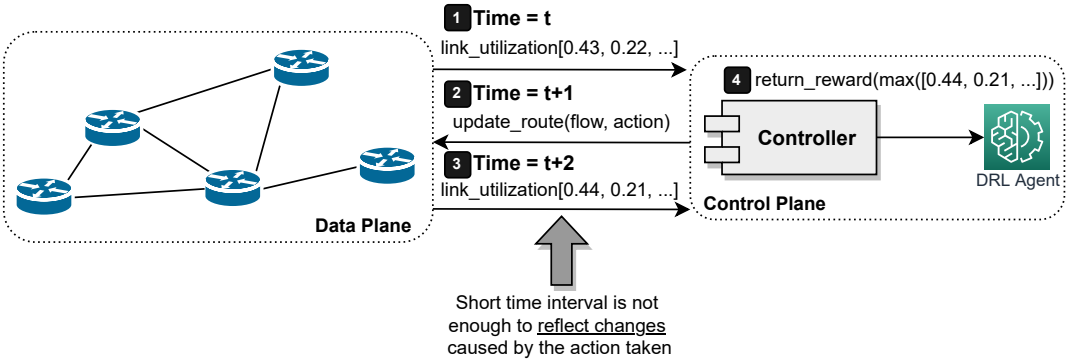
Our approach: We model the action space based on a set of predefined end-to-end routes for each pair of source-destination edge switches. First, this design eliminates the need for multiple actions per end-to-end route. Instead, the agent can observe the correlation between choosing a route (taking an action) and changes in the utilization of the links in the network (observing the next state). Additionally, restricting the action space to a set of predefined routes results in a well-defined number of possible actions. This avoids potential issues that might arise if the agent had to consider every single possible end-to-end route in the network. Finally, the set of end-to-end routes for each source-destination pair should remain the same throughout the life of the DRL agent, i.e., from training to the end of its use in load balancing. While this may seem like it could restrict the agent in achieving optimal load balancing, it is possible to generate several short paths with diversity by using an algorithm such as KSPD[6] (LIU et al., 2018). By employing paths with diversity, the agent is able to utilize several links in the network while keeping the number of possible actions small. While topology changes require recomputing the static routes and retraining the agent with the new routes, Graph Neural Networks (GNNs) could make the agent robust to topology changes (RUSEK et al., 2019). We leave this as future work.

Reward Function. Load balancing aims to improve the utilization of network resources, reducing the congestion of links and the delay observed by applications. In

[6]CrossBal does not define how to compute the set of pre-defined routes. The routes considered by the agent must be supplied by the network operator.

order for the agent to learn how to achieve this goal, we must design a reward function that reflects how well the agent is performing. Therefore, the exact formulation of the reward function can impact the performance of the agent. For instance, a reward function based on the maximum link utilization will train the agent to act differently than an agent trained with a reward function based on the imbalance of network links (e.g., variance of link utilization). Further, while in traditional DRL the impact of an action in the environment is reflected immediately, the state of the links may take some time to update in our context of load balancing the network. Figure 3.6 illustrates the problem with the delayed reward. (1) After querying the previous state (based on the link utilization), (2) the agent selects new routes for an elephant flow. However, (3) if the controller polls the link utilization immediately after installing the new routes, (4) the reward computed based on the link utilization will not properly reflect the impact of the action selected. Naturally, the new routes installed need to be applied for some time before having a noticeable effect on the link utilization.

Figure 3.6 – Links take time to reflect changes caused by the action selected by the agent.



Source: The Authors.

Our approach. We implemented and compared three different reward functions: one based on the maximum link utilization (Equation 3.1), another on the variance of the link utilization, (Equation 3.2), and the last based on the number of inactive Network Interface Cards (NICs) (Equation 3.3).

$$inv\_mlu(s') = \frac{1}{max(link\_utilization(s'))} \tag{3.1}$$

$$inv\_stdev(s') = \frac{1}{standard\_deviation(link\_utilization(s'))} \tag{3.2}$$

$$inv\_nics(s') = \frac{1}{inactive\_nics(s')} \tag{3.3}$$

After evaluating the reward functions above (Section 4.3), we observed that the reward function based on the maximum link utilization performed the best. Therefore, the reward function, $R(s, a, s')$, is given by $inv\_mlu(s')$ (Equation 3.1). Additionally, in order for the new state to reflect the consequences of the action taken, we poll link statistics from switches t milliseconds after an action is taken.

3.4 Reacting to Short-Lived Network Congestion

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

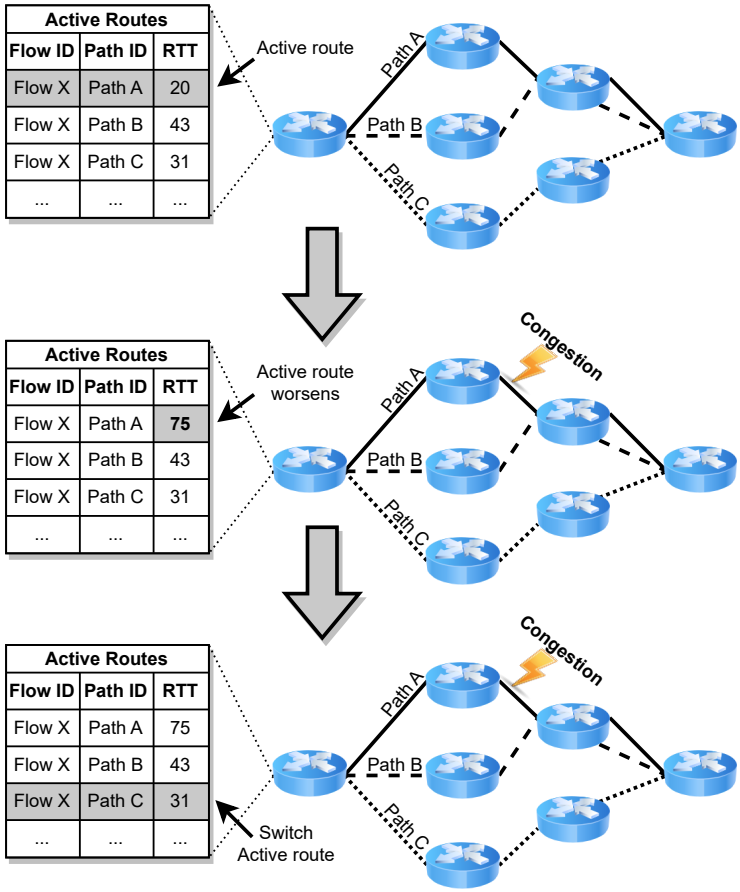
Thus, in order to be able to quickly react to network changes, CrossBal implements mechanisms for switching active paths directly in the data plane. Figure 3.7 presents an overview of this mechanism, where *(i)* the programmable data plane actively monitors the quality of the installed routes, and *(ii)* upon detecting a significant increase in RTT (which may indicate congestion), *(iii)* the forwarding device switches to the precomputed route with lowest RTT without control plane intervention.

CrossBal achieves this by having the controller install multiple routes[7] for each active elephant flow[8]. As the DRL agent computes the expected value of each route, we select the N routes with highest expected value. Also, the data plane devices are responsible for periodically probing each of the installed routes. By calculating the RTT of each route, the programmable switch is capable of detecting congestion along paths and selecting an alternate route. While we could spray the data packets of a flow through paths we wish to probe, this could lead to packet reordering at the destination end-host. Packet reordering can affect transport protocols such as TCP, thereby harming performance (GENG et al., 2016). Instead, CrossBal periodically creates probe packets to measure the RTT of

[7]Each switch only knows the local output port for each path. Only the controller must know every hop of the end-to-end path.

[8]The flow only utilizes one of the installed routes at a given time.

Figure 3.7 – Mechanism for switching active paths



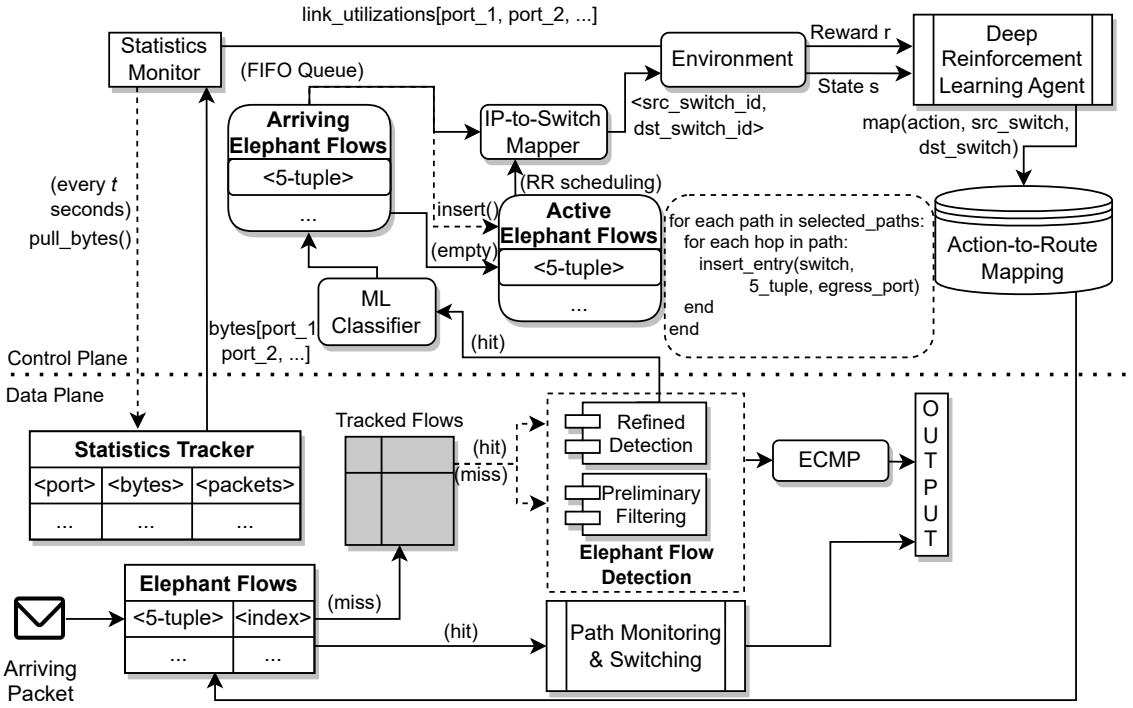
Source: The Authors.

active paths. More specifically, when a new packet of an active elephant flow arrives, if it has been longer than t ms since the last probing, CrossBal utilizes the *clone* feature of programmable switches to generate new packets. The cloned packets have their payload removed and an additional customized probing header is inserted. A packet is sent through each active route to measure the RTT of each route. As the control plane is only involved in the process when first installing the multiple routes, we can quickly detect and react to short-lived congestion.

3.5 Architecture

The collaboration between the logically-centralized controller and the data plane is a key principle of CrossBal. To this end, the architecture of CrossBal comprises a series of modules in the controller and in each programmable switch, as shown in Figure 3.8. Next, we describe each of these modules, starting with the architecture of the Control Plane (Section 3.5.1), before detailing the data plane of programmable switches (Section 3.5.2)

Figure 3.8 – Architectural implementation of CrossBal.



Source: The Authors.

3.5.1 Control Plane

The control plane is responsible for selecting the best routing strategy for active elephant flows and configuring the data plane with the chosen routes. Further, the control plane periodically polls the link utilization from each switch.

As shown in Figure 3.8, the `Statistics Monitor` periodically polls each switch for the bytes sent from each port. This module computes the link utilization based on the time elapsed and bytes sent since the last time the switch was queried. The link utilization of each switch is eventually used by the DRL agent (as previously described in Section 3.3). In our proof-of-concept prototype (further described in Section 4.1) based on BMv2 software switches, the control plane sends packets with a custom header to each switch to request the link utilization. This is done because BMv2 switches currently do not support the full set of operations defined by the P4Runtime Specification, such as reading and writing to registers. Therefore, we had to manually implement a simple protocol to recover and reset the values of these registers.

Further, the `Machine Learning Classifier` module receives flow reports from programmable switches. Flow reports carry a custom header with the 5-tuple of the potential elephant flow and the features tracked by the `Refined Detection` module (detailed next in Section 3.5.2). These reports are fed into the Random Forest (RF) responsible for the final classification of potential elephant flows, as previously described in Section 3.2.

Elephant flows identified by the RF are inserted into a First-in First-out (FIFO) queue of `Arriving Elephant Flows` (Figure 3.8). The flows in this queue are sent to the Deep Reinforcement Learning agent to be rerouted once (in a FIFO fashion), and then moved to a list of `Active Elephant Flows`. During each iteration of the DRL agent, CrossBal first checks if there are any flows in the FIFO queue. If it is empty, then CrossBal selects from the list of active elephant flows in a Round-Robin (RR) fashion.

The `IP-to-Switch Mapper` maps the 5-tuple of a flow into one-hot vectors representing the source and destination switches in order for the DRL agent to efficiently learn how to reroute flows, as previously explained in Section 3.3. The vectors of source and destination switches, along with the array of link utilization calculated by the `Statistics Monitor`, are used by the `Environment` to produce the state observed by the agent. The `Environment` module also computes the reward of the previous iteration based on the current link utilization, as described in Section 3.3.

In each iteration, the `Deep Reinforcement Learning Agent` is queried to compute the *top-n* out of the k routes for an active flow. The k routes considered for each pair of source-destination switches are static and pre-computed offline. This improves controller efficiency and helps stabilize the training of the DRL agent (as previously described in Section 3.3). The `Action-to-Route Mapping` module maps the n highest value actions to end-to-end routes, which are then translated into entries in the `Elephant Flow Table` of the P4-enabled switches (described next).

3.5.2 Data Plane

The data plane includes modules for monitoring network links, routing elephant flows, detecting new elephant flows, monitoring active end-to-end routes, and switching active end-to-end routes. Algorithm 1 provides the pseudo-code of the packet processing pipeline of the P4 switches used by CrossBal.

Forwarding Elephant Flows. The ingress processing begins by checking the match+action table of `Elephant Flows` for an entry that matches the 5-tuple of the arriving packet. The entries in this table are inserted by the control plane after detecting elephant flows. Each entry is a tuple ($flow\_tuple$*, $elephant\_index$, $path\_hops$, $last\_hop$, $version$). Below we detail the entries in the `Elephant Flow Table`:

- ***flow_tuple**** is the match key of the table. In practice, it is composed of the five fields that define a flow: source IP address, destination IP address, source TCP/UDP port, destination TCP/UDP port, and protocol (TCP or UDP). As each entry matches on an exact 5-tuple, we insert two entries for each elephant flow: one matching packets from client to server, and another matching packets from server to client. In our proof-of-concept prototype, the two entries inserted by the controller have symmetrical routes (in opposite directions). This can be easily extended by modifying the control plane software, as the implementation of the data plane already supports asymmetrical routes.
- ***elephant_index*** is used by the switch when accessing registers for this elephant flow. This includes the registers that keep track of the last two RTTs measured for each route, the timestamp when the paths for this flow were last probed, and so on.
- ***path_hops*** is an array with the output port (current hop) for each path installed for this elephant flow. As P4 does not support arrays, it is implemented as n parameters,

one for each of the n paths installed for this flow. Therefore, each switch only knows its local hop for each of the n paths.

- ***last_hop*** is a flag that tells the switch if this is the last hop for this flow. We assume that if a switch is the last hop for one path for this elephant flow, it is also the last hop for every other path for this flow. However, this can be easily extended by replacing a single flag with an array of flags - one for each of the n paths installed.
- ***version*** is an identifier for the current version of the installed paths. It is incremented each time the controller updates the routes for this elephant flow. This is used to ensure that paths are probed whenever they are updated. It could also be used to ensure that in-flight packets are not affected by the controller updating the routes installed for the elephant flow. This could be done by keeping a "shadow copy" of the previous installed routes. We leave this as future work.

As explained in Section 3.5.1, the controller periodically queries the `DRL Agent` to select better routes for active elephant flows. However, due to its location in the control plane, it may not be able to quickly react to short-lived congestion. Therefore, the data plane implements mechanisms for `Path Monitoring and Switching` (lines 2-8 of Algorithm 1). The data plane can freely switch between the n routes installed for each elephant flow. Each route has a different register array[9] to keep track of its two last observed RTTs. The route selection happens by electing an active route, which remains selected until the RTT of that route worsens by a certain threshold (e.g., 200%). Upon detecting a degradation in the selected route, the data plane picks the route with the lowest last measured RTT (lines 2-4 of Algorithm 1). Further, the data plane periodically generates probe packets (lines 5-7 of Algorithm 1) to measure the quality of the routes installed for this flow. These packets are generated by using the *clone* feature of P4-enabled switches and carry a custom header.

The `Path Monitoring and Switching` is only applied by the first switch that processes the packets of an elephant flow. As illustrated in Figure 3.9, after selecting a path according to the logic described above, the first switch inserts a custom header to the packets of the elephant flow. This header is used to inform the active (chosen) path to subsequent switches. This relieves subsequent switches from having to do any complex processing, simply having to forward the packet according to the active path (listed in the header) and the matching output port (according to the corresponding entry in the

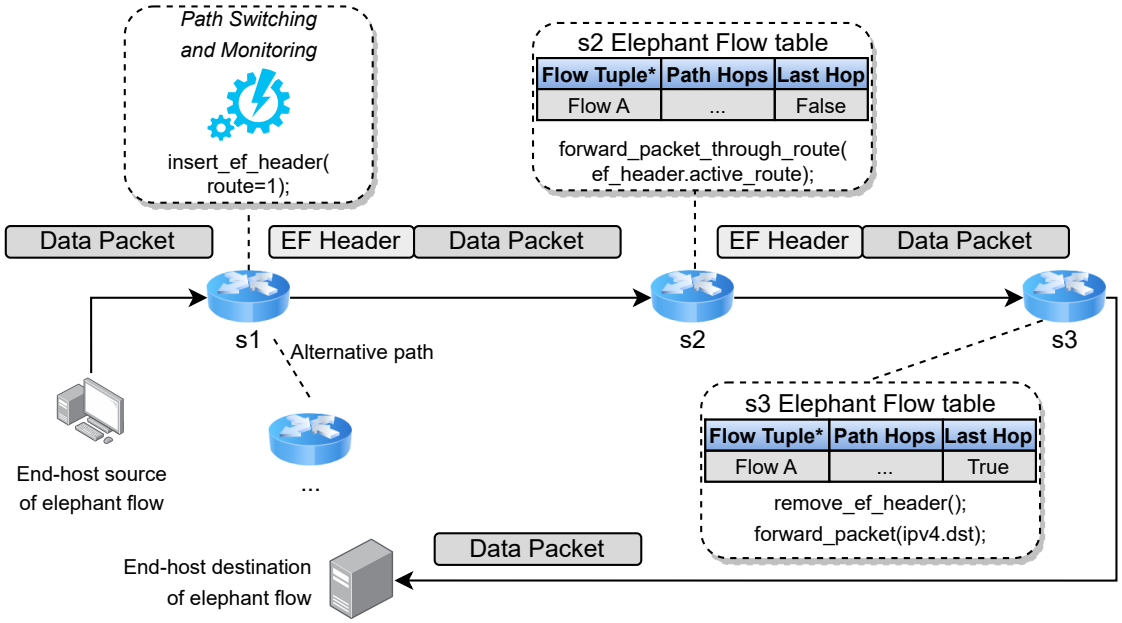
[9]The index used for each active elephant flow is configured by the controller upon installing new routes, allowing the controller to avoid any collisions.

Algorithm 1: Data plane packet processing pipeline

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

match+action table of Elephant Flows). Finally, the last switch (indicated by the $last\_hop$ flag) removes this header before forwarding the packet to the destination end-host.

Figure 3.9 – The first switch selects an end-to-end route and inserts a custom header.

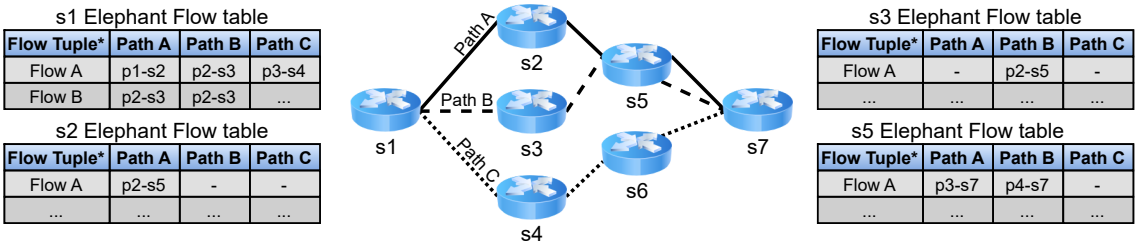


Source: The Authors.

Figure 3.10 shows an example of how switches use the entries in the `Elephant Flow Table` to forward the packets of the corresponding flow. For example, assume that Flow A has been detected by the cross-plane elephant flow detection mechanisms. Then, the controller queried the DRL agent, which selected Path A, Path B, and Path C as the best routes for this flow (as explained in Section 3.5.1). Finally, the controller installs the corresponding entries for Flow A in the Elephant Flow table of each switch (Figure 3.10). Assuming that packets from Flow A first arrive at switch s1, it will be responsible for applying `Path Monitoring and Switching`, selecting one of Path A, Path B, or Path C. If it selects Path A, it will add a custom header and then forward the packet through output port p1 to switch s2. Then, switch s2 will read the active path from the custom header (Path A) and use its Elephant Flow table to select the output port p2 (to switch s5). Note that s2 does not have entries for Path B or Path C, as these paths do not traverse s2. The same logic is applied at switch s5, which decides between two output links (p3 and p4) to switch s7.

Processing regular flows. Upon receiving a packet that does not belong to an elephant flow, the switch applies a hash function over the 5-tuple of the packet. The resulting hash is used to check if the packet belongs to a flow in the list of `Tracked Flows` (line 10 of Algorithm 1). If the flow is not being tracked, the switch applies

Figure 3.10 – Switches use the Elephant Flow Table to forward packets of corresponding flows.



Source: The Authors.

a `Preliminary Filtering` to exclude short-lived and small flows (lines 14-23 of Algorithm 1). For each flow, we use registers to track a few simple features, such as the number of bytes, packets, and flow duration. When the features of a flow exceed predefined thresholds, the flow is set to be processed by a second module (lines 22-23 of Algorithm 1) that implements a more elaborate mechanism.

The `Refined Detection` is only applied over a smaller subset of flows, enabling the data plane to keep track of more complex features for each tracked flow (lines 10-13 of Algorithm 1). The features tracked in this module include the minimum, maximum, and total inter-arrival-time and packet length of each flow. These features are tested against a set of chained conditions in order to identify potential elephant flows (line 12 of Algorithm 1). This is achieved by converting a Classification Tree to a series of conditions. If the flow is labeled as a potential elephant flow, it is exported to the controller for a final classification, along with the computed features of that flow (line 13 of Algorithm 1).

The data plane further implements the `Statistics Tracker`, which is responsible for monitoring statistics of each switch port. This module tracks the bytes received in each ingress port (implemented in the *ingress block*) and the bytes sent through each egress port (implemented in the *egress block*). At this time, the control plane only polls the statistics for each ingress port, which is used to compute the array of link utilization used by the Environment module in the control plane.

4 EVALUATION

This chapter presents the evaluation of a proof-of-concept (PoC) prototype of CrossBal. First, Section 4.1 details the PoC prototype that was implemented and evaluated. Next, Section 4.2 describes the methodology employed in our evaluation. Then, Section 4.3 shows the results of our evaluation of the load balancing capabilities of CrossBal. Finally, Section 4.4 presents an analysis of the optimizations for the preliminary filtering mechanism that we proposed previously in Section 3.2.

4.1 Prototype

The PoC prototype includes both data plane and control plane software. For the data plane, we wrote over 1800 Lines-of-Code (LoC) of P4 source-code for the BMv2[1] software switch. The control plane software comprises over 2400 LoC written in Python 3 and depends on several libraries listed next. We used graph-tool v2.45 (PEIXOTO, 2023) to compute the ECMP routes and to help with the top-k routes that constitute the action space of the agent. We also used Scapy v2.5.0 (SCAPY, 2023) to send and receive packets between the controller and the software switches[2]. In order to facilitate and speedup some of the computing, we used NumPy v1.23.4 (NUMPY, 2023). Finally, to implement the Deep Reinforcement Learning agent, we used PyTorch v1.12.1 (PYTORCH, 2023) for the DNN and Gym v0.26.2 (GYM, 2023) to create a custom Reinforcement Learning environment.

4.2 Methodology

This section describes the methodology employed in our experiments. First, we detail the setup utilized in our experiments (Section 4.2.1). Next, we characterize the workloads utilized in our evaluation (Section 4.2.2), followed by the topology used in our emulated setup (Section 4.2.3). Then, we list system and experiment parameters for our evaluation (Section 4.2.4). Finally, we describe the metrics measured in our evaluation

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

[2] BMv2 switches currently do not support the full set of operations defined by P4Runtime, such as reading and writing to registers. Therefore, we had to send packets from the controller to each switch.

Table 4.1 – Workloads used in our evaluation.

Workload A		Workload B	
Flow Size	**Distribution**	**Flow Size**	**Distribution**
20 KB	0.5	10 KB	0.2
200 KB	0.3	100 KB	0.4
2 MB	0.1	1 MB	0.2
20 MB	0.1	10 MB	0.2

Source: The Authors.

(Section 4.2.5).

4.2.1 Setup

We emulated a network using Mininet[3] v2.3.1b1. All experiments were performed on a virtual machine (VM) with guest Operating System (OS) Ubuntu 20.04.5 (focal) on a Windows 10 host. The VM was configured with 16GB RAM and access to all 8 physical cores of the host CPU. Oracle VirtualBox[4] v7.0 was used to host the VM, enabling PAE/NX and nested AMD-V in the extended features.

Considering the experiments were performed in a single VM, we had to down-scale the workloads and the topology (described next in Section 4.2.2 and Section 4.2.3, respectively). We leave a more thorough evaluation in a distributed setup as future work.

4.2.2 Workloads

In order to evaluate CrossBal, we performed experiments based on realistic workloads. In particular, we based our workloads on the work of Pizzutti and Schaeffer-Filho (2019), which was originally inspired by Alizadeh et al. (2014) and Vanini et al. (2017).

Due to restrictions in our experimental setup, we had to reduce the flow sizes in the original workload. The flow size distribution of the workloads used in our experiments is described in Table 4.1. Each switch in the topology has one host connected directly to it. Each host runs both a client and a server script. The client from each host independently generates requests to randomly chosen servers (from other hosts) according to a Poisson distribution based on the workload and the desired network load.

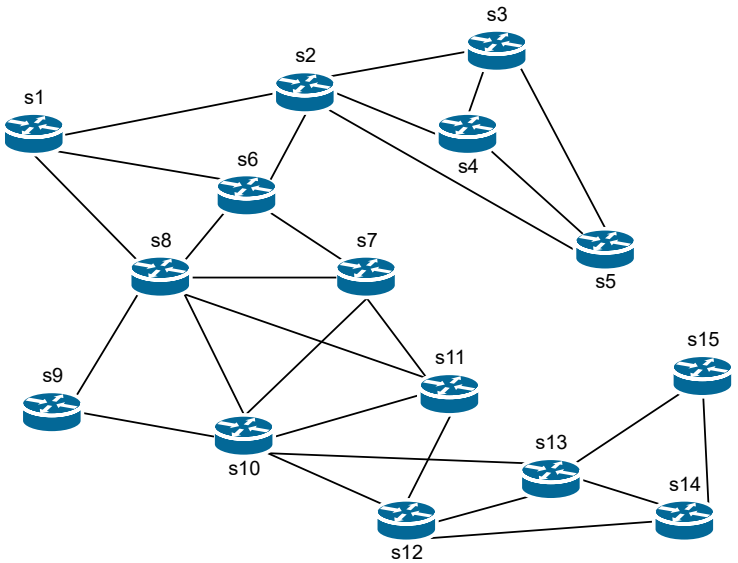
[3]Mininet: <https://mininet.org/>
[4]VirtualBox: <https://www.virtualbox.org/>

4.2.3 Topology

The topology used in our experiments was generated using iGen, with parameters inspired by Pizzutti and Schaeffer-Filho (2019). Due to hardware limitations in the setup used in our experiments, we restricted the topology to 15 switches and the emulated link speeds to 50 Mb/s. This includes the links from hosts to switches and the links between switches. We did not impose any limits on the links from each switch to the controller.

Figure 4.1 shows the topology used in our experiments. In order to obtain this topology, we first used iGen to generate 15 routers in a single domain, ensuring that all the routers were placed in continents, with the specified area set to “North America”. Then, we had iGen generate an intradomain mesh with the selected method “two-trees”. These settings can be used to generate a Hub & Spokes topology (LUIZELLI et al., 2016). According to Kamiyama et al. (2010), Hub & Spokes is a realistic class of topology that is characterized by nodes with aggregation function (PIZZUTTI; SCHAEFFER-FILHO, 2019), i.e., routers with a high degree of connectivity (hubs).

Figure 4.1 – Topology used in our experiments.



Source: The Authors.

The topology emulated in mininet only considers the routers and the links between them. While iGen also exports positional information, our setup does not emulate the physical distance between routers (P4-enabled switches).

Table 4.2 – CrossBal and Deep Reinforcement Learning agent parameters used in the evaluation.

CrossBal parameters

Parameter	Value	Description
k	3	Number of routes installed for each detected elephant flow.
n	10	Number of routes the agent considers (action space) for each detected elephant flow.
$time\_step$	0.5s	This is how long the controller waits after polling the link utilization from switches. It also dictates how often the DRL agent is queried.
$probing\_interval$	1s	Time between probing each path installed for an elephant flow.
$detection\_threshold$	30KB	Flow size threshold for the `Preliminary Filtering`.
$flowlet\_timeout$	50ms	The timeout used for flowlet switching and in the `Preliminary Filtering`.

Deep Reinforcement Learning agent parameters

Parameter	Value	Description
$hidden\_layers$	2	The number of hidden layers used in the Deep Neural Network.
$hidden\_layer\_size$	512	The number of neurons in each hidden layer of the Deep Neural Network.
$activation\_function$	$ReLU$	The activation function used to connect the layers.
$burn\_in$	150	The number of iterations the agent experiences before training.
$learn\_every$	10	The number of iterations between training the online network.
$batch\_size$	32	The number of samples in each batch used to train the agent.

Source: The Authors.

4.2.4 Parameters

Table 4.2 lists the parameters used for CrossBal and the Deep Reinforcement Learning agent in our experiments. These values were set through a combination of intuition and experimentation. We leave a more thorough analysis of the parameters as future work.

The CrossBal parameters k and n define how many routes are installed for each elephant flow and how many are considered for each source-destination pair, respectively. These parameters were set considering the size of the topology and the average number of paths between endpoints. The remaining parameters listed in Table 4.2 were set based on our setup and workloads, as described in Section 4.2.1 and Section 4.2.2, respectively.

The Deep Reinforcement Learning agent parameters $burn\_in$ and $sync\_every$ were set considering the agent is queried every $0.5s$ ($time\_step$). As the first few iterations of each experiment happen while the network is being underutilized[5], we want the agent to avoid focusing on the earlier experiences. The remaining parameters were used as a proof-of-concept prototype. We leave a more thorough exploration of the configuration of the DNN as future work.

Table 4.3 lists the experiment parameters used in our evaluation. The parameters for experiments include characteristics of the emulated topology and values that were calculated based on the workloads, as previously described in Section 4.2.3 and Section 4.2.2, respectively. These parameters are used by the clients to generate requests, according to the rationale previously explained in Section 4.2.2. In order to facilitate the im-

[5]This is due to the fact that the clients are just starting to generate requests, as explained in Section 4.2.2.

Table 4.3 – Experiment parameters used in the evaluation.

Experiment parameters

Parameter	Value	Description
$host\_MTU$	1400B	Size of the MTU for hosts. This was decreased to allow switches to insert additional headers without changing the MTU of the links between switches.
$average\_request\_size$	2.27MB 2.242MB	This is calculated based on the workload used. It is a weighted sum of the size and proportion of each flow type. The first value was calculated for Workload A and the second for Workload B (§4.2.2).
$link\_bandwidth$	50Mbps	The bandwidth for the network links. We use this value for links between switches and the links that connect switches to end-hosts, as previously explained (§4.2.3).
$desired\_load$	0.7	Proportion of the available bandwidth we wish to utilize in the experiments.
$available\_paths$	3	An estimate of the average number of paths between any source-destination pair.
$load\_multiplier$	$desired\_load * available\_paths$	This value is used to compute the average requests per second for each client.
$request\_rate$	$\frac{link\_bandwidth * load\_multiplier}{average\_request\_size}$	The number of new requests per second each host generates. This value is calculated based on the maximum link bandwidth and the load multiplier.
$request\_window$	100s	We use a poisson distribution to generate r requests for each request window. The r requests are then evenly distributed during the request window.
$experiment\_duration$	300s	The duration of each experiment. New requests are generated during this time, after which we wait for every flow to complete, even if it takes longer than this duration.
$repetitions$	40	The number of times each experiment was repeated for each configuration, such as reward function (described in §4.3), load balancer (CrossBal or ECMP), and workload (§4.2.2).

Source: The Authors.

plementation of our experiments, we configured the MTU of hosts to 1400 bytes, as shown in Table 4.3. However, the actual overhead of custom headers is orders of magnitude smaller - e.g., the routing header added to packets of detected elephant flows only requires 3 bytes. As shown in Table 4.3, we performed 40 repetitions for each experiment. During each experiment, the clients generated new requests for $300s$ ($experiment\_duration$). After this time, the experiment continued until every existing request completed.

4.2.5 Metrics

The main objective of CrossBal is to efficiently utilize available network resources, balancing the network load across the existing links. Due to limitations in the setup used during the experiments, the software switches presented high variance in packet throughput and the time spent processing each packet[6]. This happened even when emulating smaller topologies (e.g., less than 15 routers) with lower link bandwidth (e.g., lower than 50Mbps). Therefore, we opted to not use the flow completion time (FCT) as the main metric in our evaluation. Instead, we measured the following metrics in our experiments:

[6]It is worth noting that our design is aimed at programmable data plane hardware, where the time spent processing each packet is near constant.

- **Active Network Interfaces**. The percentage of active network interfaces is able to show how many links are not being utilized at each point of the experiment. Intuitively, a higher value is better, as more links are being utilized.
- **Link Utilization Imbalance**. This metric is defined as $\frac{max(LU)-min(LU)}{mean(LU)}$, where LU is an array with the utilization of each link in the network (ALIZADEH et al., 2014). Intuitively, a lower imbalance is better, as the network load is being more evenly distributed.
- **Link Utilization Standard Deviation**. The standard deviation can also be used to represent the imbalance in the utilization of network resources. Intuitively, a lower standard deviation is better, as the network load is being more evenly distributed.
- **Maximum Link Utilization**. This metric also represents how well the network load is being balanced. Intuitively, a lower maximum link utilization is better, as no single link is being overutilized.

As the links connected directly to end-hosts depend on the amount of data being sent to/from the host, we did not include these links in the calculation of any of the metrics listed above. Further, links between the controller and each switch were also not considered when calculating these metrics.

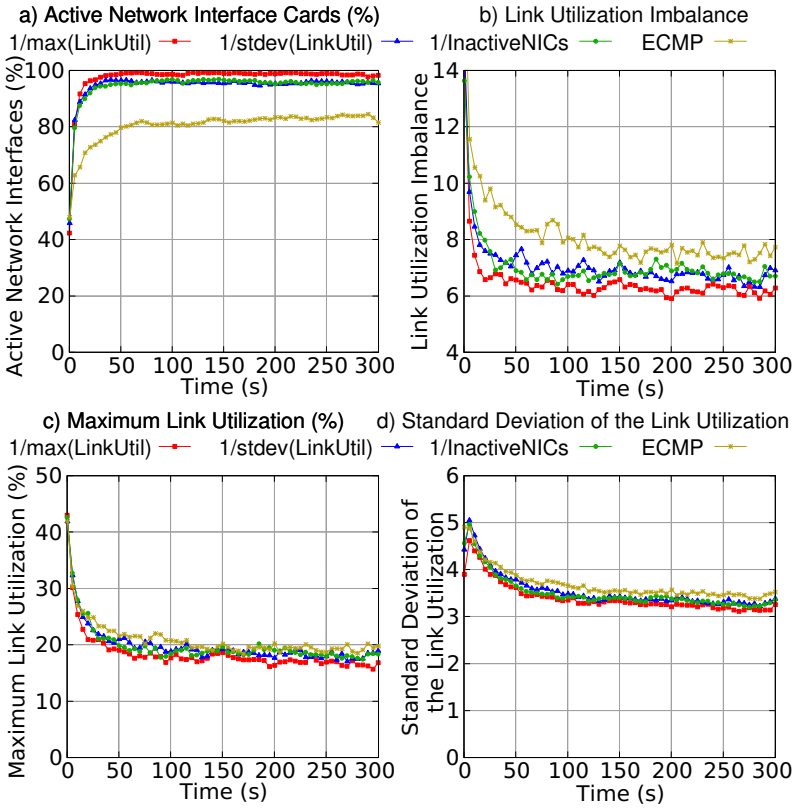
4.3 Link Utilization Analysis

We implemented and compared three different reward functions for the DRL agent (as previously mentioned in Section 3.3): (A) $inv\_mlu(s') = \frac{1}{max(link\_util)}$, (B) $inv\_stdev(s') = \frac{1}{stdev(link\_util)}$, and (C) $inv\_nics(s') = \frac{1}{inactive\_nics}$, where $link\_util$ is an array with the utilization of every link in the network and $inactive\_nics$ is the proportion of NICs not being utilized.

Figure 4.2a shows an analysis of the ratio of active links during our experiments. We can observe that CrossBal effectively utilizes nearly all available links in the network, while ECMP is incapable of utilizing as many links concurrently. Further, we can observe that the agent trained with the reward function based on the maximum link utilization, $\frac{1}{max(link\_util)}$, quickly learns how to actively use nearly every link in the network, effectively distributing the workload. Additionally, Figure 4.2b compares the Link Utilization Imbalance[7] of each approach. Again, CrossBal performs a better job at balancing

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

Figure 4.2 – Link analysis for workload 1.



Source: The Authors.

link utilization across network links compared with ECMP. As before, we can observe that the agent trained with the reward function based on the maximum link utilization, $\frac{1}{max(link\_util)}$, outperforms the agents trained with the other reward functions.

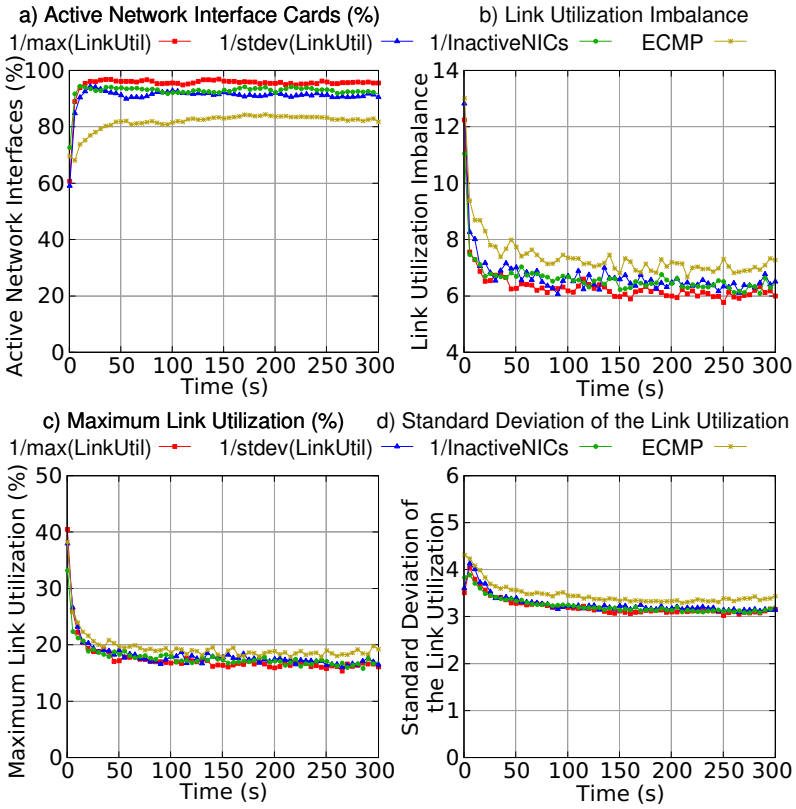
The other metrics used in our evaluation, maximum link utilization (Figure 4.2c) and the standard deviation of the utilization of links (Figure 4.2d), show results consistent with the analysis of the previous metrics. The maximum link utilization (MLU) (Figure 4.2c) could not hit 100% due to the limited setup for our experiments, as the software switches became the bottleneck. While the link speeds could not be saturated, the load balancing by CrossBal still consistently resulted in lower MLU than ECMP.

Figure 4.3 shows the results for each metric when performing the same experiments with a different workload. Workload B presents a distribution tailored towards heavier flows, as previously described in Section 4.2.2. However, the heavier flows in this workload are smaller than the ones in the first workload.

Figure 4.3a shows the ratio of active NICs during this evaluation, where we can observe a similar behavior from ECMP, but a slightly worse performance by CrossBal when compared with the previous workload (Figure 4.2a). This can likely be explained

link utilization (ALIZADEH et al., 2014).

Figure 4.3 – Link analysis for workload 2.



Source: The Authors.

by the larger number of heavier flows. As we did not change how often the DRL agent is queried ($time\_step$ in Table 4.2), the controller rerouted a smaller ratio of heavy flows.

Further, Figure 4.3b shows the link utilization imbalance during the experiments with the second workload. In this aspect, CrossBal had a similar performance when compared to the previous workload (Figure 4.2b). We can also notice that ECMP had a slightly better performance than before, which can possibly be explained by the fact that the flows in this workload are *smaller* than in the previous distribution. Therefore, hash collisions caused by the simple load balancing in ECMP are less prone to lead to link utilization imbalances.

While both CrossBal and ECMP showed slight changes in performance with different workloads, the overall patterns remained the same: CrossBal generally outperforms ECMP. Further, the agent trained with the reward function based on the maximum link utilization, $\frac{1}{max(link\_util)}$, generally performs the best out of the three agents trained.

4.4 Elephant flow detection optimizations

The preliminary filtering mechanism (previously described in Section 3.2) in the data plane must be able to analyze a large number of total flows in order to filter a small number of possible elephant flows. Therefore, it is crucial to optimize the per-flow processing and storage requirements as much as possible. An optimization mentioned in Section 3.2 is to filter packets according to a specific threshold, only accounting for packets that are not too small. This way, rather than counting bytes, it becomes possible to *count packets*, while still having a lower bound of the size of the flow.

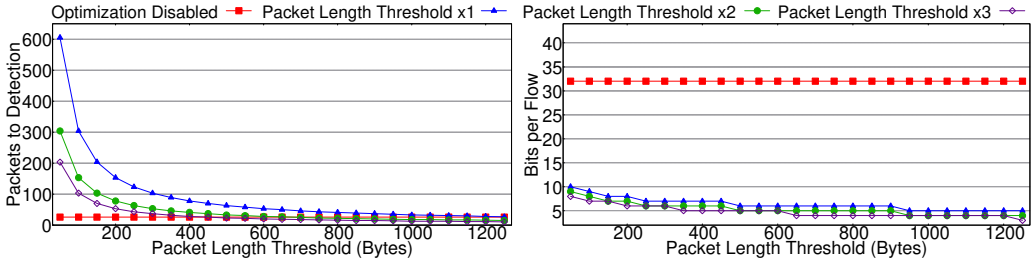
However, only accounting for the lower bound of the size of a flow may cause detection to take longer. For instance, with a packet length threshold of 500 bytes, it would take 20 packets (of at least 500 bytes) to reach a threshold of 10KB. However, by counting bytes, 7 packets of 1500KB (typical MTU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

Figure 4.4a compares the number of packets required for the `Preliminary Filtering` to forward a flow to the `Refined Detection` according to different thresholds for packet length, considering a detection threshold of 30KB[8]. We can observe that the number of packets required to identify a potential elephant flow (forward it to the next step) decreases as the packet length threshold increases. This behavior can be explained by the fact that larger flows tend to have packets with significant payloads. Therefore, a low threshold would incorrectly assume that most packets are relatively small, delaying detection as explained above.

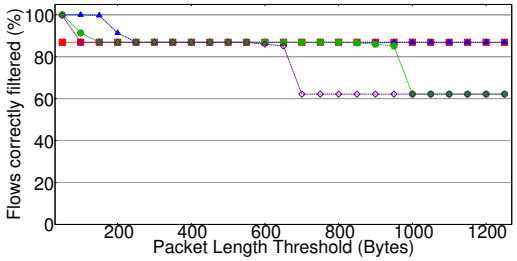
Figure 4.4b shows the number of bits required to keep track of the size of each flow. Assuming a standard width of 32 bits for the unoptimized approach, we can observe that the optimization requires, at worst, 3x less bits (10 bits). Additionally, the number of bits required per flow can be further optimized by increasing the packet length threshold. However, Figure 4.4c shows that increasing the packet length threshold also decreases the filtering accuracy, i.e., increases the number of flows incorrectly reported as possible elephant flows and forwarded to the next step, the `Refined Detection` module. Therefore, with different parameters, we can choose a trade-off between detection speed, memory efficiency, and detection accuracy. Finally, it is worth noting that elephant flows

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

Figure 4.4 – Analysis of parameters for elephant detection optimization with a 30KB threshold.
(a) Packets required for preliminary filtering. (b) Bits required per flow.



(c) Filtering accuracy.



Source: The Authors.

are also long-lived flows. However, we leave the optimization of the duration threshold in the elephant flow detection as future work.

5 RELATED WORK

In this chapter, we discuss related work in network load balancing. First, Section 5.1 details how we characterize the related work discussed in this chapter. Then, we analyze the literature in load balancing in the following sections. First, in Section 5.2, we discuss approaches that delegate load balancing to the data plane. Next, in Section 5.3, we move to the control plane, comparing related work that perform the bulk of processing for path selection in the controller. Then, Section 5.4 covers systems that do not fit in either of the aforementioned categories: *end-host* and *hybrid* load balancers. Finally, Section 5.5 concludes the chapter with a brief discussion about the state-of-the-art in network load balancing.

5.1 Characterization of the Reviewed Literature

In order to provide a comprehensible comparison of the related work, we focused on the following main characteristics of each work:

- **Decision Plane** is the plane responsible for the strategy employed to route flows. In particular, we identified load balancers that utilize the *data plane*, while others utilize the *control plane* to implement a load balancing strategy. In addition to these, we identified two emerging types of systems: *end-host* and *hybrid* load balancers.
- **Path Generation** specifies the type of strategy used to load balance. We classify existing work into systems that utilize *Machine Learning* to route flows, and systems that employ less complex *heuristics* for path generation.
- **Path Selection** refers to how the load balancers select a path for each flow (or packet). *Per-hop* path selection is similar to decentralized routing strategies, where each switch (or router) locally selects a hop according to its strategy, while *fine-grained* strategies select an end-to-end route, generally at the first switch (or at the controller). There are also systems that compute *link weights* for WCMP-based load balancing (previously described in Section 2.2.2).
- **Topology** is the general structure of computer networks that the load balancer supports. While most systems can support any *generic* topology, a few are limited to *datacenter* topologies, such as multi-rooted trees.

Table 5.1 – Data Plane load balancers.

Path Selection	Related Work	Benefits	Limitations
Per-hop	HULA (KATTA et al., 2016) LetFlow (VANINI et al., 2017) MP-HULA (BENET et al., 2018) BurstBalancer (LIU et al., 2022b)	Data plane decision-making and per-hop selection lead to high scalability and responsiveness.	Heuristic-based path generation can lead to suboptimal network utilization. Per-hop path selection provides less control over the end-to-end route.
Fine-grained	CONGA (ALIZADEH et al., 2014)	Data plane decision-making leads to high responsiveness. Fine-grained path selection provides control over the end-to-end route.	Heuristic-based path generation can lead to suboptimal network utilization. Fine-grained path selection leads to scalability issues as the topology grows.

Source: The Authors.

5.2 Data Plane Load Balancers

This section compares state-of-the-art load balancers implemented on the data plane. Table 5.1 highlights the main characteristics of each work.

Data plane load balancers employ simple *heuristics* to generate and select paths. This is due to the limited hardware of programmable switches, which impose restrictions on the per-packet processing, as previously explained in Section 2.1. While offloading this task to the data plane allows reactive load balancing, the decision-making process is inferior to Machine Learning-based systems (discussed next in Section 5.3). It is also worth mentioning that only LetFlow (VANINI et al., 2017) is designed for any topology out of the approaches in Table 5.1, with the remaining load balancers being limited to datacenter topologies.

During our review of the literature, we found two main techniques for *path selection* being employed in data plane load balancers: (i) per-hop (decentralized) and (ii) end-to-end, fine-grained (centralized) path selection. In (i) per-hop path selection, each switch locally decides on the best next hop for the packet. This includes **random path selection** similar to ECMP (e.g., LetFlow (VANINI et al., 2017), BurstBalancer (LIU et al., 2022b)), and choosing the **least congested next hop** (e.g., HULA (KATTA et al., 2016), MP-HULA (BENET et al., 2018)). As an alternative, (ii) end-to-end path selection provides more control over the path taken by each flow (e.g., CONGA (ALIZADEH et al., 2014)). However, implementing end-to-end path selection entirely in the data plane can lead to scalability issues in large networks.

5.3 Control Plane Load Balancers

This section covers load balancers that implement path generation and selection in the control plane. In order to provide a more in-depth analysis, we first cover heuristic-

Table 5.2 – Control Plane load balancers based on heuristics.

Path Selection	Related Work	Benefits	Limitations
Fine-grained	Hedera (AL-FARES et al., 2010) Chameleon (BEMTEN et al., 2020) Mahout (CURTIS; KIM; YALAGANDULA, 2011)	Controller has a global view of the network, leading to better path selection.	Control Plane involvement compromises scalability and responsiveness. Heuristic-based path generation provides decision-making inferior to Machine Learning-based approaches.
Multi-Controller	HELIX (ZAICU et al., 2021) BlastShield (KRISHNASWAMY et al., 2022)	Multi-controller implementation provides logically centralized path selection, improving scalability and responsiveness.	Heuristic-based path generation can lead to suboptimal network utilization. Controller involvement still provides lower responsiveness. Design focuses on WANs.
Link Weights	Parham et al. (2021) Magnouche et al. (2021) Le et al. (2021)	Path selection based on link weights is highly scalable.	Path selection based on link weights can lead to suboptimal network utilization due to hash collisions. Controller involvement limits responsiveness.

Source: The Authors.

based systems (Section 5.3.1), before comparing Machine Learning-based systems (Section 5.3.2).

5.3.1 Heuristics-based Controllers

Table 5.2 provides an overview of heuristic-based control plane load balancers. Out of the systems listed in the table, only Hedera (AL-FARES et al., 2010) is limited to datacenter topologies, while the others support any topology.

A number of load balancing systems implement *fine-grained* path selection in the control plane, as resources are less scarce and the programming model is more flexible (e.g., Hedera (AL-FARES et al., 2010), Chameleon (BEMTEN et al., 2020), Mahout (CURTIS; KIM; YALAGANDULA, 2011)). However, requiring controller involvement to select an end-to-end route leads to severe scalability and responsiveness issues. While *multi-controller* approaches can improve scalability, heuristic-based path generation and selection can lead to suboptimal network utilization. Further, multi-controller approaches are generally more focused on Wide Area Networks (WANs) (e.g., HELIX (ZAICU et al., 2021), BlastShield (KRISHNASWAMY et al., 2022)).

Alternatively, several systems employ heuristics to compute *link weights* for WCMP (e.g., Parham et al. (2021), Magnouche et al. (2021), Le et al. (2021), DOTE (PERRY et al., 2023)). However, as previously explained in Section 2.2.2, WCMP requires control plane intervention to update weights, limiting its ability to detect and react to transient congestion. Further, there are Machine Learning-based systems capable of providing near-optimal weights for WCMP (covered next).

Table 5.3 – Control Plane load balancers that employ Machine Learning models.

Path Selection	Related Work	Benefits	Limitations
Link Weights	Valadarsky et al. (2017) DRL-TE (XU et al., 2018) DPRO (LI et al., 2020) DOTE (PERRY et al., 2023)	ML-based approaches can compute near-optimal link weights.	Path selection based on link weights can lead to suboptimal network utilization due to hash collisions. Controller involvement limits responsiveness.
Strategy	GQNN (GEYER; CARLE, 2018) RouteNet (RUSEK et al., 2019) RBB (RUSEK et al., 2022)	ML-based approaches can learn more efficient routing strategies.	Does not utilize redundant paths, potentially leading to suboptimal network utilization. Path selection provides no responsiveness to congestion.
Fine-grained	NGR (XIAO et al., 2020)	ML-based approaches can select end-to-end routes that optimize network utilization.	Fine-grained path selection based on ML severely limits scalability and responsiveness due to controller involvement.

Source: The Authors.

5.3.2 Machine Learning-based Controllers

Table 5.2 provides an overview of Machine Learning-based control plane load balancers. A common characteristic of these systems is that they can support any type of topology.

Machine Learning-based approaches can compute near-optimal *link weights* for WCMP (e.g., DPRO (LI et al., 2020), DRL-TE (XU et al., 2018), Valadarsky et al. (2017)). However, WCMP-based load balancing systems suffer from several issues, as previously mentioned in Section 2.2.2 and Section 5.3.1. Machine Learning can also be used to perform *fine-grained* path selection for each flow (e.g., NGR (XIAO et al., 2020)). However, querying a DNN to select a path for each flow poses severe scalability challenges. Additionally, reacting to transient congestion would require querying the DNN at a packet or flowlet granularity, further limiting scalability.

As an alternative, researchers have explored utilizing Machine Learning to generate a *strategy* for selecting paths. For instance, it is possible to use Machine Learning to generate distributed routing protocols (e.g., GQNN (GEYER; CARLE, 2018)), or to select a routing strategy (e.g., shortest-path or least-utilized) that will maximize a desired metric (e.g., RouteNet (RUSEK et al., 2019), RBB (RUSEK et al., 2022)). However, these techniques generate static routing strategies, not being able to properly utilize multi-path infrastructures.

5.4 Emerging categories of Load Balancers

This section covers load balancers that do not fit in any of the previous categories. Table 5.4 presents two emerging types of load balancers: *end-host* and *hybrid* load bal-

Table 5.4 – Emerging categories of load balancers.

Decision Plane	Related Work	Benefits	Limitations
End-Host	Hermes (ZHANG et al., 2017) PLB (QURESHI et al., 2022)	End-host decision-making is highly scalable and generally responsive.	Requires modifying end-hosts, severely limiting deployability. Heuristic-based approach can lead to suboptimal path selection.
Hybrid	Pizzutti and Schaeffer-Filho (2019) CONTRA (HSU et al., 2020a)	Combines data plane responsiveness with control plane visibility and path generation.	Heuristic-based path generation can lead to suboptimal network utilization.

Source: The Authors.

ancers. Considering every work in this section employs *heuristics* to generate paths and *fine-grained* path selection, we did not include these columns in the table. Further, the load balancing systems covered here can support any type of topology.

End-host load balancers are highly scalable and can quickly react to congestion along paths (e.g., Hermes (ZHANG et al., 2017), PLB (QURESHI et al., 2022)). However, this requires modifying end-hosts, which limits deployability to specific cases, such as datacenters or cloud environments. Further, existing end-host load balancers are based on simple *heuristics*, which can lead to suboptimal network utilization.

Hybrid load balancers combine reactive data plane processing with controller-based path generation. These systems can be highly scalable due to collaboration between the data and control planes. Additionally, hybrid load balancers can quickly detect and react to transient congestion in the data plane, while also providing efficient network utilization with *fine-grained* path selection in the control plane (e.g., Pizzutti and Schaeffer-Filho (2019), CONTRA (HSU et al., 2020a)). However, we believe existing work can be improved upon, as current strategies are limited to *heuristic*-based path generation and selection.

5.5 Discussion

This chapter reviewed data plane load balancers (Section 5.2) and control plane load balancers (Section 5.3). These two classes of load balancers have opposing characteristics, usually trading off scalability and reactiveness for more intelligent decision-making. Alternatively, end-host load balancing approaches have several benefits, but are limited to datacenter and cloud environments, where end-host modification is easily achieved.

Hybrid load balancers are an emerging type of load balancing systems that combine reactive data plane processing with intelligent decisions by the controller. Existing solutions (e.g., Pizzutti and Schaeffer-Filho (2019), CONTRA (HSU et al., 2020a)) al-

ready demonstrate the benefits of a hybrid approach, providing high scalability and efficient network utilization.

CrossBal improves upon related work by employing a Deep Reinforcement Learning agent responsible for selecting the best routes for each rerouted flow. Further, the fast decision loop in the data plane is capable of quickly reacting to transient congestion on installed paths. Finally, by focusing its efforts on elephant flows, CrossBal minimizes the number of flows to be actively rerouted. Only a few of the related work highlighted (e.g., Hedera (AL-FARES et al., 2010), Mahout (CURTIS; KIM; YALAGANDULA, 2011)) focus their efforts on elephant flows. As elephant flows are large and long-lasting flows that tend to have a high impact on the network, focusing on these flows can lead to efficient network utilization, while also improving control plane scalability, as there are significantly fewer flows to reroute.

6 CONCLUDING REMARKS

This chapter presents our concluding remarks. Section 6.1 provides a summary of our contributions, while Section 6.2 discusses possible future research directions.

6.1 Summary of Contributions

This work presented CrossBal, a hybrid load balancer that combines an intelligent decision-loop in the control plane with a reactive decision-loop in the programmable data plane. CrossBal utilizes Machine Learning models in the control plane to provide intelligent decision-making. The controller collaborates with programmable switches, responsible for a fast decision-loop that complements the more intelligent decision-loop.

CrossBal was designed after identifying a gap in existing network load balancers. In general, existing proposals present a trade-off between intelligent decision-making and scalability or ability to react to transient congestion. This can be attributed to most load balancers heavily relying on either the control or the data plane. While there have been a few attempts at designing hybrid load balancers, existing approaches employ simple heuristics for path selection. Further, few load balancers focus their efforts on elephant flows - high-throughput, long-lasting flows that can cause a large impact on the network.

CrossBal addresses the challenges in load balancing the network by employing both a logically-centralized controller and the data plane of programmable switches. The control plane implements a Deep Reinforcement Learning agent to periodically reroute high-impact elephant flows. The agent is fed with updated statistics of network link utilization polled from every switch, providing it with a global view of the network. This allows the agent to optimize the routing of high-impact flows in order to prevent network congestion.

Seeking to complement the intelligent decision-loop in the controller, CrossBal delegates the task of reacting to transient congestion to the programmable data plane. As programmable switches can perform simple logic and arithmetic operations at line-rate, CrossBal utilizes P4-enabled switches to implement a simpler, but fast, decision-loop. In particular, the data plane probes the quality of installed paths by measuring RTT and, upon detecting a significant increase in RTT, changes the active path for detected elephant flows. This mechanism allows programmable switches to quickly detect and react to transient congestion by observing the RTT of routes installed by the controller.

Finally, CrossBal employs cooperative cross-plane mechanisms in order to provide scalable and accurate elephant flow detection. This is achieved by first distributing the identification of possible elephant flows into two complementary mechanisms entirely in the data plane, before a final mechanism in the control plane. The first mechanism applies a *preliminary filtering* in order to rule out small and short-lived flows. After this first step reduces the number of flows to consider, the data plane applies a *refined detection* over the remaining flows. After identifying a flow as a likely elephant flow, the switch exports a set of features computed for this flow to the controller. As the control plane presents fewer programming and resource constraints, we implement a Machine Learning model in order to provide accurate identification of elephant flows.

Our evaluation showed that CrossBal outperforms ECMP at balancing different workloads over available network links. In particular, CrossBal consistently utilizes most available network links, while ensuring lower link utilization imbalance. We also compared the performance of agents trained with different reward functions, identifying the agent trained with a reward function based on the maximum link utilization as the best performing agent. Finally, we proposed and analyzed optimizations for the preliminary filtering mechanism used in identifying elephant flows in the data plane. The analysis showed how our optimization can reduce memory utilization, a critical but scarce resource in programmable switches. Additionally, it also showed how we can change optimization parameters in order to trade-off detection speed for filtering accuracy.

In summary, the main contributions of our work are the following:

- **Review of the state-of-the-art**. We performed a study of the literature in network load balancing, comparing the main characteristics of each work (Chapter 5);
- **Design and implementation of CrossBal**. We designed and implemented a hybrid load balancing system based on cross-plane collaboration and state-of-the-art Machine Learning models. Further, our system employs programmable data planes to detect and react to congestion in selected paths (Chapter 3);
- **Evaluation a Proof-of-Concept prototype**. We evaluated a prototype of CrossBal with BMv2 switches in an emulated environment with a realistic network topology and workloads. Our results showed that CrossBal outperforms ECMP at load balancing with different workloads. Further, we compared the design of DRL agents trained with different reward functions (Chapter 4);
- **Design of a Deep Reinforcement Learning agent.** We described our design of a Deep Reinforcement Learning agent capable of actively load balancing network

flows. We also highlighted the challenges in designing an agent and possible issues with alternative designs (Section 3.3).

Finally, we believe that CrossBal shows the benefits of a cross-plane load balancing approach based on Machine Learning models.

6.2 Future Work

CrossBal improves upon existing hybrid load balancers (previously discussed in Section 5.4) by applying Deep Reinforcement Learning to actively reroute high impact flows, while employing the programmable data plane to aid in the detection of elephant flows, identify congestion along installed paths, and switch the active path of detected elephant flows. However, there are alternative designs that can be explored, as well as possible improvements to our proof-of-concept prototype that was implemented and evaluated in Chapter 4. Next, we discuss possible directions for future work.

- **Exploration of alternative designs for preliminary filtering and refined detection**. The *preliminary filtering* module is based on the need to filter out a large number of small flows while keeping resource utilization at a minimum. The *refined detection* builds upon the previous module, being able to perform a more in-depth analysis over the remaining flows. While we found that simple thresholds produced satisfactory results for the *preliminary filtering*, an interesting direction for future work is to explore alternative mechanisms, such as sketches (SONG et al., 2020) or other compact data structures. Similarly, future work for the *refined detection* includes performing a more thorough feature selection (as mentioned in Section 3.2) and exploring alternative mechanisms for this module;
- **Alternative designs for the DRL agent**. Our design for the DRL agent was based on the review of the state-of-the-art in ML for networking, discussing the weaknesses and strengths of each design, and some experimentation (Section 3.3). An interesting direction for future work is to compare alternative designs for the agent. An example is to extend the state space to include an identifier for the current elephant flow being rerouted and the previous routes selected for it. Further, employing Graph Neural Networks (RUSEK et al., 2019) could improve the agent's ability to select routes;
- **In-depth evaluation and parameter analysis**. As mentioned in Section 4.2, our

emulated setup had some limitations. Additionally, CrossBal depends on a few important parameters (listed in Section 4.2.4). Therefore, a further evaluation in a distributed setup (emulated or on a testbed) with larger topologies and workloads is an important future work. Additionally, future work in evaluating CrossBal should include a more thorough analysis on the impact of system parameters.

REFERENCES

AL-FARES, M. et al. Hedera: Dynamic flow scheduling for data center networks. In: **Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation**. USA: USENIX Association, 2010. (NSDI'10), p. 19.

ALIZADEH, M. et al. Conga: Distributed congestion-aware load balancing for datacenters. In: **Proceedings of the 2014 ACM Conference on SIGCOMM**. New York, NY, USA: Association for Computing Machinery, 2014. (SIGCOMM '14), p. 503–514. ISBN 9781450328364. Available from Internet: <https://doi.org/10.1145/2619239.2626316>.

ALIZADEH, M. et al. Data center tcp (dctcp). In: **Proceedings of the ACM SIGCOMM 2010 Conference**. New York, NY, USA: Association for Computing Machinery, 2010. (SIGCOMM '10), p. 63–74. ISBN 9781450302012. Available from Internet: <https://doi.org/10.1145/1851182.1851192>.

BEMTEN, A. V. et al. Chameleon: Predictable latency and high utilization with queue-aware and adaptive source routing. In: **Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2020. p. 451–465. ISBN 9781450379489. Available from Internet: <https://doi.org/10.1145/3386367.3432879>.

BENET, C. H. et al. Mp-hula: Multipath transport aware load balancing using programmable data planes. In: **Proceedings of the 2018 Morning Workshop on In-Network Computing**. New York, NY, USA: Association for Computing Machinery, 2018. (NetCompute '18), p. 7–13. ISBN 9781450359085. Available from Internet: <https://doi.org/10.1145/3229591.3229596>.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2656877.2656890>.

BREIMAN, L. Random forests. **Machine Learning**, v. 45, n. 1, p. 5–32, Oct 2001. ISSN 1573-0565. Available from Internet: <https://doi.org/10.1023/A:1010933404324>.

COELHO, B.; SCHAEFFER-FILHO, A. Backorders: Using random forests to detect ddos attacks in programmable data planes. In: **Proceedings of the 5th International Workshop on P4 in Europe**. New York, NY, USA: Association for Computing Machinery, 2022. (EuroP4 '22), p. 1–7. ISBN 9781450399357. Available from Internet: <https://doi.org/10.1145/3565475.3569074>.

CURTIS, A. R.; KIM, W.; YALAGANDULA, P. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In: **2011 Proceedings IEEE INFOCOM**. Shanghai, China: IEEE Press, 2011. p. 1629–1637.

CURTIS, A. R. et al. Devoflow: Scaling flow management for high-performance networks. In: **Proceedings of the ACM SIGCOMM 2011 Conference**. New York, NY, USA: Association for Computing Machinery, 2011. (SIGCOMM '11), p. 254–265. ISBN 9781450307970. Available from Internet: <https://doi.org/10.1145/2018436.2018466>.

DURNER, R.; KELLERER, W. Network function offloading through classification of elephant flows. **IEEE Transactions on Network and Service Management**, v. 17, n. 2, p. 807–820, 2020.

ELIYAHU, T. et al. Verifying learning-augmented systems. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. New York, NY, USA: Association for Computing Machinery, 2021. (SIGCOMM '21), p. 305–318. ISBN 9781450383837. Available from Internet: <https://doi.org/10.1145/3452296.3472936>.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: An intellectual history of programmable networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2602204.2602219>.

FOUNDATION, L. **Data Plane Development Kit (DPDK)**. 2015. Available from Internet: <http://www.dpdk.org>.

GAVRILUŢ, V.; PRUSKI, A.; BERGER, M. S. Constructive or optimized: An overview of strategies to design networks for time-critical applications. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 55, n. 3, feb 2022. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3501294>.

GENG, Y. et al. Juggler: A practical reordering resilient network stack for datacenters. In: **Proceedings of the Eleventh European Conference on Computer Systems**. New York, NY, USA: Association for Computing Machinery, 2016. (EuroSys '16). ISBN 9781450342407. Available from Internet: <https://doi.org/10.1145/2901318.2901334>.

GEYER, F.; CARLE, G. Learning and generating distributed routing protocols using graph-based deep learning. In: **Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks**. New York, NY, USA: Association for Computing Machinery, 2018. (Big-DAMA '18), p. 40–45. ISBN 9781450359047. Available from Internet: <https://doi.org/10.1145/3229607.3229610>.

GLOROT, X.; BORDES, A.; BENGIO, Y. Deep sparse rectifier neural networks. In: . [S.l.: s.n.], 2010. v. 15.

GREENBERG, A. et al. Vl2: A scalable and flexible data center network. In: **Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2009. (SIGCOMM '09), p. 51–62. ISBN 9781605585949. Available from Internet: <https://doi.org/10.1145/1592568.1592576>.

GUO, L.; MATTA, I. The war between mice and elephants. In: **Proceedings Ninth International Conference on Network Protocols. ICNP 2001**. [S.l.: s.n.], 2001. p. 180–188.

GYM. 2023. Available from Internet: <https://www.gymlibrary.dev/>. Accessed in: 2023-05-30.

HO, T. K. Random decision forests. In: **Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1**. USA: IEEE Computer Society, 1995. (ICDAR '95), p. 278. ISBN 0818671289.

HØILAND-JØRGENSEN, T. et al. The express data path: Fast programmable packet processing in the operating system kernel. In: **Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2018. (CoNEXT '18), p. 54–66. ISBN 9781450360807. Available from Internet: <https://doi.org/10.1145/3281411.3281443>.

HONG, C.-Y. et al. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2018. (SIGCOMM '18), p. 74–87. ISBN 9781450355674. Available from Internet: <https://doi.org/10.1145/3230543.3230545>.

HSU, K.-F. et al. Contra: A programmable system for performance-aware routing. In: **17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)**. Santa Clara, CA: USENIX Association, 2020. p. 701–721. ISBN 978-1-939133-13-7. Available from Internet: <https://www.usenix.org/conference/nsdi20/presentation/hsu>.

HSU, K.-F. et al. Adaptive weighted traffic splitting in programmable data planes. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2020. (SOSR '20), p. 103–109. ISBN 9781450371018. Available from Internet: <https://doi.org/10.1145/3373360.3380841>.

JAIN, S. et al. B4: Experience with a globally-deployed software defined wan. In: **Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM**. New York, NY, USA: Association for Computing Machinery, 2013. (SIGCOMM '13), p. 3–14. ISBN 9781450320566. Available from Internet: <https://doi.org/10.1145/2486001.2486019>.

JURKIEWICZ, P. Boundaries of flow table usage reduction algorithms based on elephant flow detection. In: **2021 IFIP Networking Conference (IFIP Networking)**. [S.l.: s.n.], 2021. p. 1–9.

JURKIEWICZ, P. Boundaries of flow table usage reduction algorithms based on elephant flow detection. In: **2021 IFIP Networking Conference (IFIP Networking)**. Espoo and Helsinki, Finland: IEEE Press, 2021. p. 1–9.

KAMIYAMA, N. et al. Impact of topology on parallel video streaming. In: **2010 IEEE Network Operations and Management Symposium - NOMS 2010**. [S.l.: s.n.], 2010. p. 607–614.

KATTA, N. et al. Hula: Scalable load balancing using programmable data planes. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2016. (SOSR '16). ISBN 9781450342117. Available from Internet: <https://doi.org/10.1145/2890955.2890968>.

Kreutz, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, 2015.

KRISHNASWAMY, U. et al. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In: **19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)**. Renton, WA: USENIX Association, 2022. p. 325–338.

ISBN 978-1-939133-27-4. Available from Internet: <https://www.usenix.org/conference/nsdi22/presentation/krishnaswamy>.

LE, V. A. et al. Multi-time-step segment routing based traffic engineering leveraging traffic prediction. In: **2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)**. [S.l.: s.n.], 2021. p. 125–133.

LI, Q. et al. Data-driven routing optimization based on programmable data plane. In: **2020 29th International Conference on Computer Communications and Networks (ICCCN)**. Honolulu, HI, USA: IEEE Press, 2020. p. 1–9.

LIATIFIS, A. et al. Advancing sdn from openflow to p4: A survey. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 55, n. 9, jan 2023. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3556973>.

LIU, H. et al. Finding top-k shortest paths with diversity. **IEEE Transactions on Knowledge and Data Engineering**, v. 30, n. 3, p. 488–502, 2018.

LIU, Y. et al. Rslb: Robust and scalable load balancing in software-defined data center networks. **IEEE Transactions on Network and Service Management**, v. 19, n. 4, p. 4706–4720, 2022.

LIU, Z. et al. Burstbalancer: Do less, better balance for large-scale data center traffic. In: **2022 IEEE 30th International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2022. p. 1–13.

LUIZELLI, M. C. et al. How physical network topologies affect virtual network embedding quality: A characterization study based on isp and datacenter networks. **Journal of Network and Computer Applications**, v. 70, 05 2016.

LUONG, N. C. et al. Applications of deep reinforcement learning in communications and networking: A survey. **IEEE Communications Surveys & Tutorials**, v. 21, n. 4, p. 3133–3174, 2019.

MAGNOUCHE, Y. et al. Distributed utility maximization from the edge in ip networks. In: **2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)**. [S.l.: s.n.], 2021. p. 224–232.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/1355734.1355746>.

MNIH, V. et al. Human-level control through deep reinforcement learning. **Nature**, v. 518, p. 529–533, 2015.

NOORMOHAMMADPOUR, M.; RAGHAVENDRA, C. S. Datacenter traffic control: Understanding techniques and tradeoffs. **IEEE Communications Surveys & Tutorials**, v. 20, n. 2, p. 1492–1525, 2018.

NUMPY. 2023. Available from Internet: <https://numpy.org/>. Accessed in: 2023-05-30.

OPEN NETWORKING FOUNDATION. **OpenFlow Switch Specification**. [S.l.], 2009. Version 1.0.0. Available from Internet: <https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf>.

OPEN NETWORKING FOUNDATION. **OpenFlow Switch Specification**. [S.l.], 2015. Version 1.5.1. Available from Internet: <https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>.

PARHAM, M. et al. Traffic engineering with joint link weight and segment optimization. In: **Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2021. (CoNEXT '21), p. 313–327. ISBN 9781450390989. Available from Internet: <https://doi.org/10.1145/3485983.3494846>.

PARIZOTTO, R. et al. Offloading machine learning to programmable data planes: A systematic survey. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, jun 2023. ISSN 0360-0300. Just Accepted. Available from Internet: <https://doi.org/10.1145/3605153>.

PEIXOTO, T. **graph-tool**. 2023. Available from Internet: <https://graph-tool.skewed.de/>. Accessed in: 2023-05-30.

PERRY, Y. et al. DOTE: Rethinking (predictive) WAN traffic engineering. In: **20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)**. Boston, MA: USENIX Association, 2023. p. 1557–1581. ISBN 978-1-939133-33-5. Available from Internet: <https://www.usenix.org/conference/nsdi23/presentation/perry>.

PIZZUTTI, M.; SCHAEFFER-FILHO, A. E. Adaptive multipath routing based on hybrid data and control plane operation. In: **IEEE INFOCOM 2019 - IEEE Conference on Computer Communications**. Paris, France: IEEE Press, 2019. p. 730–738. Available from Internet: <https://doi.org/10.1109/INFOCOM.2019.8737398>.

POLIKAR, R. Polikar, r.: Ensemble based systems in decision making. ieee circuit syst. mag. 6, 21-45. **Circuits and Systems Magazine, IEEE**, v. 6, p. 21 – 45, 10 2006.

PYTORCH. 2023. Available from Internet: <https://pytorch.org/>. Accessed in: 2023-05-30.

QUINLAN, J. R. Simplifying decision trees. **Int. J. Man-Mach. Stud.**, Academic Press Ltd., GBR, v. 27, n. 3, p. 221–234, sep. 1987. ISSN 0020-7373. Available from Internet: <https://doi.org/10.1016/S0020-7373(87)80053-6>.

QUINLAN, J. R. **C4.5: Programs for Machine Learning**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1558602380.

QURESHI, M. A. et al. Plb: Congestion signals are simple and effective for network load balancing. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. New York, NY, USA: Association for Computing Machinery, 2022. (SIGCOMM '22), p. 207–218. ISBN 9781450394208. Available from Internet: <https://doi.org/10.1145/3544216.3544226>.

REDA, W. et al. Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 50, n. 2, p. 11–23, may 2020. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/3402413.3402416>.

RESTUCCIA, F.; MELODIA, T. Deepwierl: Bringing deep reinforcement learning to the internet of self-adaptive things. In: **IEEE INFOCOM 2020 - IEEE Conference on Computer Communications**. Toronto, ON, Canada: IEEE Press, 2020. p. 844–853.

RUSEK, K. et al. **Fast Traffic Engineering by Gradient Descent with Learned Differentiable Routing**. 2022.

RUSEK, K. et al. Unveiling the potential of graph neural networks for network modeling and optimization in sdn. In: **Proceedings of the 2019 ACM Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2019. (SOSR '19), p. 140–151. ISBN 9781450367103. Available from Internet: <https://doi.org/10.1145/3314148.3314357>.

SAPIO, A. et al. In-network computation is a dumb idea whose time has come. In: **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2017. (HotNets-XVI), p. 150–156. ISBN 9781450355698. Available from Internet: <https://doi.org/10.1145/3152434.3152461>.

SCAPY. 2023. Available from Internet: <https://scapy.net/>. Accessed in: 2023-05-30.

SILVA, M. V. B. da et al. Ideafix: Identifying elephant flows in p4-based ixp networks. In: **2018 IEEE Global Communications Conference (GLOBECOM)**. [S.l.: s.n.], 2018. p. 1–6.

SILVA, M. V. Brito da; SCHAEFFER-FILHO, A. E.; GRANVILLE, L. Z. Hashcuckoo: Predicting elephant flows using meta-heuristics in programmable data planes. In: **GLOBECOM 2022 - 2022 IEEE Global Communications Conference**. [S.l.: s.n.], 2022. p. 6337–6342.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. **Nature**, v. 529, p. 484–489, 01 2016.

SIVARAMAN, V. et al. Heavy-hitter detection entirely in the data plane. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2017. (SOSR '17), p. 164–176. ISBN 9781450349475. Available from Internet: <https://doi.org/10.1145/3050220.3063772>.

SONG, C. H. et al. Fcm-sketch: Generic network measurements with data plane support. In: **Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2020. (CoNEXT '20), p. 78–92. ISBN 9781450379489. Available from Internet: <https://doi.org/10.1145/3386367.3432729>.

SUTTON, R.; BARTO, A. **Reinforcement Learning, second edition: An Introduction**. MIT Press, 2018. (Adaptive Computation and Machine Learning series). ISBN 9780262039246. Available from Internet: <https://books.google.com/books?id=6DKPtQEACAAJ>.

TALBI, E.-G. Automated design of deep neural networks: A survey and unified taxonomy. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 2, mar 2021. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3439730>.

VALADARSKY, A. et al. Learning to route. In: **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2017. (HotNets-XVI), p. 185–191. ISBN 9781450355698. Available from Internet: <https://doi.org/10.1145/3152434.3152441>.

VANINI, E. et al. Let it flow: Resilient asymmetric load balancing with flowlet switching. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 407–420. ISBN 978-1-931971-37-9. Available from Internet: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini>.

XIAO, S. et al. Neural packet routing. In: **Proceedings of the Workshop on Network Meets AI & ML**. New York, NY, USA: Association for Computing Machinery, 2020. (NetAI '20), p. 28–34. ISBN 9781450380430. Available from Internet: <https://doi.org/10.1145/3405671.3405813>.

XU, Z. et al. Experience-driven networking: A deep reinforcement learning based approach. In: **IEEE INFOCOM 2018 - IEEE Conference on Computer Communications**. Honolulu, HI, USA: IEEE Press, 2018. p. 1871–1879. Available from Internet: <https://doi.org/10.1109/INFOCOM.2018.8485853>.

ZAICU, N. F. et al. Helix: Traffic engineering for multi-controller sdn. In: ____. **Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)**. New York, NY, USA: Association for Computing Machinery, 2021. p. 80–87. ISBN 9781450390842. Available from Internet: <https://doi.org/10.1145/3482898.3483354>.

ZHAN, Y.; ZHANG, J. An incentive mechanism design for efficient edge learning by deep reinforcement learning approach. In: **IEEE INFOCOM 2020 - IEEE Conference on Computer Communications**. Toronto, ON, Canada: IEEE Press, 2020. p. 2489–2498.

ZHANG, H. et al. Resilient datacenter load balancing in the wild. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 253–266. ISBN 9781450346535. Available from Internet: <https://doi.org/10.1145/3098822.3098841>.

ZHANG, J. et al. Smartentry: Mitigating routing update overhead with reinforcement learning for traffic engineering. In: **Proceedings of the Workshop on Network Meets AI & ML**. New York, NY, USA: Association for Computing Machinery, 2020. (NetAI '20), p. 1–7. ISBN 9781450380430. Available from Internet: <https://doi.org/10.1145/3405671.3405809>.

ZHANG, J. et al. Load balancing in data center networks: A survey. **IEEE Communications Surveys & Tutorials**, v. 20, n. 3, p. 2324–2352, 2018.

ZHU, L. et al. Sdn controllers: A comprehensive analysis and performance evaluation study. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 6, dec 2020. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3421764>.

APPENDIX A – RESUMO EXPANDIDO

Este trabalho apresenta o CrossBal, um balanceador de carga híbrido que combina uma lógica de decisão inteligente no plano de controle com uma lógica de decisão reativa no plano de dados programável. O CrossBal utiliza modelos de Aprendizado de Máquina (*Machine Learning*) no plano de controle, responsável por escolher rotas de maneira inteligente. O controlador colabora com *switches* programáveis, responsáveis por um ciclo de decisão rápido que complementa o ciclo de decisão mais inteligente do controlador.

Com o crescimento acelerado da *Internet* nas últimas décadas, as infra-estruturas de rede têm sido continuamente expandidas. As redes de computadores muitas vezes implementam caminhos redundantes entre os pontos finais, a fim de aumentar a confiabilidade e atender às demandas dos usuários e dos serviços.

Contudo, para utilizar eficientemente a infra-estrutura disponível, o tráfego de entrada deve ser adequadamente dividido entre os links de saída redundantes, evitando links sobrecarregados enquanto outros permanecem subutilizados. Neste cenário de múltiplos caminhos, as estratégias clássicas de roteamento de caminho mais curto são incapazes de fornecer uma utilização eficiente da rede. Equal-Cost Multi-Path (ECMP) é uma técnica de balanceamento de carga amplamente utilizada devido à sua simplicidade, estando disponível em switches comerciais. Entretanto, o ECMP pode apresentar sérios problemas de desempenho, sendo incapaz de fornecer uma divisão eficiente do tráfego da rede.

O CrossBal foi projetado após identificar uma lacuna nos balanceadores de carga de rede existentes. Em geral, as propostas existentes apresentam uma escolha entre decisões inteligentes e a eficiência do sistema, como a escalabilidade e a capacidade de reagir a congestionamentos de curta duração. Isso pode ser atribuído ao fato da maioria dos balanceadores de carga focarem no plano de controle ou no plano de dados. Embora alguns trabalhos proponham balanceadores de carga híbridos, as abordagens existentes empregam heurísticas simples para seleção de caminhos. Além disso, poucos balanceadores de carga concentram seus esforços em fluxos elefantes – fluxos de alto rendimento e de longa duração que podem causar um grande impacto na rede.

O CrossBal aborda os desafios do balanceamento de carga da rede empregando um controlador logicamente centralizado e o plano de dados de switches programáveis. O plano de controle implementa um agente de Aprendizado por Reforço Profundo (*Deep Reinforcement Learning*) para periodicamente rerotear fluxos elefantes de alto impacto. O agente é alimentado com estatísticas atualizadas de utilização dos *links* de rede, pro-

porcionando uma visão global da rede. Isto permite ao agente otimizar o roteamento de fluxos de alto impacto para evitar congestionamentos na rede.

A fim de complementar o ciclo de decisão inteligente no controlador, o CrossBal delega a tarefa de reagir ao congestionamento transitório ao plano de dados programável. Como os *switches* programáveis podem realizar operações lógicas e aritméticas simples em taxa de linha, o CrossBal utiliza *switches* P4 para implementar uma lógica de decisão mais simples, porém rápida. Em particular, o plano de dados estima a qualidade dos caminhos instalados medindo o RTT e, ao detectar um aumento significativo no RTT, altera o caminho ativo para os fluxos elefantes detectados. Este mecanismo permite que os *switches* programáveis detectem e reajam rapidamente a congestionamentos transitórios, observando o RTT das rotas instaladas pelo controlador.

Finalmente, o CrossBal emprega mecanismos cooperativos para fornecer detecção eficiente e precisa de fluxos elefantes. Isto é feito primeiro distribuindo a identificação de possíveis fluxos elefantes em dois mecanismos complementares inteiramente no plano de dados, além de um mecanismo final no plano de controle. O primeiro mecanismo aplica uma filtragem preliminar para excluir fluxos pequenos e de curta duração. Após esta primeira etapa reduzir o número de fluxos a considerar, o plano de dados aplica uma detecção refinada sobre os fluxos restantes. Depois de identificar um fluxo como provável fluxo elefante, o *switch* exporta um conjunto de características (*features*) desse fluxo para o controlador. Como o plano de controle apresenta menos restrições do modelo de programação e recursos disponíveis, implementamos um modelo de Aprendizado de Máquina (*Machine Learning*) para fornecer uma identificação precisa dos fluxos elefantes.

Nossa avaliação mostrou que o CrossBal supera o ECMP no equilíbrio de diferentes cargas de trabalho nos *links* de rede disponíveis. Em particular, o CrossBal utiliza consistentemente a maioria dos *links* de rede disponíveis, garantindo ao mesmo tempo um menor desequilíbrio na utilização dos *links*. Também comparamos o desempenho de agentes treinados com diferentes funções de recompensa, identificando o agente treinado com uma função de recompensa baseada na utilização máxima do link (*maximum link utilization*) como o agente com melhor desempenho. Por fim, descrevemos e analisamos otimizações para o mecanismo de filtragem preliminar utilizado na identificação de fluxos elefantes no plano de dados. A análise mostrou como nossa otimização pode reduzir a utilização de memória, um recurso crítico, mas escasso em switches programáveis. Além disso, nossa avaliação também mostrou como podemos alterar os parâmetros de otimização para fazer ajustes, trocando a velocidade de detecção pela precisão da filtragem.

Em resumo, as principais contribuições do nosso trabalho são as seguintes:

- Revisão do estado-da-arte. Realizamos um estudo da literatura em balanceamento de carga de redes, comparando as principais características de cada trabalho;
- Projeto e implementação do CrossBal. Projetamos e implementamos um sistema de balanceamento de carga híbrido baseado em colaboração entre planos e modelos recentes de aprendizado de máquina. Além disso, nosso sistema emprega planos de dados programáveis para detectar e reagir ao congestionamento em caminhos selecionados;
- Avaliação de um protótipo de prova-de-conceito. Avaliamos um protótipo do CrossBal com *switches* BMv2 em um ambiente emulado com topologia de rede e cargas de trabalho realistas. Nossos resultados mostraram que o CrossBal supera o ECMP no balanceamento de carga com diferentes cargas de trabalho. Além disso, comparamos o design de agentes de Aprendizado por Reforço Profundo (*Deep Reinforcement Learning*) treinados com diferentes funções de recompensa;
- Projeto de um agente de Aprendizado por Reforço Profundo (*Deep Reinforcement Learning*). Descrevemos nosso projeto de um agente de Aprendizado por Reforço Profundo (*Deep Reinforcement Learning*) capaz de balancear os fluxos de rede. Também destacamos os desafios na concepção de um agente e possíveis problemas com propostas alternativas.

Por fim, acreditamos que o CrossBal mostra os benefícios de uma abordagem híbrida de balanceamento de carga baseado em modelos de Aprendizado de Máquina (*Machine Learning*).

APPENDIX B – ACCEPTED PAPER – CNSM 2023

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

- **Title:** CrossBal: Data and Control Plane Cooperation for Efficient and Scalable Network Load Balancing
- **Conference:** CNSM 2023 - 19th International Conference on Network and Service Management
- **Type:** Main Track (Full Paper)
- **Qualis:** A4
- **Date:** 30 October - 2 November 2023
- **Location:** Niagara Falls, Canada

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

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

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

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

I. Introduction

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

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

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

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

In summary, this work presents the following contributions:

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

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

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

II. Background and Motivation

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

A. Deep Reinforcement Learning

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

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

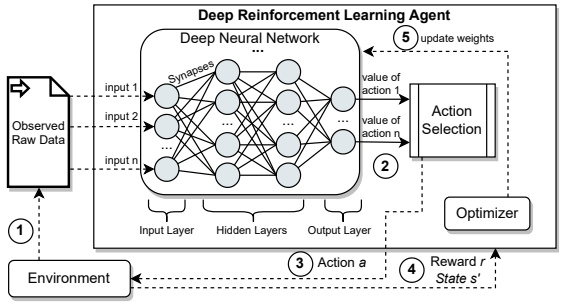


Fig. 1: Steps of an iteration of a Deep Learning Agent

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

B. Programmable Data Planes

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

III. CrossBal: Cross-Plane Load Balancing

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

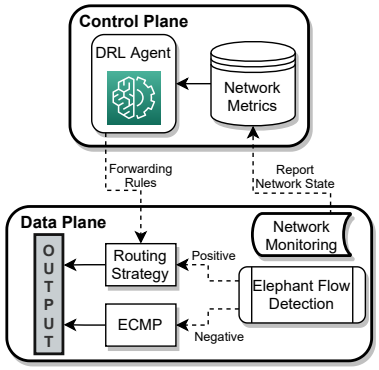


Fig. 2: CrossBal employs cross-plane collaboration.

A. Approach Overview

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

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

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

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

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

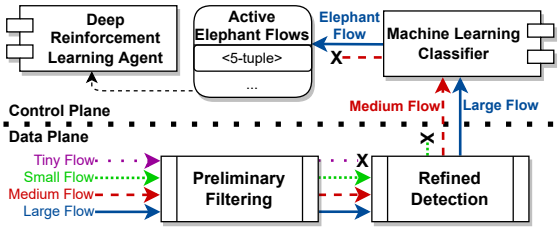


Fig. 3: Overview of the cross-plane elephant flow detection.

B. Identifying Elephant Flows at Line-Rate

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

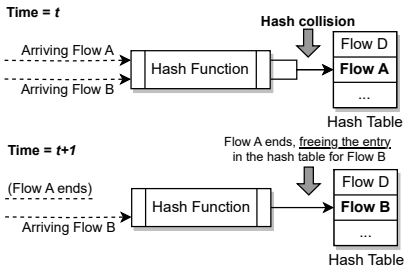
CrossBal *decomposes the detection* of elephant flows into three levels of complementary mechanisms, balancing the tradeoffs between fast and lightweight detection in the data plane with more accurate and heavyweight detection in the control plane (Figure 3):

- ***Preliminary filtering:*** the data plane implements a threshold-based detection that tracks the number of bytes, number of packets, and duration for each active flowlet[2]. The main aspect of this mechanism is that it must handle a large number of flows, thus limiting the amount of processing and storage available for each flow. Therefore, as shown in Figure 3, this step acts as an early filtering of low-throughput and short flows in order to save hardware resources. A further optimization is proposed and evaluated (§V-D), where only packets larger than a threshold are accounted for. By utilizing this strategy, the number of packets of a flowlet can be seen as a lower bound of the number of bytes transmitted by the flowlet. This can lower the number of bits utilized to store this information, saving precious on-board memory. The same reasoning can be applied to track the number of flowlet timeouts of a flow rather than the entire duration.

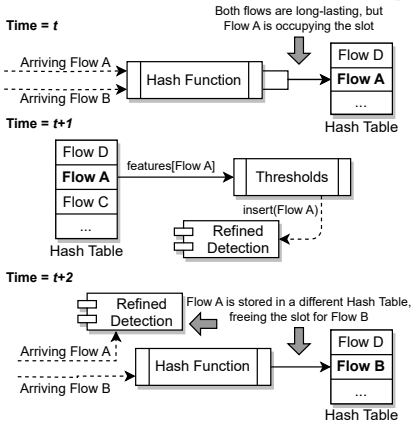
[2]Flowlets are bursts of packets of a flow separated by an idle interval.

- ***Refined detection:*** While the threshold-based mechanism mentioned above can exclude a large number of small and short flows, it may lead to a high number of false positives if used by itself. To address this, CrossBal employs further mechanisms to detect elephant flows. The intuition is that because the preliminary threshold-based filtering already excluded a large number of unimportant flows, it is now possible to implement a slightly *refined detection* mechanism over the remaining flows of interest, as shown in Figure 3. In particular, CrossBal implements a classification tree in the form of *if-else* statements in the programmable data plane. As there is a smaller number of flows to consider, it is possible to dedicate slightly more on-board memory to track *features* of each flowlet. In our PoC prototype, we track simple statistics of the inter-arrival-time and packet size of each flowlet. We leave a more thorough feature selection as future work.
- ***Cross-plane detection:*** Although CrossBal implements multiple mechanisms for the identification of elephant flows directly in the data plane, ensuring line-rate processing requires sacrificing accuracy for efficiency. In order to provide a more accurate detection of elephant flows, CrossBal employs cross-plane collaboration, as shown in Figure 3. This builds upon the *preliminary filtering* (which reduces the amount of flows of interest) and upon the *refined detection* (which tracks additional features for relevant flows). Therefore, CrossBal implements a classifier in the control plane that receives the information tracked by the data plane. As the controller provides a more flexible programming model, and considering the features extracted by the data plane, we implement a Random Forest in the control plane, providing higher accuracy than a single classification tree.

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



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



(b) Hash collision between two elephant flows.

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

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

C. Deep Reinforcement Learning Agent

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

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

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

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

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

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

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

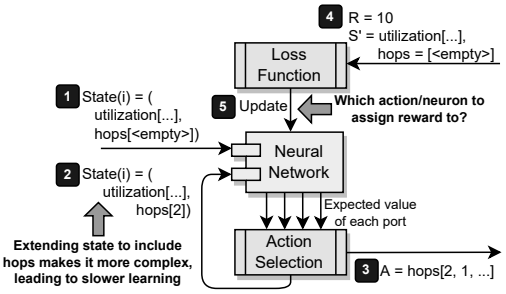


Fig. 5: Mapping actions to hops leads to reward assignment issues.

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

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

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

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

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

D. Reacting to Short-Lived Network Congestion

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

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

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

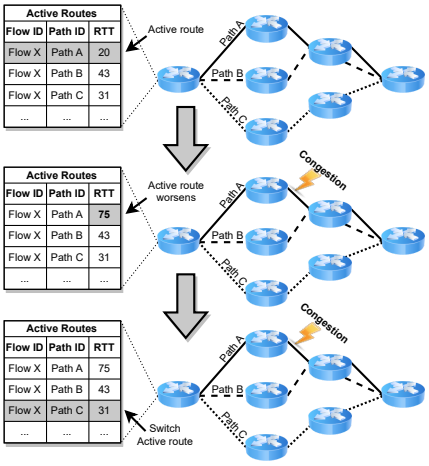


Fig. 6: Mechanism for switching active paths

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

IV. Architecture

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

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

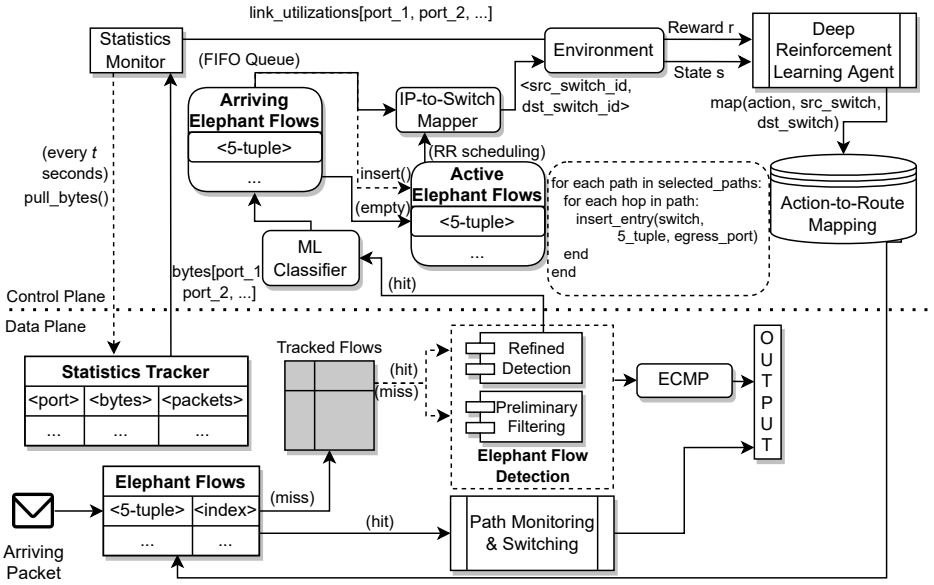


Fig. 7: Architectural implementation of CrossBal.

Algorithm 1: Data plane packet processing pipeline

Data: $pkt \leftarrow$ Packet In
Data: $flow \leftarrow pkt.5\_tuple$
1 **if** $flow$ *is in* $elephant\_flows$ **then**
2 $rtt\_diff \leftarrow active\_route.curr\_rtt - active\_route.prev\_rtt$;
3 **if** $rtt\_diff \geq$ *RTT Threshold* **then**
4 $active\_route[flow] \leftarrow min(installed\_routes[flow])$;
5 $time\_since\_probing \leftarrow curr\_time - last\_probe[flow]$;
6 **if** $time\_since\_probing \geq$ *Probing Interval* **then**
7 $create\_probes(installed\_routes[flow])$;
8 $egress\_port \leftarrow active\_route[flow].egress\_port$;
9 **else**
10 **if** $flow$ *is in* $refined\_detection.tracked\_flows$ **then**
11 $features[flow] \leftarrow compute\_features(flow)$;
// `If` statements are automatically generated
12 **if** $feature\_1[flow] \geq$ *Feature 1 Threshold* ***AND*** $feature\_3[flow] <$ *Feature 3 Threshold* **then**
13 $notify\_controller(flow, features[flow])$;
14 **else**
15 **if** *Bytes Optimization is enabled* **then**
16 **if** $pkt.length >$ *Length Threshold* **then**
17 $packets[flow] \leftarrow packets[flow] + 1$;
18 $bytes[flow] \leftarrow packets[flow] *$ Length Threshold;
19 **else**
20 $bytes[flow] \leftarrow bytes[flow] + pkt.length$;
21 $flow\_duration \leftarrow curr\_time - flow\_start[flow]$;
22 **if** $bytes[flow] \geq$ *Bytes Threshold* ***AND*** $flow\_duration \geq$ *Duration Threshold* **then**
23 $refined\_detection.track(flow)$;
24 $egress\_port \leftarrow ecmp(pkt.5\_tuple)$;

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

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

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

V. Evaluation

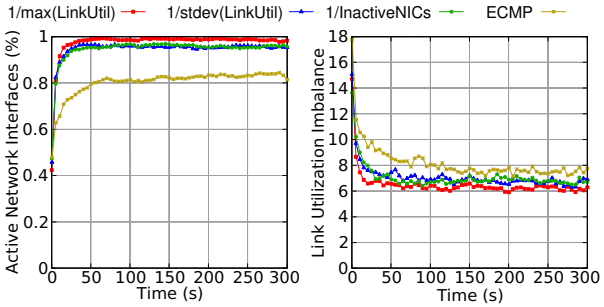
We implemented and evaluated a prototype of CrossBal in order to validate its design. Our experiments aimed to evaluate how well CrossBal could perform load balancing, as well as understand how key parameters might impact its performance.

A. Prototype

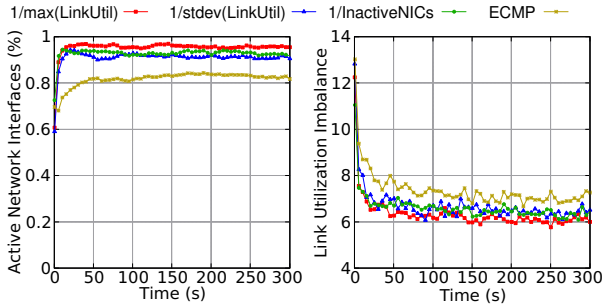
The prototype includes data plane software written in P4 and control plane software written in Python 3. The P4 source-code

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

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



(a) Link analysis for workload 1.



(b) Link analysis for workload 2.

Fig. 8: Link Utilization Imbalance and Active NICs.

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

B. Methodology

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

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

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

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

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

TABLE I: Workloads used in our evaluation.

C. Link Utilization Analysis

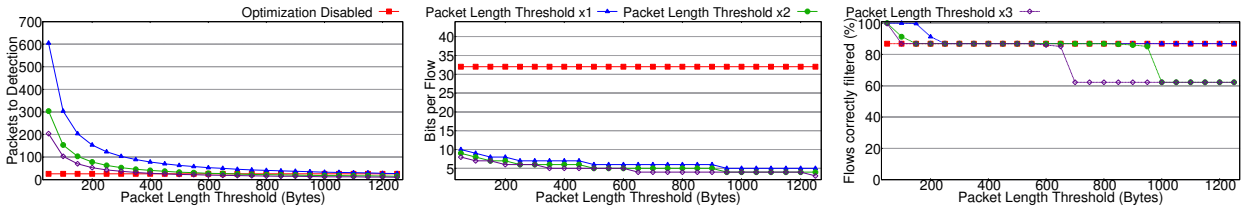
We implemented and compared three different reward functions for the DRL agent (§III-C): (A) $\frac{1}{max(link\_util)}$, (B) $\frac{1}{stdev(link\_util)}$, and (C) $\frac{1}{inactive\_nics}$, where $link\_util$ is an array with the utilization of every link in the network and $inactive\_nics$ is the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

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

D. Elephant flow detection optimizations

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

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



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

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

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

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

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

VI. Related Work

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

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

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

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

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

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

VII. Conclusion

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

TABLE II: Comparison of related work

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

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

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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