The OpenFlow rules placement problem : a black box approach

Xuan-Nam Nguyen

UNIVERSITÉ NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour l'obtention du grade de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis
Mention : Informatique

Présentée et soutenue par
Xuan Nam NGUYEN

The OpenFlow Rules Placement Problem: a Black Box approach

Thèse dirigée par Thierry TURLETTI et Walid DABBOUS

Inria, France

Soutenue le 22 avril 2016

Jury :

Rapporteurs :	Giuseppe BIANCHI	University of Rome Tor Vergata
	Steve UHLIG	Queen Mary University of London
Directeurs :	Thierry TURLETTI	Inria
	Walid DABBOUS	Inria
Examinateurs :	Mathieu BOUET	Thales
	Laurent VANBEVER	ETH Zurich
Président :	Guillaume URVOY-KELLER	University of Nice Sophia Antipolis

Acknowledgements

This thesis summarizes my research results during my Ph.D. study at the research team DIANA, Inria, France from 2012 to 2016.

Firstly, I would like to express my sincere gratitude to my advisor Dr. Thierry TURLETTI and Dr. Walid DABBOUS for their belief, patience, motivation, vision, and immense knowledge. Their guidance helped me to pass difficult moments of the research. It is my great pleasure to be their student.

Besides my advisors, I would like to thank Dr. Damien SAUCEZ, Dr. Chadi BARAKAT, and Dr. Truong Khoa PHAN for their brilliant ideas, discussions, valuable comments. They look like mentors, friends than professors, and I learned a lot from them.

I thank my DIANA fellow labmates, my co-authors, my collagues at Aalto University, and my friends for the time we were working together, for exchanges of knowledge, skills and for unfogettable moments we had in four years.

I recognize that this research would not have been possible without the financial support from Inria, the Nice Sophia Antipolis University, and I would like to express my gratitude to those agencies.

Last but not the least, I would like to thank my parents, my sisters, and my fiance Huong, for their sacrifice and unconditional support during these years. They have cherished with me every great moment and been beside me whenever I needed them.

The OpenFlow Rules Placement Problem: a Black Box approach

Abstract: The massive number of connected devices combined with an ever increasing volume of data traffic push network operators to their limit by limiting their profitability. Software-Defined Networking (SDN), which decouples network control logics from forwarding devices, has been proposed to tackle this problem. An important part of the SDN concepts is implemented by the OpenFlow protocol that abstracts network communications as flows and processes them using a prioritized list of matching-actions rules on the network forwarding elements. While the abstraction offered by OpenFlow allows to implement a large panel of applications, it raises the new problem of how to define the rules and where to place them in the network while respecting all technical and administrative requirements, which we refer as the OpenFlow Rules Placement Problem (ORPP).

In this thesis, we focus on the ORPP, and propose a black box abstraction that can hide the complexity of rules management. First, we formalize that problem, classify, and discuss existing solutions. We discover that most of the solutions enforce the routing policy when placing rules, which is not memory efficient in some cases. Second, by trading routing for better resource efficiency, we propose OFFICER and aOFFICER, two complementary rules placement frameworks that select and place OpenFlow rules satisfying policies and network constraints, while minimizing overheads. The main idea of OFFICER and aOFFICER is to install rules on efficient paths for important, large flows, and let other flows follow default, less appropriate paths. On one hand, OFFICER is designed based on optimization techniques to solve the offline ORPP, in which the set of flows is assumed known and stable in a period. On the other hand, aOFFICER uses adaptive control techniques to cope with the online ORPP, where the set of flows is unknown and varies over time. These proposals are evaluated and compared extensively to existing solutions in realistic scenarios. Finally, we study a use case of the black box abstraction, in which we target to improve the performance of content delivery services in cellular networks.

Keywords: Software-Defined Networking, OpenFlow, Rules Placement, Content Placement, Optimization

Contents

1 Introduction **1**
- 1.1 Problem Statement and Motivation . . . 1
- 1.2 Example Scenarios . . . 4
- 1.3 Research Methodology . . . 7
- 1.4 Thesis Outline . . . 7
- 1.5 Publications . . . 9

2 Preliminaries **11**
- 2.1 Linear Programming . . . 11
- 2.2 Greedy Algorithms . . . 12
- 2.3 Exponentially Weighted Moving Average Model . . . 13
- 2.4 Increase/Decrease Algorithms . . . 14

3 Literature Review **17**
- 3.1 Introduction . . . 17
- 3.2 OpenFlow Rules Placement Problem . . . 18
  - 3.2.1 Problem Formalization . . . 18
  - 3.2.2 Challenges . . . 20
- 3.3 Efficient Memory Management . . . 21
  - 3.3.1 Eviction . . . 22
  - 3.3.2 Compression . . . 24
  - 3.3.3 Split and Distribution . . . 28
- 3.4 Reducing Signaling Overhead . . . 32
  - 3.4.1 Reactive and Proactive Rules Placement . . . 33
  - 3.4.2 Delegating Functions to OpenFlow switches . . . 34
- 3.5 Conclusion . . . 35

4 Offline Rules Placement in OpenFlow Networks **37**
- 4.1 Introduction . . . 37
- 4.2 General Model to Allocate Rules in OpenFlow . . . 40
- 4.3 Rule Allocation Under Memory Constraints . . . 43
  - 4.3.1 Minimizing Memory Usage . . . 44
  - 4.3.2 Maximizing Traffic Satisfaction . . . 44
  - 4.3.3 Heuristic . . . 45
- 4.4 Evaluation . . . 48
  - 4.4.1 Methodology . . . 49
  - 4.4.2 Results . . . 51
- 4.5 Discussion . . . 53
  - 4.5.1 Routing Policy . . . 53
  - 4.5.2 Rule Aggregation . . . 53

4.5.3 Multipath . . . 53
4.5.4 Related Work . . . 54
4.6 Conclusion . . . 55

5 Adaptive Rules Placement in OpenFlow Networks 57
5.1 Introduction . . . 57
5.2 aOFFICER: Adaptive OpenFlow Rules Placement . . . 59
5.2.1 Objectives . . . 59
5.2.2 Design . . . 60
5.2.3 Adaptive Threshold . . . 61
5.3 Evaluation . . . 66
5.3.1 Setup . . . 66
5.3.2 Adaptive Threshold . . . 69
5.3.3 Adaptive Timeout and Deflection Technique . . . 73
5.4 Conclusion . . . 77

6 Use Case: Improving Content Delivery in LTEs 79
6.1 Introduction . . . 79
6.2 Background . . . 81
6.3 LTE In-network Caching Architecture . . . 81
6.3.1 Multi-level Caching Scheme . . . 82
6.3.2 Enabling Backhaul Caching with OpenFlow . . . 83
6.4 Content Allocation Model . . . 83
6.4.1 Content Allocation Problem Approximation . . . 86
6.5 Evaluation . . . 87
6.5.1 Simulation Setup . . . 87
6.5.2 Benefits of Caching at Different Levels . . . 89
6.5.3 Impact of Several Levels Caching . . . 92
6.5.4 Advantages of Using Opportunistic Caching for Networks with Loss . . . 93
6.6 Related Work . . . 94
6.7 Conclusion . . . 95

7 Conclusions and Future Work 97
7.1 Conclusions . . . 97
7.2 Future Work . . . 99
7.2.1 Robust and Fault Tolerant Rules Placement . . . 99
7.2.2 Impact of Default Devices . . . 99
7.2.3 Multilevel Table Rules Placement . . . 99
7.2.4 Network Function Virtualization . . . 100

Bibliography 101

Acronym

CDF Cumulative Distribution Function

PDF Probability Mass Function

IP Internet Protocol

ISP Internet Service Provider

ILP Integer Linear Programming

MILP Mixed Integer Linear Programing

MPLS Multi-protocol Label Switching

OSPF Open Shortest Path First

ECMP Equal Cost Multi-Path

QoS Quality of Service

SDN Software-Defined Networking

TCAM Ternary Content Addressable Memory

TE Traffic Engineering

WAN Wide Area Network

ORPP OpenFlow Rules Placement Problem

LTE Long-Term Evolution

CCN Content-Centric Networking

ICN Information-Centric Networking

ASIC Application-specific Integrated Circuit

LRU Least Recently Used

Glossary

Access control rule	Rule having actions field *drop/permit packets*.
Commodity switch	OpenFlow switch that stores rules in TCAM.
Default devices	Devices (e.g., software OpenFlow switches) that store rules in non-TCAM (e.g., RAM), used to process non matching packets.
Default path	Sequence of nodes from an ingress switch to the default devices, formed by default rules.
Default rule	Lowest priority rule, matching all the packets.
Elephant flow	Flow that sends many packets or bytes.
Endpoint Policy	Policy that defines the endpoints for each flow (e.g., egress links, gateways, firewalls).
Exact-matching rule	Rule that does not contain ternary elements ($*$) in its matching pattern.
Flow table hit	A flow is processed by a non-default rule.
Flow table miss	A flow is processed by the default rule.
Flow table	List of prioritized rules on the switch.
Flow	A sequence of packets that have common header fields (e.g., destination IP address).
Forwarding rule	Rule having the actions field *forwarding packets to an interface*.
Mouse flow	Flow that sends few packets or bytes.
Routing policy	Policy that specifies the path that the flow must follow.
Rule space	Set of all possible rules for selection.
Rules placement	A configuration that indicates which rules are placed on which switch.
Rule	An instruction for the OpenFlow switch specifying how to process the packets.
Wildcard rule	Rule that contains ternary elements ($*$) in its matching pattern.

Chapter 1

Introduction

Contents

1.1 Problem Statement and Motivation 1
1.2 Example Scenarios 4
1.3 Research Methodology 7
1.4 Thesis Outline 7
1.5 Publications 9

1.1 Problem Statement and Motivation

Nowadays, the Internet is an integral part of our modern life, and it has revolutionized the way we communicate. Due to technology advances, more people can access to the Internet using cheaper, more portable, more powerful devices (e.g., mobile phones, tablets, laptops). In recent years, the number of Internet-connected devices and the traffic volume have increased dramatically. Facebook, the most popular social network, has reached one billion users in a single day [Fac]. According to a Cisco's report [Cis15], the number of Internet connected devices was nearly two per capita in 2014, and will be three per capita in 2019. Furthermore, the annual global traffic has increased five times in the past five years (2009-2014), will surpass the zettabyte (10^{21} bytes) in 2016 and two zettabytes in 2019. Also, the video streaming traffic (e.g., Video-on-Demand, IPTV) accounts for 64% of all Internet traffic in 2014, and that portion will be 80% in 2019.

To keep pace with increasing demands, Internet Services Providers (ISP) often have to upgrade and reconfigure the network, e.g., buying and configuring new network devices. Normally, operators often have to transform high-level policies (e.g., the firewall policy, the routing policy) into low-level, vendor-specific configuration commands for each device, while manually adapting them to cope with network changes. However, this process is complicated, error-prone and time-consuming because of a large number of diverse network devices, such as switches, routers, middleboxes. Automatic reconfiguration does not exist in current networks [KREV+15]. The primary reason for this inconvenience is the vertical integrated, tightly coupled architecture of network devices and the proprietary software controlling them. With this architecture, the network devices are closed boxes, which are hard for operators to innovate. If operators want a new feature, they often have to wait until next

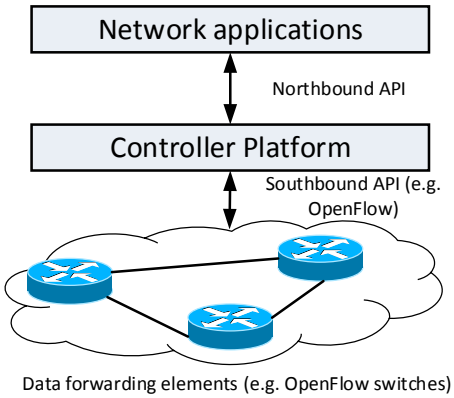


Figure 1.1: SDN architecture

device life cycles or firmware releases from vendors. This inconvenience may slow down the operators' development plans and incur high management costs.

To accelerate the innovation process, the operators need more flexibility to control, to customize network devices. To that goal, Software-Defined Networking (SDN) [NMN+14, XWF+15, KREV+15] advocates the separation between forwarding devices (the data plane) and the network control logic (the control plane). There exists similar ideas that did not succeed in the past, for example, in Active Networking [TSS+97]. However, by appearing at the intersection of ideas, technology maturity and future needs, SDN offers a new potential approach to many existing and new network problems [NMN+14].

Fig. 1.1 shows the SDN architecture. The separation of the control plane and the data plane is realized by a well-defined southbound application programming interface (API). In SDN, network control logic is implemented in an entity, called the controller. The controller is logically centralized but can be physically distributed for scalability. Also, a switch may have multiple controllers for fault tolerance and robustness. Via the southbound API, the controller directly manages the state of forwarding devices to respond to a wide range of network events, for example, when a link is congested, the controller reroutes active flows using this link to other paths. Furthermore, the controller exposes high level northbound APIs for operators to implement high level policies (e.g. firewall, load balancing). With this architecture, SDN promises to reduce costs and to simplify network management thanks to commodity, open forwarding hardwares, and high level management interfaces.

OpenFlow [MAB+08a] is the most popular implementation of the SDN southbound API [NMN+14, XWF+15, KREV+15, JKM+13, KSP+14]. Although OpenFlow starts as academic experiments, it is receiving much attention from both academic and industry. Many vendors have supported OpenFlow in their commercial products [NMN+14]. As an example, Google is using OpenFlow in its WAN for traffic engineering applications [JKM+13]. Open Network Foundation [Fou13], an

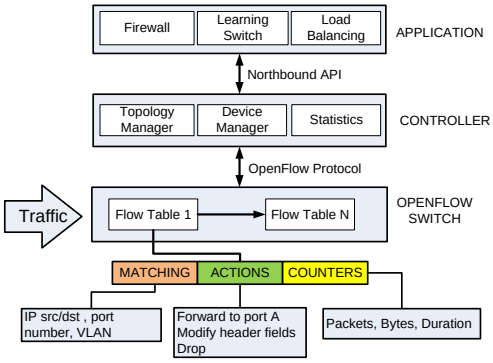


Figure 1.2: OpenFlow Architecture [NMN+14]

industrial-driven organization including the biggest operators (e.g., Google, Facebook, Yahoo, Microsoft) is improving the OpenFlow specification [Ope15b], and promoting OpenFlow as the standard southbound API of SDN.

The architecture of OpenFlow is depicted in Fig. 1.2. Forwarding devices are called OpenFlow switches, and all forwarding decisions are flow-based instead of destination-based like in traditional routers. An OpenFlow switch consists of flow tables, each containing a prioritized list of rules that determine how packets are processed by the switch.[1] A rule consists of three main components: a *matching pattern*, an *actions field* and a *counter*. Generally, a matching pattern is a sequence of $0-1$ bits and "don't care" (denoted as $*$) bits, that forms a predicate for packet meta-information (e.g., $src\_ip = 10.0.0.*$). All packets making true the matching pattern predicate are said to belong to the same *flow*.

The actions field specifies the actions applying to every packet of the corresponding flow, e.g., *forwarding*, *dropping*, or *rewriting* the packets. Finally, the counter records the number of processed packets (i.e., that made the predicate hold true) along with the lifetime of this rule. As a packet may match multiple matching patterns of different rules, each rule is associated with a priority number. Only the rule with the highest priority number that matches the packet is considered to take actions on it. The prioritization of rules permits constructing default actions that can be applied on packets only if no other rule can be used. Examples of default actions are *dropping packets*, or *forwarding to a default interface*. For efficiency and flexibility reasons, the latest versions of OpenFlow [Ope15b] support pipeline processing where a packet might be processed by several rules from different flow tables.

The behavior of an OpenFlow switch strongly depends on the set of rules it holds. With appropriate rules, an OpenFlow switch can act like a Layer-2 switch, a router, or a middlebox. Also, many network applications can be implemented using

[1] We follow the OpenFlow model terminology where a packet consists of any piece of information traveling through the network and a switch stands for any device processing such a packet.

OpenFlow, e.g., monitoring, accounting, traffic shaping, routing, access control, and load balancing [MAB+08a]. To implement these applications and operators policies, it is important to select and install corresponding rules on each OpenFlow switch. However, most of popular controller platforms [GKP+08, Flo15, Eri13] still force operators to manage their network at the level of individual switches, by selecting and installing rules to satisfy network constraints and policies [KLRW13]. Installing inappropriate rules may lead to frequently usages of default actions (e.g., forwarding to the controller), that may overload the controller and degrade network performance, such as packet latency.

Beyond flow abstraction provided by OpenFlow, our motivation is to raise the network abstraction towards a higher one: the *black box*. Using the black box abstraction, operators do not need to care about the complexity and diversity of underlying networks, and how to manage resources efficiently. Furthermore, operators can focus on specifying high level policies, which will be automatically compiled into appropriate OpenFlow rules.

To realize the blackbox abstraction, it is important to study and propose solutions for the problem of selecting and distributing OpenFlow rules, referred as the OpenFlow Rules Placement Problem (ORPP). This problem is not trivial, because in production networks, many rules are required and available for selection [MYSG12], but only a limited amount of resources, and in particular memory [SCF+12], is available on OpenFlow switches. Also, the ORPP is NP-hard in general, as we show in Chapter 4. Despite its complexity, solving this problem is essential to realize the black box abstraction.

1.2 Example Scenarios

In the following, we describe two representative scenarios, that motivate why OpenFlow is needed and why ORPP is challenging.

Access Control As a part of endpoint policy, the firewall policy is critical to the network security. Most of firewall policies can be defined as a list of prioritized access control rules, that specify which flows are authorized and where. OpenFlow is a potential candidate to implement access control applications, because it supports flexible matching patterns and multiple actions.

Ideally, all access control rules should be placed on the ingress switches to filter unwanted network traffic. However, the switch memory constraints prohibit placing all rules in ingress switches. An alternative solution is to put all rules in the software switches having large memory capacity (e.g. RAM), and to direct all traffic to them. However, software switches are generally slower than hardware-accelerated switches (e.g. using TCAM), because of high lookup delay. Therefore, a solution is required to select and distribute rules over all the switches such that the semantic of the original access control list is preserved, and resource constraints are satisfied.

Fig. 1.3 shows an example of access control rules placement. The firewall policy

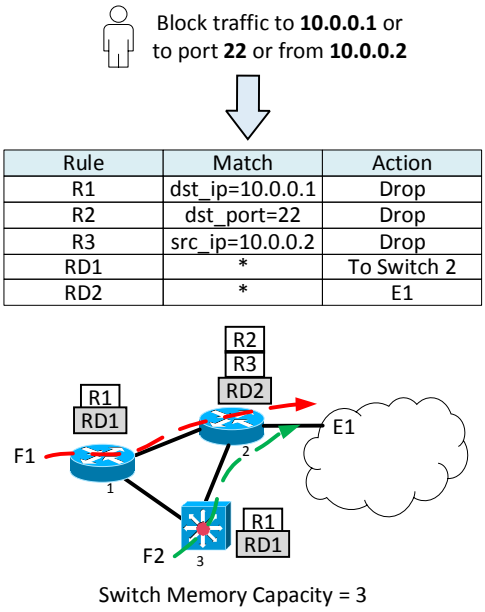


Figure 1.3: An example of access control rules placement. The firewall policy is compiled into a list of rules $R1, R2, R3$ for blocking matching packets, and two default rules $RD1, RD2$ for forwarding non-matching packets towards the endpoint E. These rules are distributed on several switches to ensure that flows $F1$ and $F2$ pass through all rules $R1, R2, R3$ to enforce the firewall policy.

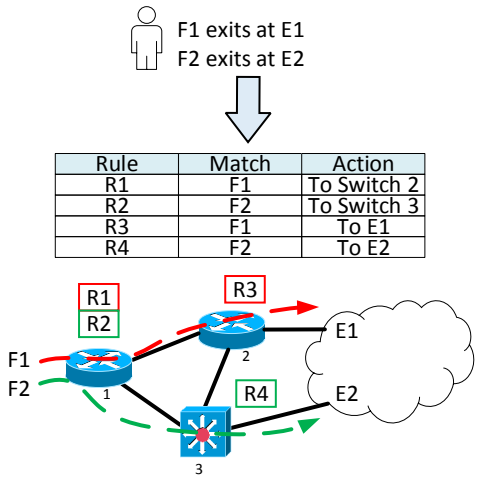


Figure 1.4: An example of forwarding rules placement. Forwarding rules are installed on appropriate paths to make sure that the endpoint policy is satisfied. Rules $R1, R3$ (resp. $R2, R4$) are installed on switches $1, 2$ (resp. $1, 3$) to route F1 (resp. F2) towards its endpoint $E1$ ($E2$).

must be enforced on all flows originated from switch 1 and 3. A solution is to use rules $R1, R2, R3$ for blocking matching packets, and use the default rules $RD1, RD2$ for forwarding non-matching packets towards endpoint E. Then, these rules are distributed on the switches, according to the memory capacity, to enforce the firewall policy on all flows.

Traffic engineering The role of the network is to deliver packets towards their destinations and to satisfy the operator's requirements (e.g., low latency, low loss rate). OpenFlow allows defining rules matching any type of packets and forwarding them on any paths, which promises to support a wide range of policies.

Normally, the forwarding rules matching the packets should be placed on the shortest paths from their sources to their endpoints, to satisfy traffic engineering goals (e.g., delay, throughput). However, due to switch memory limitations, all required rules may not be fit into the shortest paths. Therefore, it is important to select and install forwarding rules on the appropriate paths, to satisfy requirements.

An example of forwarding rules placement is shown in Fig. 1.4. Both flows $F1, F2$ must be forwarded to their corresponding endpoint $E1, E2$. To that aim, a solution is to install forwarding rules $R1, R3$ (resp. $R2, R4$) on switches $1, 2$ (resp. $1, 3$) to route $F1$ (resp. $F2$) towards its endpoint $E1$ (resp. $E2$).

1.3 Research Methodology

Our motivation is to realize the black box abstraction using OpenFlow. To that aim, it is important to understand the ORPP, what are its challenges and existing approaches. Even there are some OpenFlow surveys [XWF+15, KREV+15, NMN+14], but none of them formalizes and discusses that problem comprehensively. Therefore, we first propose a formalization for the problem, classify and discuss existing solutions to find new insights and new potential approaches.

After finding a new potential approach, we use it to address two instances of the problem: (*i*) the *offline* ORPP, in which the set of traffic flows is known and stable in a period, and (*ii*) the *online* ORPP, in which the set of flows is unknown and varies over time. Each assumption allows to apply well-known techniques to solve them. More precisely, we apply optimization techniques for the first instance, and adaptive control techniques for the second one.

For both instances, our goal is to design algorithms that generate rules satisfying policies, network constraints (e.g., memory, bandwidth) while reducing the costs, in terms of the signaling load and default load.[2] On one hand, reducing the signaling load allows the controller to handle more devices and to process requests faster. On the other hand, reducing the default load also improves the network performance. For example, software switches's processing introduces a higher delay than ASIC processing. By reducing the load on these devices (i.e., the default load), the total network delay can be improved.

To evaluate and to compare our proposals to existing solutions, we implement numerical and packet-level simulators using Python. Python is used because of its simplicity, and that it supports network simulation libraries (e.g., NetworkX [Net15], FNSS [SCP13]). Our experiments are performed on cluster platforms such as INRIA NEF [INR15], that allow performing simulations with many configurations simultaneously. Both real and synthetic inputs (e.g., topologies and packet traces) are considered. The outputs are analyzed using Pandas [Pan] and visualized using Matplotlib [Mat]. Beside simulations, some of our proposals (e.g., Wrapper [NST13b, NST13a]) are verified on emulators (e.g., Mininet [LHM10]) and on a commodity OpenFlow switch (Pronto 3290 [PRO15]).

1.4 Thesis Outline

In the following, we summarize the content of each chapter and the obtained results.

In Chapter 2, we present the basic preliminaries used in this thesis. This chapter includes Linear Programming to model optimization problems, Greedy heuristics to find approximate solutions, Exponentially Weighted Moving Average (EWMA) to model and predict future means of a variable, and Increase/Decrease algorithms to adapt a parameter.

[2] load on the default devices (e.g., software OpenFlow switches) that are used to process non-matching packets.

In Chapter 3, we present background related to the fundamental problem of how to select rules and their locations such that network constraints and policies are respected, referred as the *OpenFlow Rules Placement Problem* (ORPP). First, we formalize that problem, and identify two main challenges, including resource limitations and the signaling overhead. Second, we classify, and discuss pros and cons of existing solutions extensively. In the best of our knowledge, this is the first survey focusing on the ORPP.

In Chapter 4, we analyze and demonstrate a limitation of existing solutions. To satisfy endpoint policy, most of existing solutions [KHK13, KLRW13] place rules to enforce flows following the shortest paths. This constraint is sometimes necessary to meet the traffic engineering goals (e.g., low latency). However, in some cases, strictly enforcing this constraint may lead to unfeasible rules placement due to resource constraints (e.g., switch memory). Also, we believe that with the blackbox abstraction, operators do not need to care about the path selection, and can delegate the decision for the controller. Therefore, we propose to trade routing for better resource efficiency, to increase the number of possible paths for rules placement. This approach comes the fact that flows may follow a longer path, but it is compensated by better resource utilization.

Using the new approach, we study the offline ORPP, in which the set of flows is known and stable in the considered period. First, we propose a heuristic that selects the paths consuming less memory, called the *deflection* technique. Second, we prove that the offline ORPP is NP-Hard and formalize it as an Integer Linear Programming (ILP). That model supports various objective functions, and includes necessary constraints such as memory, endpoint policies, bandwidth constraints. Optimal rules can be found by solving that ILP using LP solvers, such as CPLEX [IBM]. A Greedy heuristic, called OFFICER, is designed to find rules in polynomial time complexity for the problem instance with large inputs, e.g., large number of flows. We then perform numerical simulations in realistic scenarios, to compare OFFICER to the optimum and a random rules placement solution.

In reality, flows are often unknown and hard to predict accurately [BAAZ11]. Therefore, the solutions proposed in Chapter 4 can not be directly applied when flows are unknown. In Chapter 5, we study the online ORPP, in which the set of flows is unknown and varies over time. To solve this problem, existing controller platforms [Flo15, GKP+08, Eri13] treat all flows equally and place rules reactively. However, this approach has several cons. First, it incurs a high signaling load, a high latency due to many flows. Second, rules for large flows may not be installed because resources are already occupied by other flows. As consequences, policies can be violated, and network performance are degraded.

We argue that in case of resource limitations (e.g., switch memory), only rules for important, large flows should be installed. To that goal, we propose aOFFICER, an adaptive rules placement framework, that can detect candidate flows and install rules for them on efficient paths. Furthermore, aOFFICER can adapt the parameter automatically to respond to fluctuations in flow demands. Our simulation results in realistic scenarios confirm that aOFFICER can reduce costs and does not introduce

large signaling overhead, compared to existing solutions.

With the black box abstraction, operators can implement high level, flexible endpoint policies using OpenFlow, thanks to algorithms OFFICER and aOFFICER proposed in Chapter 4 and 5. In Chapter 6, we exploit a use case of the black box abstraction, in which we improve the performance of content delivery services in cellular networks.

Nowadays, traffic from content delivery services will continue to grow in coming years, which increases CAPEX and OPEX of network management significantly. OpenFlow is a potential approach to address this problem. First, OpenFlow can enable in-network caching functionalities (e.g., using our technical solution [NST13b, NST13a]), so caches can be deployed everywhere in the network. Second, we propose a novel caching framework, named Arothron. The main idea is to split the cache storage on each node into two parts: one uses opportunistic caching, and the other uses preset caching. On one hand, opportunistic caches, which store and replace contents using the LRU mechanism, can absorb short term fluctuations in content demands. On the other hand, preset caches, which store popular contents, can satisfy long term content demands with high cache hit ratios. To decide which content is stored in which preset cache, a Mixed Linear Integer Programming and a Greedy heuristic are used. With extensive simulations in realistic scenarios, we show that network performances are better if each storage unit combines both opportunistic and preset caching, compared to using only opportunistic caching or using only preset caching. Second, we observe that the optimal ratio between the opportunistic and the preset cache on each node is not the same, and it depends on the node location.

Finally, in Chapter 7, we summarize the content of the thesis, and discuss potential research directions.

1.5 Publications

The complete list of my publications is the following.

International Journals

[WNTS16] M. Wetterwald, **X.N. Nguyen**, T. Turletti, and D. Saucez, "**SDN for Public Safety Networks**", under submission, 2016

[NSBT15b] **X.N. Nguyen**, D. Saucez, C. Barakat and T. Turletti, "**Rules Placement Problem in OpenFlow Networks: a Survey**", IEEE Communications Surveys and Tutorials, October 2015 (Impact Factor: 6.490)

[NMN+14] BAA Nunes, M. Mendonca, **X.N. Nguyen**, K. Obraczka, T. Turletti, "**A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks**", IEEE Communications Surveys and Tutorials, February 2014 (Impact Factor: 6.490)

International Conferences, Workshops

[KNS+15] M. Kimmerlin, **X.N. Nguyen**, D. Saucez, J. Costa-Requena and T. Turletti, "**Arothron: a Versatile Caching Framework for LTE**", under

submission, 2015

[NSBT15a] **X.N. Nguyen**, D. Saucez, C. Barakat and T. Turletti, "**OFFICER: A general Optimization Framework for OpenFlow Rule Allocation and Endpoint Policy Enforcement**", IEEE INFOCOM 2015, Hongkong, China, April 2015 (acceptance ratio: 19%)

[NSBT14] **X.N. Nguyen**, D. Saucez, C. Barakat and T. Turletti, "**Optimizing rules placement in OpenFlow networks: trading routing for better efficiency**", ACM HotSDN 2014, Chicago, United States, August 2014 (acceptance ratio: 28.9%)

[NST13a] **X.N. Nguyen**, D. Saucez and T. Turletti, "**Efficient caching in Content-Centric Networks using OpenFlow**", IEEE INFOCOM 2013 Workshop Proceedings, Turin, Italy, April 2013 (acceptance ratio: 14.3%)

Research Reports

[NST13b] **X.N. Nguyen**, D. Saucez and T. Turletti, "**Providing CCN functionalities over OpenFlow switches**", INRIA Research Report 00920554, 2013

[Ngu12] **X.N. Nguyen**, "**Software Defined Networking in Wireless Mesh Networks**", Master Ubinet Thesis, University of Nice Sophia Antipolis, 2012

Chapter 2

Preliminaries

Contents

2.1 Linear Programming 11
2.2 Greedy Algorithms 12
2.3 Exponentially Weighted Moving Average Model 13
2.4 Increase/Decrease Algorithms 14

In this chapter, we present some preliminaries that we use throughout this thesis.

2.1 Linear Programming

Linear Programming (LP) is a general method to model and to achieve the best outcomes of many combinatorial problems [Sch86, Chv83], such as the 0-1 Knapsack problem. Given a set of items with different weights and values, the motivation of the 0-1 Knapsack problem is to find which item should be selected, such that the total weight is less than or equal to a given limit, and the total value is as large as possible.

LP is widely applied to various domains, such as business, economic, engineering. LP has proved its useful in modeling and solving diverse types of problems (e.g., planning, scheduling, assignment).

Basically, a LP is composed of a linear *objective* function, a set of linear inequality *constraints* formalized by *variables* (i.e., the problem outputs) and *parameters* (i.e., the problem inputs). The objective function represents the optimization target, and it can be written in terms of minimizing or maximizing, e.g., minimizing memory consumption, maximizing user satisfaction. If the goal is just to find feasible solutions satisfying constraints, the objective function can be omitted. Normally, a LP is expressed as:

$$max\{c^T x : Ax \leq b; x \geq 0\} \tag{2.1}$$

In Eq. 2.1, x represents the vector of variables, b, c are vectors of coefficients, A is matrix of coefficients, and $(.)^T$ is the matrix transpose function.

If some variables in x are integrals, the LP is called a Mixed Integer Linear Programming (MILP). if all variables in x are integrals, the LP is called an Integer

Linear Programming (ILP). For example, the 0-1 Knapsack problem mentioned above can be represented as the following ILP:

$$max\{\sum_{i=1}^{n} v_i x_i : \sum_{i=1}^{n} w_i x_i \leq W; x_i \in \{0;1\}\} \tag{2.2}$$

In Eq. 2.2, W is the weight limit; v_i, w_i are the value and the weight of item i; x is the binary variable that indicates if item i is selected ($x_i = 1$) or not ($x_i = 0$).

The problem in the form 2.1 is called the primal problem, and there exists a dual problem:

$$min\{b^T y : A^T y \geq c; y \geq 0\} \tag{2.3}$$

The objective of dual problem, at any feasible solution, is always greater than the primal's objective function, at any feasible solution. Furthermore, if the primal problem has an optimal solution x^*, then its dual has an optimal solution y^*, such that:

$$c^T x^* = b^T y^* \tag{2.4}$$

These above properties are often used to find bounds for the objective function. In some cases, bounds are also used as a stopping condition for solving algorithms.

Due to its wide applications, many methods have been proposed to solve LP, such as cutting plane, brand and bound, column generation, and row generation [Sch86, Chv83]. These methods are usually implemented in LP solvers (e.g., CPLEX [IBM], GLPK [GNU13]), which can find exact and approximate solutions for the problem. A brief view of popular solvers can be found in [CCK+10]. In our study, we use CPLEX [IBM] because it outperforms other open-source solvers in many cases [MT13], and it is free for academic use.

In this thesis, we apply LP to model OpenFlow Rules Placement Problem as an ILP (Chapter 4) and the content placement problem as a MILP (Chapter 6). These LPs are then implemented on CPLEX to find the optimal placements for rules and contents.

2.2 Greedy Algorithms

In general, most of placement problems are NP-Hard, which means that there is no known polynomial-complexity algorithms to find the optimal solutions. Therefore, using LP solvers for large problem instances (e.g., large number of variables, large number of constraints) is not practical due to a large execution time.

To cope with this limitation, heuristics are used to find near optimal solutions in acceptable execution time. Starting from a trivial solution, a heuristic tries to improve the solution in each step, and terminates when the obtained solution is good enough. There are many popular heuristics, such as Greedy, Simulated Anneal [Sch86]. Each heuristic and their parameters are implemented and optimized differently for different problems.

To evaluate the performance of a heuristic, the approximation factor (or approximation ratio) p is used. A heuristic is called p-approximation if for any inputs, heuristic's objective value over optimum objective value is at least p.

Greedy is a common heuristic, widely used in different domains, for example, machine learning, artificial intelligence. In ad-hoc mobile networks, Greedy is also used to route packets with the fewest number of hops and the smallest delay.

Basically, Greedy follows the locally optimal choice in each step with the hope of finding the global optimum. For example, to find solutions for the 0-1 Knapsack problem, one possible greedy strategy is that in each step, the unselected item with maximum value of (v_i/w_i) is selected. In many problems, Greedy does not guarantee to find the global optimum, but it can approximate the global optimum in a reasonable execution time. For example, Greedy has been proven that it is 1/2-approximation for the Knapsack problem [Sch86].

Using Greedy, the solution for small problem instances can be easy and straightforward. However, for large problem instances, in some cases, short term decisions may lead to worst long term outcome.

Basically, a Greedy has the following components:

- A candidate set, from which a solution is created.
- A selection function, which decides how to select the best candidate for a set.
- A feasible function, which checks if a candidate can be selected.
- A stop condition, which indicates when the algorithm should stop.

For example, in Knapsack problem, the *candidate set* is the set of items. The *selection function* picks the largest-value item among available items. After that, this item is checked by the *feasible function* (e.g., if its weight $<$ the available capacity). If the item can be added, it is marked as selected. Then, the same process is repeated for the rest of items. Finally, Greedy returns the solutions when all items are verified or the *stop condition* is satisfied (e.g., if the available capacity $<$ 5% total capacity).

In this thesis, we design our heuristics based on Greedy, to find rules placement solutions in Chapter 4 and content placement solutions in Chapter 6. The key ideas of these heuristics are to install rules for the largest flows, or to place the most popular contents first, since they contribute most to the objective function. The evaluation results in realistic scenarios confirm the simplicity and the efficiency of these heuristics, compared to the optimum.

2.3 Exponentially Weighted Moving Average Model

In statistics, a *moving average* is the average that moves. More precisely, it is a set of numbers, each corresponding to the average of a subset of a larger data set. A moving average is commonly used to analyze financial data, such as stock prices,

to get an overall idea about the trend of the data set. Furthermore, it can smooth out short-term fluctuations, and forecast long term trends. Mathematically, it can also be considered as a low pass filter, where high frequencies (e.g., short term fluctuations) are removed from the original signal.

For example, given a time series $D = \{1, 3, 2, 3, 4, 5\}$ that represents the price of a stock in each day. 3-days moving average S_i, represents the evolving of the stock prices, is computed based as the average price of 3 consecutive days:

$$S_1 = (1 + 3 + 2)/3 = 2 \tag{2.5}$$
$$S_2 = (3 + 2 + 3)/3 = 2.33 \tag{2.6}$$
$$S_3 = (2 + 3 + 4)/3 = 3 \tag{2.7}$$
$$S_4 = (3 + 4 + 5)/3 = 4 \tag{2.8}$$

There are different kinds of moving averages. An *Exponential Weighted Moving Average* (EWMA) model [KO90] is a kind of *moving average*, where the weight factor of each older datum decreases exponentially with ages. EWMA has an advantage compared to simple MA, as EWMA remembers a fraction of its history and accounts them in future values. Given a time series $X = \{X_0, X_1, ...\}$, the EWMA for a time series X is calculated recursively as the following:

$$S_0 = X_0 \tag{2.9}$$
$$S_{t+1} = \alpha S_t + (1 - \alpha) X_t \tag{2.10}$$

In Eq. 2.10, X_t, S_t are the variable value, and the EWMA at time t. The parameter $0 < \alpha \leq 1$ represents the weight contributed of old samples into the future mean value. Smaller α discounts old samples faster, and highlights the important of the most recent sample. Higher α indicates slow decay in time series, in other terms, the time series falls off more slowly. Selecting the right value of α is a matter of preferences and experiences.

In Chapter 5, we apply the EWMA model to estimate the future mean value for a time series of binary installation trials (0: fail, 1: success).

2.4 Increase/Decrease Algorithms

The Increase/Decrease (ID) algorithm is a type of feedback control algorithms, best known for its use in TCP Congestion Avoidance [APB09], to adjust the transmission rate (window size) of the transmitters.

ID algorithms combines addictive/multiplicative (denoted as A/M) increasing and addictive/multiplicative decreasing to adjust a parameter, when conditions are satisfied or periodically. For example, in TCP Congestion Avoidance, AIMD algorithm is used. More precisely, the transmission rate is addictive increasing (AI) by a fixed amount for every round trip time. When the congestion (e.g., loss occurs) is detected, the transmitter decreases the transmission rate by a multiplicative factor (1/2). Moreover, if multiple flows use AIMD algorithms, they will eventually

converge to use an equal amount of link capacity. Other types of ID algorithms, such as MIMD and AIAD, do not converge for this case.

Basically, an ID algorithm can be expressed as follows:

$$W_{t+1} = M_1 * W_t + A_1 \tag{2.11}$$

$$W_{t+1} = W_t / M_2 - A_2 \tag{2.12}$$

In Eq. 2.11 and 2.12, W_t is the value of the variable at time t, $M_1, A_1, M_2, A_2 \in [0; \infty)$ are constant addictive/multiplicative factors. There values are selected based on experiences, and they affect to the convergence speed, size of oscillations, and the possible values of the control parameter. Depending on conditions, increasing phase (Eq. 2.11) or decreasing phase (Eq. 2.12) is called.

In Chapter 5, we use the MIMD algorithm to adjust the threshold H according to the average success rate r (estimated using EWMA model) periodically. More precisely, H is multiplicative increasing when the $r < r_0$ (r_0 is an expected value for r), and multiplicative decreasing when $r > r_0$. The MIMD algorithm is used because it quickly converges in our scenarios.

Chapter 3

Literature Review

Contents

3.1 Introduction 17
3.2 OpenFlow Rules Placement Problem 18
3.2.1 Problem Formalization 18
3.2.2 Challenges 20
3.3 Efficient Memory Management 21
3.3.1 Eviction 22
3.3.2 Compression 24
3.3.3 Split and Distribution 28
3.4 Reducing Signaling Overhead 32
3.4.1 Reactive and Proactive Rules Placement 33
3.4.2 Delegating Functions to OpenFlow switches 34
3.5 Conclusion 35

In this chapter, we review the state of the art around the problem of selecting and distributing OpenFlow rules, which we refer as OpenFlow Rules Placement Problem. To the best of our knowledge, this is the most comprehensive survey about the OpenFlow Rules Placement Problem, including a formalization, a classification, and a discussion of the related work. The remainder of this chapter corresponds to our publication [NSBT15b].

3.1 Introduction

Computer networks today consist of many heterogeneous devices (e.g., switches, routers, middleboxes) from different vendors, with a variety of sophisticated and distributed protocols running on them. Network operators are responsible for configuring policies to respond to a wide range of network events and applications. Normally, operators have to manually transform these high level policies into low level vendor specific instructions, while adjusting them according to changes in network state. As a result, network management and performance tuning are often complicated, error-prone and time-consuming. The main reason is the tight coupling of network devices with the proprietary software controlling them, thus making it difficult for operators to innovate and specify high-level policies [MAB+08a].

Software-Defined Networking (SDN) advocates the separation between forwarding devices and the software controlling them in order to break the dependency on a particular equipment constructor and to simplify network management. In particular, OpenFlow implements a part of the SDN concept through a simple but powerful protocol that abstracts network communications in the form of flows to be processed by intermediate network equipments with a minimum set of primitives [MAB+08a].

OpenFlow offers many new perspectives to network operators and opens a plethora of research questions such as how to design network programming languages, obtain robust systems with centralized management, control traffic at the packet level, perform network virtualization, or even co-exist with traditional network protocols [FHF+11, VKF12, VWY+13, NMN+14, XWF+15, KREV+15, VCB+15]. For all these questions, finding how to allocate rules such that high-level policies are satisfied while respecting all the constraints imposed by the network is essential. The challenge being that while potentially many rules are required for traffic management purpose [MYSG12], in practice, only a limited amount of resources, and in particular memory [SCF+12], is available on OpenFlow switches. In this chapter, we survey the fundamental problem when OpenFlow is used in production networks, that we refer to it as the *OpenFlow Rules Placement Problem*. We focus on OpenFlow as it is the most popular southbound SDN interface that has been deployed in production networks [JKM+13].

The contributions of this chapter include:

- A generalization of the OpenFlow rules placement problem and an identification of its main challenges involved.
- A presentation, classification, and comparison of existing solutions proposed to address the OpenFlow rules placement problem.

This chapter is organized as follows. In Sec. 3.2, we formalize the OpenFlow rules placement problem, discuss the challenges. We continue with existing ideas that address the two main challenges of the OpenFlow rules placement problem: memory limitation in Sec. 3.3, and signaling overhead in Sec. 3.4.

3.2 OpenFlow Rules Placement Problem

3.2.1 Problem Formalization

In the following, we formalize the OpenFlow Rules Placement Problem using the notations in Table 3.1.

The network is modeled as a directed graph $G = (V, E)$, where V is the set of nodes and each node $v \in V$ can store C_v rules, E is the set of directed links of the network. O is the set of endpoints where the flow used to exit the network (e.g., peering links, gateways, firewalls). A flow can have many endpoints $o_f \in O$. P is the set of possible paths that flows can use to reach their endpoints $o_f \in O$. Each path $p \in P$, consists of a sequence of nodes $v \in V$. F, R are the set of flows and rules for selection, respectively.

Table 3.1: Notations used in this chapter

Notation	Definition
V	set of OpenFlow nodes
E	set of links
O	set of endpoints (e.g., peering links)
F	set of flows (e.g., source – destination IP flows)
R	set of possible rules for selection
T	set of time values
$FT(v,t)$	flow tables of node i at time $t \in T$
C_v	memory capacity of node v (e.g., in total number of rules)
P	set of possible paths to the endpoints (e.g., shortest paths)
m	matching pattern field (e.g., $srcIP = 10.0.0.*$)
a	actions field (e.g., dropping the packets)
q	priority number (0 to 65535)
t_{idle}	idle timeout (s)
t_{hard}	hard timeout (s)
EP	endpoint policy, defines the endpoint(s) $o \in O$ of $f \in F$
RP	routing policy, defines the path(s) $p \in P$ of $f \in F$

The **output** of the problem is the content of the flow table of node $FT(v,t) = [r_1, r_2, ...] \subset R$, which defines the set of rules required to install node $v \in V$ at time $t \in T$. $T = [t_1, t_2, ...]$ is the set of the time instants at which $FT(v,t)$ is computed and remains unchanged during the period $[t_i, t_{i+1}]$. Each rule r_j is defined as a tuple, which contains values for matching pattern m, actions a, priority number q and timeouts t_{idle}, t_{hard}, selected by the solvers.[1] The flow table content of all nodes $FT(v,t), \forall v \in V$ at a time t is defined as a *rules placement* solution. Furthermore, $FT(v,t)$ changes over time t to adapt with network changes (e.g., topology changes, traffic fluctuation). In order to construct rules placement, the following **inputs** are considered:

- **Traffic flows** F, which stand for the network traffic. The definition of a flow, implemented with the matching pattern, depends on the granularity needed to implement the operator policies. For example, network traffic can be modeled as the set of Source-Destination (SD) flows, each flow is a sequence of packets having the same source and destination IP address.

- **Policies**, which are defined by the operator, can be classified into two categories: (*i*) the *end-point policy* $EP : F \rightarrow O$ that defines where to ultimately deliver packets (e.g. the cheapest link) and (*ii*) the *routing policy* $RP : F \rightarrow P$ that indicates the paths that flows must follow before being delivered (e.g., the shortest path) [KLRW13]. The definition of these policies is often the result

[1] We focus on important fields only. The complete list of fields of an OpenFlow rule can be found in the OpenFlow specifications [Ope15b].

of the combination of objectives such as cost reduction, QoS requirements and energy efficiency [GMP14, NSBT15a, HLGY14, KLRW13, LLG14].

- **Rule space** R, which defines the set of all possible rules for selection, depending on applications. For example, an access control application allows selecting rules that contain matching m for 5-tuples IP fields (source/destination IP address, source/destination port number, protocol number) while a load balancing application requires rules that contain matching m for source/destination IP address [WBR11]. The combination of fields and values forms a large space for selection.

- **Resource constraints**, such as memory, bandwidth, CPU capacity of the controller and nodes. Rules placement solutions must satisfy these resource constraints. As an example, the total number of rules on a node should not exceed the memory capacity of the nodes: $|FT(v,t)| \leq C_v, \forall (v,t) \in V \times T$.

There might be a countless number of rules placement possibilities that satisfy the above inputs. Therefore, $FT(v,t)$ is usually selected based on additional requirements, such as in order to minimize the overall rule space consumption $\sum_{v \in V} |FT(v,t)|$. Note that in general, the OpenFlow Rules Placement problem is NP-hard, as we will proof in Chapter 4 by reducing it to the Knapsack problem.

3.2.2 Challenges

Elaborating an efficient rules placement is challenging due to the following reasons.

3.2.2.1 Resource limitations

In most of production environments, a large number of rules is required to support policies whereas network resources (e.g., memory) are unfortunately limited. For example, up to 8 millions of rules are required in typical enterprise networks [YRFW10] and up to one billion for the task management in the cloud [MYSG12]. According to Curtis et al. [CMT+11], a Top-of-Rack switch in data centers may need 78,000 rules to accommodate the traffic.

While the number of rules needed can be very large, the memory capacity to store rules is rather small. Typically, OpenFlow flow tables are implemented using Ternary Content Addressable Memory (TCAM) on switches to ensure matching flexibility and high lookup performance. However, TCAM is board-space costly, is 400 times more expensive and consumes 100 times more power per Mbps than RAM-based storage [KARW14]. Also, the size of each flow entry is 356 bits [Ope15b], which is much larger than the 60-bit entries used in conventional switches. As a consequence, today commercial off-the-shelf switches support only from 2k to 20k rules [SCF+12], which is several orders of magnitude smaller than the total number of rules needed to operate networks. Kobayashi et al. [KSP+14] confirm that the flow table size of commercial switches is an issue when deploying OpenFlow in production environments.

Recently, software switches built on commodity servers (e.g., OpenvSwitch [Ope15c]) are becoming popular. Such switches have large flow table capacity and can process packets at high rate (e.g., 40 Gbps on a quad-core machine [KARW14]). However, software switches are more limited in forwarding and lookup rate than commodity switches [MYSG13] for two main reasons. Firstly, software switches use general purpose CPU for forwarding, whereas commodity switches use Application-Specific Integrated Circuits (ASICs) designed for high speed throughput. Secondly, rules in software switches are stored in the computer Random Access Memory (RAM), which is cheaper and larger, while rules in commodity switches are stored in TCAM, which allows faster lookup but has limited size. For example, an 8-core PC supports forwarding capacities of 4.9 millions packets/s, while modern switches using TCAMs do forwarding at a rate up to 200 millions packets/s [MNL+10].

To accelerate switching operations in software switches, flow tables can be stored in CPU caches. Nevertheless, these caches are rather small, which brings the same problem than with ASICs.

In Sec. 3.3, we extensively survey the techniques proposed in the literature to cope with the memory limitation in the context of the OpenFlow Rules Placement Problem.

3.2.2.2 Signaling overhead

Installing or updating rules for flows triggers the exchange of OpenFlow messages. Inefficient rules placement solutions might also cause frequent flow table misses that would require the controller to act. While the number of messages per flow is of the order of magnitude of the network diameter, the overall number of messages to be exchanged may become large. For instance, in a data center with 100,000 new flows per second [BAM10], at least 14 Gbps of overall control channel traffic is required [IMS13]. Comparably, in dynamic environments, rules need to be updated frequently (e.g., routing rules may change every 1.5s to 5s [MYSG12] and forwarding rules can be updated hundreds times per second [ADRC14]).

In situations with large signaling load, the controller or switches might be overloaded, resulting in the drop of signaling messages and consequently in potential policy violations, blackholes, or forwarding loops. High signaling load also impacts the CAPEX as it implies investment in powerful hardware to support the load.

We discuss rules placement solutions that deal with signaling overhead in Sec. 3.4.

3.3 Efficient Memory Management

As explained in Sec. 3.2.2, all required rules might not fit into the flow table of a switch because of memory limitations. In this section, we classify the different solutions proposed in the literature to manage the switch memory into three categories. In Sec. 3.3.1, we detail solutions relying on *eviction* techniques. The idea of eviction is to remove entries from a flow table before installing new entries. Afterwards, in Sec. 3.3.2, we describe the techniques relying on *compression*. In OpenFlow,

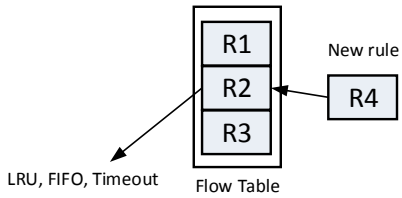


Figure 3.1: An example of eviction. Rule $R2$ in the flow table is reactively evicted using replacement algorithms (e.g., LRU, FIFO) when the flow table is full and a new rule $R4$ needs to be inserted. $R2$ can also be proactively evicted using, for example, a timeout mechanism.

compressing rules corresponds to building flow tables that are as compact as possible by leveraging redundancy of information between the different rules. Then, in Sec. 3.3.3, we explain the techniques following the *split and distribution* concept. In this case, switches constitute a global distributed system, where switches are inter-dependent instead of being independent from each other. Finally, we provide in Table 3.2 a classification of the related work and corresponding memory management techniques.

3.3.1 Eviction

Because of memory limitation, the flow table of a switch may be filled up quickly in presence of large number of flows. In this case, eviction mechanisms can be used to recover the memory occupied by inactive or less important rules to be able to insert new rules. Fig. 3.1 shows an example where the flow table is full and new rule $R4$ needs to be inserted. In this case, rule $R2$ in the flow table is evicted using replacement algorithms (e.g., LRU, FIFO). $R2$ can also be proactively removed using OpenFlow timeout mechanism.

The main challenge in using eviction is to identify high value rules to keep and to remove inactive or least used rules.

Existing eviction techniques have been proposed such as *Replacement algorithms* (Sec. 3.3.1.1), *Flow state-based eviction* (Sec. 3.3.1.2) and *Timeout mechanisms* (Sec. 3.3.1.3).

3.3.1.1 Replacement algorithms

Well-known caching replacement algorithms such as Least Recent Used (LRU), First-In First-Out (FIFO) or Random replacement can be implemented directly in OpenFlow switches. Replacement algorithms are performed based on lifetime and importance of rules, and is enabled by setting the corresponding flags in OpenFlow switches configuration. As eviction is an optional feature, some OpenFlow switches may not support it [Ope15b]. If the corresponding flags are not set and when the

flow table is full, the switch returns an error message when the controller tries to insert a rule.

Replacement algorithms can also be implemented by using delete messages (OFPFC_DELETE). If the flag OFPFF_SEND_FLOW_REM is set when the rule is installed, the switch returns a message containing the removal reason (e.g., timeout) and the statistics (e.g., flow duration, number of packets) on rule removal [Ope15b]. From OpenFlow 1.4, the controller can get early warning about the current flow table occupation, so it can react to avoid flow table being full [KLC+14]. The desired warning threshold is defined and configured by the controller.

Vishnoi et al. [VPMB14] argue that replacement algorithms are not suitable for OpenFlow. First, implementing them on the switch side violates one of the OpenFlow principles, which is to delegate all intelligence to the controller. Second, implementing them at the controller side is unfeasible because of large signaling overhead (e.g., large number of statistic collections and delete messages).

Among replacement algorithms, LRU outperforms others and improves flow table hit ratio, by keeping most recently used rules in flow table, according to studies [ZGL14, KLC+14]. However, the abundance of mice flows in data center traffic can cause elephant flows' rules to be evicted from the flow table [LKA13]. Therefore, in some cases, replacement algorithms need to be designed to favor rules for important flows.

3.3.1.2 Flow state-based Eviction

In practical, flows vary in duration and size, some flows are much larger and longer than other flows, according to studies [BAM10, KSG+09]. Flow state information can be used to evict rules before their actual expiration, as proposed in [ZGL14, Nev14, KB14]. For example, based on observation of flow packet's flags (e.g., TCP FIN flag), the controller can decide to remove the rule used for that flow by sending delete messages. However, eviction algorithms relying on flow state can be expensive and laborious to implement, because of large signaling overhead [ZGL14].

3.3.1.3 Timeout mechanisms

Rules in flow tables can also be *proactively evicted* after a fixed amount of time (*hard_timeout*) t_{hard} or after some period of inactivity (*idle_timeout*) t_{idle} using the timeout mechanism in OpenFlow switches [Ope15b], if these values are set when the controller installs rules.

Previous controllers have assigned static idle timeout values ranging from $5s$ in NOX [GKP+08], to $10s$ and $60s$ in DevoFlow [CMT+11]. Zarek et al. [ZGL14] study different traces from different networks and observe that the optimal *idle_timeout* value is $5s$ for data centers, $9s$ for enterprise networks, and $11s$ for core networks.

Flows can vary widely in their duration [BAM10], so setting the same timeout value for all rules may lead to inefficient memory usage for short lifetime flows. Therefore, adaptive timeout mechanisms [VPMB14, XZZ+14, KB14, ZFLJ15, WWJ+15]

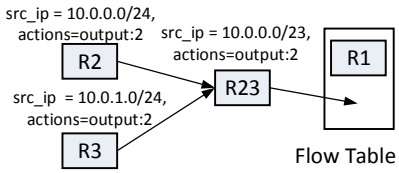


Figure 3.2: An example of compression. $R2$ and $R3$ have the same actions field and are compressed into a single rule $R23$ that has matching pattern covering both matching patterns of $R2$ and $R3$. Thus, rule space consumption is reduced while the original semantic is preserved.

have been proposed. In these studies, the timeout value is chosen and adjusted based on flow state, controller capacity, current memory utilization, or switch location in the network. These approaches lead to better memory utilization and do not require the controller to explicitly delete entries. However, obtaining an accurate knowledge about flow state is usually expensive as it requires a large signaling overhead to monitor and collect statistics at high frequency.

In the original scheme of OpenFlow, when a packet matches a rule, the idle timeout counter of the rule is reset but the gain is limited [XZZ+14]. Therefore, Xie et al. [XZZ+14] propose that switches should accumulate remaining survival time from the previous round to the current round, so that the rules with high matching probability will be kept in the flow table. Considering the observation that many flows never repeat themselves [BAM10], a small idle timeout value in the range of 10ms – 100 ms is recommended for better efficiency, instead of using the current minimum timeout of 1s [Ope15b, VPMB14]. These improvements require modifications in the implementation of OpenFlow.

All of the above studies advocate using idle timeout mechanism, since using hard timeout mechanism may cause rules removal during transmission of burst packets and leads to packet loss, increased latency, or degraded network performance [Nev14].

3.3.2 Compression

Compression (or aggregation) is a technique that reduces the number of required rules while preserving the original semantics, by using wildcard rules. As a result, an original list of rules might be replaced by a smaller one that fits the flow table. As an example in Fig. 3.2, $R2$ and $R3$ have the same actions field and are compressed into a single rule $R23$ that has matching pattern covering both matching patterns of $R2$ and $R3$. Thus, rule space consumption is reduced while the original semantic is preserved.

Traditional routing table compression techniques for IP such as ORTC [DKVZ99] cannot be directly applied to compress OpenFlow rules because of two reasons. First, OpenFlow switches decide which rule will be used based on rule priority number

when there are several matching rules. Second, rules may contain multiple actions and multiple matching conditions, not restricted to IP.

The challenge when using compression is to maintain the original semantics, keep an appropriate view of flows and achieve the best tradeoff between compression ratio and computation time. The limitation of this approach is that today not all the OpenFlow matching fields support the wildcard values (e.g., transportation port numbers).

In the following, we discuss the compression techniques used for access control rules in Sec. 3.3.2.1 and forwarding rules in Sec. 3.3.2.2. Compression techniques may reduce flow visibility and also delay network update; therefore, we discuss its shortcoming and possible solutions in Sec. 3.3.2.3.

3.3.2.1 Access control rules compression

Most of firewall policies can be defined as a list of prioritized access control rules. The matching pattern of an access control rule usually contains multiple header fields, while the action field is a binary decision field that indicates to drop or permit the packets matching that pattern. Normally, only rules with *drop* action are considered in the placement problem since rules with *permit* action are complementary [ZIL+14]. Because the action field is limited to *drop* action, access control rules can be compressed by applying compression techniques on rules matching patterns, to reduce the number of rules required.

To that aim, rule matching patterns are represented in a bit array and organized in a multidimensional space [KHK13, KLRW13], where each dimension represents a header field (e.g., IP source address). Afterwards, heuristics such as Greedy are applied on this data structure to compute optimized wildcard rules. For example, two rules with matching $m_1 = 000$ and $m_2 = 010$ can be replaced by a wildcard rule with $m = 0 * 1$.

Matching patterns usually have dependency relationships. For example, packets matching $m_1 = 000$ also match $m_2 = 00*$, therefore m_2 depends on m_1. When rules with these matching patterns are placed, the conflict between them needs to be resolved. An approach for compression and resolving conflicts is to build a rule dependency graph [KARW14, ZIL+14], where each node represents a rule and a directed edge represents the dependency between them. Analyzing this graph makes it possible to compute optimized wildcard rules and to extract the dependency constraints to fetch for their optimization placement model.

The network usually has a network-wide blacklisting policy shared by multiple users, for example, packets from a same IP address are dropped. Therefore, rules across different access control lists from different users can also be merged to further reduce the rule space required [ZIL+14]. Also, traditional techniques exist to compress access control rules on a single switch [ACJ+07, LMT10, MLT12].

3.3.2.2 Forwarding rules compression

In OpenFlow networks, forwarding rules can be installed to satisfy endpoint and routing policies. A naive approach is to place exact forwarding rules for each flow on the chosen path. However, this can lead to huge memory consumption in presence of large number of flows. Therefore, compression can be applied to reduce the number of rules to install.

Matching pattern of forwarding rules are usually simpler than access control rules, but they have a larger palette of actions (e.g., bounded by the number of switch ports) and they outnumber access control rules by far [BM14]. In addition, forwarding rules compression has stricter time constraints than access control rules compression when it comes to satisfying fast rerouting in case of failure.

OpenFlow forwarding rules can be interpreted as logical expressions [BM14], for example, `('11*', 2)` represents for rules matching prefix `'11*'` and the action field is to forward to port 2. Normally, rules with same forwarding behavior are compressed into one wildcard rule. Also, it is important to resolve conflicts between rules, for example, by assigning higher priority for rule `('11*', 3)` to avoid wrong forwarding caused by rule `('1**', 2)`. To compress and to resolve conflicts, the Expresso heuristic [BM14] borrowed from logical minimization can be applied to obtain an equivalent sets with a smaller size, which represents corresponding rules. Another approach is to compress forwarding rules based on source, destination using the heuristic MINNIE proposed in [RHC+15].

The routing policy plays an important role in applying compression techniques, as it decides the paths where forwarding rules are placed to direct flows towards their endpoints. Single path routing has been widely used because of its simplicity, however, it is insufficient to satisfy QoS requirement, such as throughput [HLGY14]. Hence the adoption of multipath routing. Normally, forwarding rules are duplicated on each path to route flows towards their destinations. By choosing appropriate flow paths such that they transit on the same set of switches, forwarding rules on these switches can be compressed [HLGY14]. For example, flow F uses path $P1 = (S1, S2, S3)$ and $P2 = (S3, S2, S4)$ that have Switch $S2$ in common. On the latter switch, two rules that forward F to $S3, S4$ can be compressed into one rule $match(F) \rightarrow select(S3, S4)$. Also, forwarding rules may contain the same source (e.g., ingress port), that can also be compressed [WWJ+15].

Generally, OpenFlow switches have a default rule with the lowest priority that matches all flows. Forwarding rules can also be compressed with the default rule if they have the same actions (e.g., forwarding with the same interface) [NSBT15a]. Also, forwarding paths for flows can be chosen such that they leverage the default rules as much as possible. In this way, flows can be delivered to their destinations with the minimum number of forwarding rules.

Even though the actions field of a rule may contain several actions (e.g., encapsulate, then forward), the number of combinations of actions is much less than the number of rules and can thus be represented with few bits (e.g., 6 bits) [CSS10]. Several studies [CSS10, IMS13] propose to encode the actions for all intermediate

nodes in a list. This list is added to the header of each packet (e.g., using VLAN field [IMS13]) by the ingress switch. Afterwards, each intermediate node identifies its actions in the list (e.g., using pop VLAN action) and executes them. Finally, the egress switch recovers the original packet and forwards it to the destination. This idea is similar to IP source routing [Pos81]. This approach allows decreasing significantly the number of forwarding rules in the core nodes, but at the same time, it increases the packet size headers of all packets.

3.3.2.3 Shortcomings of the Compression approach

The compression approach reduces the number of required rules, but it makes flows less visible since the wildcard rule is not used for a single flow and consequently, the controller cannot control a flow (e.g., monitoring, rate limitation) without impacting other flows. In many applications, one rule is required for each flow to ensure flow visibility and controllability [JLG+14]. Moreover, finding a rule placement with high compression ratio may require high computation time [LYL14]. Studies [CMT+11, IMS13] point out several insights to address these shortcomings.

First, in many scenarios, full control and visibility over all flows is not the right goal as only some important, significant flows need full control [CMT+11]. For example, load balancing requires handling long lived, high throughput flows. According to traffic analysis studies [BAM10], only a few percentages of flows (called elephant flows), send a large number of bytes and the rest of flows, send a small number of bytes. Therefore, wildcard rules [CMT+11] or default rules [NSBT15a] can be used to handle these flows locally on switches and dedicated rules are installed for elephant flows. In this manner, the number of rules required can be reduced, since according to the flow size distribution [BAM10], the number of elephant flows is much smaller than the number of other kinds of flows.

Second, even if each flow requires full control, usually only one exact-matching rule for the flow in the network is needed [IMS13], and on the rest of the flow path wildcard rules are used to handle it. In [IMS13], a solution is proposed to install an exact forwarding rule for the flow at the first switch, which usually consists in a software switch with a high capacity flow table. At intermediate switches, forwarding rules that have the same output actions can be compressed into one rule [CSS10, IMS13]. Other solutions [NHL+13, ADRC14] leverage exact-matching tables (e.g., MAC forwarding tables), beside the wildcard matching flow tables in switches. More precisely, the network is divided into two domains: one where flows are controlled through wildcard rules and the other with exact-matching rules in these tables. The controller computes and defines the best tuning point (i.e., where the flow starts to use exact-matching rules) per flow basis.

Above solutions can reduce the number of forwarding rules while preserving exact matching rules for flow management (e.g., monitoring, rate limitation). However, the first hop switch is required to have high capacity and to perform intensive computations (e.g., packets header changes), which incurs performance penalty.

Compression also incurs computational overhead and slows down the network

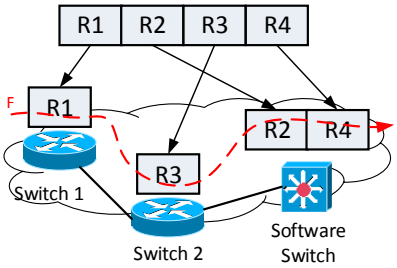


Figure 3.3: An example of distribution approach. The list of access control rules $R1, R2, R3, R4$ is split and distributed, according to the device capacity and such that every flow in F passes through all the rules in the list.

configuration update. Moreover, during the updating time, forwarding errors such as reachability failures and loops are susceptible to occur [LYL14]. Therefore, to be efficient, compression algorithms must achieve a trade-off between compression ratio and update time. In general, most of compressed rules do not change during the update process, so, the designed algorithm only needs to identify and re-compress the affected rules. An example of such an algorithm is iFFTA [LYL14].

3.3.3 Split and Distribution

In general, a single switch does not have sufficient TCAM to store all rules. Therefore, the set of rules is usually split and distributed over the network in a way that satisfies policies. As shown in Fig. 3.3, the list of access control rules $R1, R2, R3, R4$ is split and distributed on the switches, according to the device capacity and such that every flow F passes through all the rules in the list.

The common approach to distribute rules is to formalize an optimization model that decides which rules are placed on which node, such that policy constraints, memory constraints, and rule dependency constraints are satisfied. The objective functions are flexible and depend on applications, such as to minimize the total number of rules needed [KLRW13, KHK13, ZIL+14], to minimize energy consumption [GMP14], or to maximize traffic satisfaction [NSBT15a]. Since the rules placement problem is NP-hard in most of the cases, these studies also propose heuristics to obtain near optimal rules placement solutions.

We first show in Sec. 3.3.3.1 the different options to distribute rules over a network composed of multiple commodity OpenFlow switches built around TCAM-based flow tables. Finally, in Sec. 3.3.3.2 we present how elementary network functions can be performed by software switches or additional network devices to reduce the controller overhead without impairing the management flexibility offered by OpenFlow.

3.3.3.1 Rules distribution among commodity switches

Access control rules distribution There are different solutions to split and distribute access control rules in OpenFlow networks [KHK13, KLRW13, ZIL+14, KARW14, HCWL15].

The first challenge is how to split original access control rules into small, semantic equivalent subsets to fit in flow tables. The common approach is to represent access control rules as a directed dependency graph [KHK13, KARW14, ZIL+14], which can be decomposed into subgraphs (e.g., using Cut Based Decomposition algorithm [KHK13]), corresponding to subsets of rules that maintain original semantics. Other approaches propose splitting rules based on range [YRFW10] or using the Pivot Bit Decomposition algorithm [KHK13].

The second challenge is how to distribute and assign these subsets of rules to switches. To that aim, linear programming models are formalized to assign subsets of rules to switches. Kanizo's model [KHK13] distributes rules over all shortest paths from ingress to the egress node, such that each flow passes through all access control rules. However, as shown in [KLRW13], this approach is suboptimal for two reasons. First, only some paths require enforcing all access control rules. Second, their algorithm cannot use all available switches when the shortest path's length is small. In Kang's model [KLRW13], paths are derived from the routing policy and only the rules that affect packets traversing that path is installed. Zhang's model [ZIL+14] captures the rules dependency and accounts for the compression across rules from different ingress points to further reduce the number of rules required.

Forwarding rules distribution Different forwarding rules distribution algorithms have been proposed to implement forwarding plane for different objectives ([ADRC14, IMS13, NHL+13, HLGY14, NSBT15a, GMP14, LLG14]). The key challenge in forwarding rules distribution is how to select paths to install the forwarding rules that satisfy the policies and network constraints.

Path choice plays an important role in forwarding rules placement. Flows use rules on the paths to reach their endpoints and each path requires different rules. Some paths are more efficient than others; for example, the shortest hop paths are preferred because the minimum number of forwarding rules is needed [IMS13, NHL+13]. The path choice also depends on the traffic engineering goals (e.g., energy efficiency [GMP14]). As shown in Fig. 3.4, to satisfy the endpoint policy (i.e., flow F exits at $E1$), rules can be placed on two different paths, one path needs two rules $R1, R2$ and the other needs three rules $R3, R1, R2$. In this case, the former path is preferred since less memory is consumed.

Flow may not always be carried on a single path (e.g., because of bandwidth constraints). Paths can be chosen such that they satisfy the requirements while maximizing the number of nodes between them, so that all the forwarding rules required can be reduced thanks to compression [HLGY14]. In context of user mobility, paths can be predicted based on velocity and direction, and then forwarding rules can

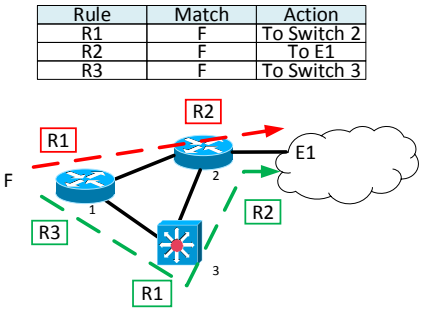


Figure 3.4: An example of path choice. To satisfy the endpoint policy (e.g., flow F exits at $E1$), rules can be placed on two different paths, one path needs two rules $R1, R2$ and the other needs three rules $R3, R1, R2$. In this case, the former path is preferred since less memory is consumed.

be installed on potential paths to avoid transmission interruption [LLG14, WWJ+15, DLOX15].

Proposed studies can be classified into two groups: one group enforcing routing policy (e.g., using shortest paths) [KHK13, KLRW13, ZIL+14, NHL+13, IMS13, LLG14, WWJ+15] and another group that does not [HLGY14, NSBT15a, MYSG12, YRFW10, KARW14, GMP14]. In the first group, the path is an input to the problem, while it is an output in the second group.

Strictly following the routing policy (e.g., using shortest paths) is sometimes necessary to obtain the required performance (e.g., throughput, latency). However, the paths specified by the routing policy may not always have enough capacity to place all the necessary rules.

As our motivation is to make the network becomes a blackbox, we believe that the operators do not need to care about the routing policy. Therefore, we suggest to use relaxing routing policy, which give more flexibility to use resources on other paths so resource utilization, in particular switch memory, can be improved. However, relaxing routing policy may lead to numerous possible paths, which is difficult to choose a suitable one. We will discuss this approach and propose a path heuristic in Chapter 4.

3.3.3.2 Rules distribution among commodity switches and additional resources

Studies presented in Sec. 3.3.3.1 aim to distribute rules on commodity switches and cannot directly be applied to under-provisioned networks where memory budget, in particular with TCAMs, is limited [NSBT14, KARW14].

In practice, some flows are more sensitive to network conditions than others. For example, flows from delay-sensitive applications (e.g., VoIP) require lower latency than best effort traffic (e.g., Web browser). As a consequence, one can allow some flows to be processed on low performance paths and let room for critical flows on

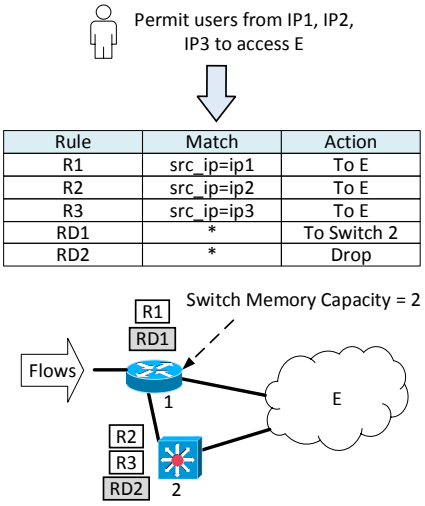


Figure 3.5: An example of using additional devices to offload flow processing from commodity switches. Since Switch 1 does not have enough capacity to place all necessary rules ($R1, R2, R3$), it uses the default rule $RD1$ to redirect flows to software switch 2 for further processing. Flow from $IP1$ passes through switch 1 to reach E, while flow from $IP2, IP3$ passes through switches 1,2 to reach E.

high-performance paths.

Recent studies suggest placing rules on additional, inexpensive devices without TCAM (e.g., software switches) to offload the memory burden for commodity switches [MYSG12, KARW14]. The default rules on the commodity switches can be used to redirect flows that do not match any rule to these devices (e.g., the controller). These devices usually have large capacity (e.g., large flow tables implemented in RAMs), they are cheap to build (e.g., using Open vSwitch on a general purpose CPU [Ope15c]) but have limitations in forwarding and lookup performance, compared to commodity switches. An example is shown in Fig. 3.5; since Switch 1 does not have enough capacity to place all necessary rules ($R1, R2, R3$), the default rule $RD1$ is used to redirect flows to software Switch 2 for further processing. Flows from $IP1$ pass through Switch 1 to reach E, while flows from $IP2, IP3$ pass through Switches $1, 2$ to reach E.

With the support from additional devices, resources are split into two kinds: *fast* (e.g., TCAM matching) and *default* (e.g., software switch matching). Studies [MYSG12, NSBT15a, KARW14] propose rules placement solutions that achieve the best trade-off between performance and cost. Basically, each rule is assigned an importance value, based on its priority and its dependency to other rules. Afterwards, linear programming models and heuristics are used to decide the most profitable rules to keep on commodity switches and the remaining rules to be installed on software switches. The aim of objective functions can be to minimize the

redirection cost [MYSG12], or to maximize the whole values of rules installed on TCAM [KARW14].

The *split and distribution* approach combines different types of resources to perform network-wide optimization and to reduce CAPEX. For example, a switch with a large flow table capacity can be more expensive than several switches with smaller flow tables. Most of the studies formalize an optimization model for rules placement to maximize or minimize an objective function while satisfying different constraints. The main advantage of this approach is the flexibility in objective functions it allows, and its capacity to handle many constraints in a single framework.

However, this approach usually induces a redirection overhead (e.g., redirecting packets causing a flow table miss to other nodes), computation overhead (e.g., solving the optimization model), or rules duplication. Some studies require prediction of the traffic matrix, or future location of users, to be able to solve some optimization models. Such an accurate prediction is costly, because it requires a large signaling overhead to collect network statistics and continuous calibration of the prediction model.

Table 3.2: Comparison of related work by rule placement **mode** (R: Reactive, P: Proactive), memory management techniques (**eviction, compression, distribution**), **use cases** and **validation** methodology

Related work	Mode	Eviction	Compression	Distribution	Use Cases	Validation
Zarek et al. [ZGL14]	R	v			-	Simulation
Kim et al. [KLC+14]	R	v			-	Emulation
Xie et al. [XZZ+14]	R	v			Traffic Engineering	Simulation
Zhu et al. [ZFLJ15]	R	v			Traffic Engineering	Simulation
Vishnoi et al. [VPMB14]	R	v			Traffic Engineering	Prototype
Curtis et al. [CMT+11]	P		v		Flow Management in Data Centers	Simulation
Chiba et al. [CSS10]	R		v		-	Prototype
Luo et al. [LYL14]	-		v		-	Simulation
Braun et al. [BM14]	P		v		BGP Flow Table Management	Simulation
Yu et al. [YRFW10]	R		v	v	Flow Management	Prototype
Agarwal et al. [ADRC14]	R		v	v	Data Forwarding in Data Centers	Prototype
Moshref et al. [MYSG12]	P		v	v	Cloud, Data Centers	Prototype
Nakagawa et al. [NHL+13]	P		v	v	Traffic Engineering	Prototype
Iyer et al. [IMS13]	P		v	v	Traffic Engineering	Emulation
Kanizo et al. [KHK13]	P		v	v	Distributed ACLs	Simulation
Kang et al. [KLRW13]	P		v	v	Distributed ACLs, Load Balancer	Simulation
Katta et al. [KARW14]	P		v	v	Distributed ACLs	Prototype
Huang et al. [HLGY14]	P		v	v	Traffic Engineering	Simulation
Zhang et al. [ZIL+14]	P		v	v	Distributed ACLs	Simulation
Huang et al. [HCWL15]	P		v	v	Distributed ACLs	Simulation
Giroire et al. [GMP14]	P		v	v	Energy efficiency routing	Simulation
Li et al. [LLG14]	P			v	Data forwarding in Mobile networks	Simulation
Nguyen et al. [NSBT15a]	P		v	v	Traffic engineering	Simulation
Wang et al. [WWJ+15]	P	v	v	v	Data forwarding in Vehicle networks	Simulation
Rifai et al. [RHC+15]	R		v	v	Traffic Engineering	Prototype

3.4 Reducing Signaling Overhead

As explained in Sec. 3.2.2, the signaling overhead should not be neglected while solving the OpenFlow rules placement problem. Reducing the signaling overhead is a key factor to increase the scalability of any rules placement solution. In this

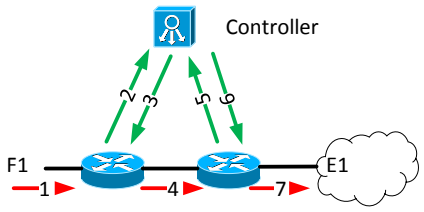


Figure 3.6: An example of reactive rules placement. Rules are placed on demand, after flows arrive.

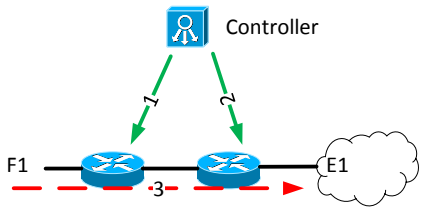


Figure 3.7: An example of proactive rules placement. Rules are placed in advance, before a flow arrives.

section, we summarize the ideas that have been proposed to reduce the signaling overhead.

3.4.1 Reactive and Proactive Rules Placement

There are two approaches for rules placement in OpenFlow: *reactive* and *proactive*.

3.4.1.1 Reactive

With the *Reactive* approach, rules are populated on demand to react upon flow events. As stated in the OpenFlow specification [Ope15b], for each new flow, the switch enqueues the packet and informs the controller about the arrival of a new flow. Afterwards, the controller computes the rules to be associated with the new flow and installs them in the network. So normally, a new flow requires $2n$ messages, where n is the number of path hops. Once the rules are installed on the switches, packets are dequeued and forwarded in the network. Any subsequent packet of the flow will then be processed by the freshly installed rules without further intervention of the controller. An example of reactive rules placement is shown in Fig. 3.6, in which a flow is queued at two switches (Arrow 1, 4). Four OpenFlow messages are required, including two new flow messages (Arrow 2, 5) and two rule installation messages (Arrow 3, 6), to forward the packets of flows towards the endpoint $E1$ (Arrow 7).

Reactive rules placement is required to adjust network configuration continuously with the current network state. For example, a new coming flow requires the controller to setup the path, whereas a down link event requires the controller to reroute all the affected flows.

However, using a reactive approach for all the flows is not the right solution because the controller and the switch buffer may be overloaded, e.g., in presence of large number of new flows (e.g., 100k new flows per second in a cluster [KSG+09]). Another drawback is the additional latency (e.g., 10ms to 20ms in data centers [CMT+11]). Therefore, the reactive approach should not be used for all flows.

3.4.1.2 Proactive

In this approach, rules are populated in advance, i.e., before the first packet of a new flow arrives. The proactive approach nullifies the setup delay of rules and reduces the overall number of signaling messages. An example of reactive rules placement is shown in Fig. 3.7. The controller installs rules for flow $F1$ in advance (Arrow 1, 2), before the flow $F1$ arrives (Arrow 3). So, two OpenFlow messages are required and there is no setup delay.

This *proactive* approach is common in studies focusing on access control [KHK13, KLRW13, ZIL+14, KARW14], as access control rules are predefined by operators independently of the traffic. The same approach can be used to decide forwarding rules in the network but it requires predicting or estimating in advance, the traffic demand or the user location [GMP14, NSBT15a, LLG14, HLGY14, WWJ+15]. In some practical situations, achieving accurate prediction is difficult as it incurs the collection of data and induces signaling messages [BAAZ11]. Therefore, the proactive approach is suitable only for the flows that can be predicted with high accuracy.

We classify the related work that uses proactive and reactive approach in Table 3.2.

3.4.2 Delegating Functions to OpenFlow switches

Rules placement solutions need to be frequently updated often to adapt with current network state. But updating network and collecting statistics incurs load on the controller (e.g., CPU, bandwidth, memory) when done frequently. In this section, we discuss several solutions that can be used to reduce the signaling overhead.

Elementary network functions such as MAC learning and ICMP processing can be delegated to the switches, not only to reduce the signaling overhead, but also to keep basic network functions when controllers are not reachable [KREV+15].

To reduce both signaling overhead and delay caused by new flow setup, several studies [CMT+11, KREV+15, NHL+13] suggest delegating some functions to OpenFlow switches. Instead of querying the controller for each flow, switches can identify and process some flows (e.g., mice flows) and interrogate the controller when decisions are necessary.

Other mechanisms such as *rule cloning* and *local actions* also contribute to

reducing the signaling overhead [CMT+11]. More precisely, *rule cloning* allows the switch to clone a rule from a pre-installed wildcard rule to handle a flow; *local actions* allows the switch to change the action field in rules, for example, fast re-routing to another path in case of link failures, without invoking the controller. Another approach is to use *authority* switches [YRFW10], which are built on top of OpenFlow switches. Authority switches can be used to handle flow table misses from edge switches, thus keeping the packets causing misses in the data-plane.

On rule removal (e.g., because of a timeout), signaling messages are required to inform the controller. To reduce the removal and re-installation overhead, eviction mechanisms like LRU or timeouts (mentioned in Sec. 3.3.1) can be directly implemented on the switches to keep rules with high matching probability in the flow table while automatically freeing space for new flows, everytime without invoking the controller.

Rules placement is computed using statistics queried from the network. For example, by collecting the number of bytes sending so far, the controller can detect that some flows are elephant and then install forwarding rules using the shortest paths. In general, high accuracy inputs require intensive collection of traffic statistics.

To reduce the overhead due to the collection of statistics, the default pull-based mechanism (i.e., the controller requests and receives statistics) can be replaced by a push-based mechanism [CMT+11] (i.e., the switch pushes the statistics to the controller when defined conditions are satisfied, for example, when the number of packets exceeds a threshold). Another complementary solution is to replace current OpenFlow counters by software defined counters [MC12], which support additional features such as data compression and elephant flows detection. In this manner, the statistics collection overhead can be further reduced.

Delegating elementary functions to switches is a way to reduce the signaling overhead between controllers and switches and to increase the overall scalability (e.g., the controller is less loaded) and the availability (e.g., basic network functionalities remain available upon controller failure). However, this approach requires more complex software and hardware than vanilla OpenFlow switches, which increases the cost of the device and may cause inconsistencies as each device makes its own decision [VCB+15].

3.5 Conclusion

Software Defined Networking and OpenFlow offer the ability to simplify network management and reduce costs by raising the level of network abstraction. An abstraction layer between operators and the network is desired to compile the high-level policies from operators into low level OpenFlow rules. To that aim, it is important to solve the *OpenFlow Rules Placement Problem*, that decide the OpenFlow rules that must be deployed and where to install them in order to efficiently use network resources, such as bandwidth and memory, while respecting operational constraints and policies.

In this chapter, we present the body of the literature related to that problem. We first formalize the problem and identify two main challenges: resource limitations and signaling overhead. We then classify and discuss existing solutions to solve these two challenges. Moreover, we identify a limitation of existing solutions when enforcing the routing policy. In the following chapter, we use a new approach to design a novel rules placement algorithm.

Chapter 4

Offline Rules Placement in OpenFlow Networks

Contents

4.1 Introduction . . . 37
4.2 General Model to Allocate Rules in OpenFlow . . . 40
4.3 Rule Allocation Under Memory Constraints . . . 43
4.3.1 Minimizing Memory Usage . . . 44
4.3.2 Maximizing Traffic Satisfaction . . . 44
4.3.3 Heuristic . . . 45
4.4 Evaluation . . . 48
4.4.1 Methodology . . . 49
4.4.2 Results . . . 51
4.5 Discussion . . . 53
4.5.1 Routing Policy . . . 53
4.5.2 Rule Aggregation . . . 53
4.5.3 Multipath . . . 53
4.5.4 Related Work . . . 54
4.6 Conclusion . . . 55

In previous chapter, we define OpenFlow Rules Placement Problem (ORPP) and discuss related work. We also identify a limitation of existing solutions when enforcing the routing policy.

This chapter presents a new approach for offline ORPP, in which the set of flows are known in advance or predicted, for example, traffic matrices that represent Source-Destination (SD) Flows, can be predicted [LHO+14]. We aim to design novel algorithms that generate rules satisfying endpoint policies, network constraints (e.g., memory, bandwidth), while minimizing the percentage of traffic processed by the default devices. The content of this chapter corresponds to our publications [NSBT14, NSBT15a].

4.1 Introduction

The role of a network is to route each packet from an ingress link (i.e., the link from which the packet entered the network) to an egress link (i.e., the link at which the

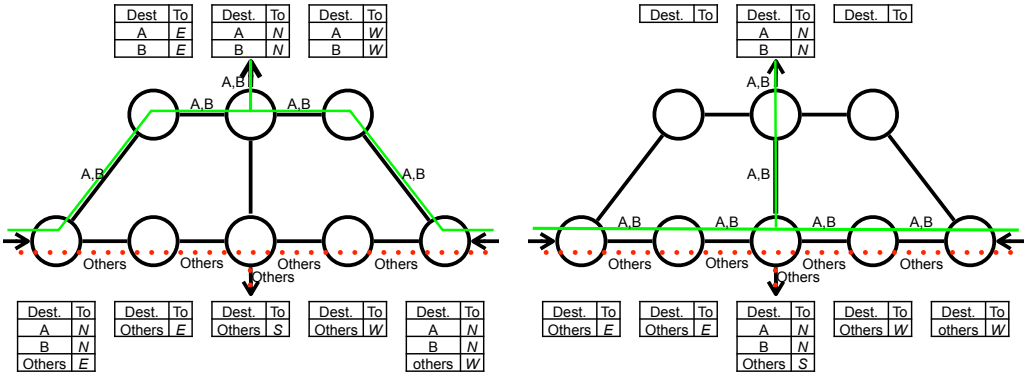
packet leaves the network).[1] According to operational and economical requirements, the choice of the egress link to which a packet must be forwarded is dictated by the *Endpoint Policy* and the actual path followed by a packet in the network is decided by the *Routing Policy* [KLRW13].

Endpoint policies are driven by high-level economical and technical considerations. For example, shared-cost links are often privileged by ISPs and data-centers make sure that packets are delivered to servers able to handle them. On the other hand, routing policies are related to the good usage of resources in the network. Shortest-path routing is the most common routing policy. Its advantages stem from the fact that it minimizes the amount of links and nodes traversed by a packet across the network and that routing tables are computed in polynomial time [Bel58] but other routing policies are also possible, for instance, compact routing [TZ01].

From that point of view, respecting the endpoint policy is essential while the routing policy is just a tool to achieve this goal [JKM+13]. Unfortunately, relaxing routing policies and removing strong path requirements is not practically doable when the network relies on distributed routing algorithms as it would imply a high signaling overhead to ensure consistency of decisions [POB+14]. But with the advent of Software-Defined Networking (SDN) and OpenFlow in particular, it is now possible to manage routing using a centralized approach without losing in terms of scalability or robustness [MAB+08b]. OpenFlow allows operators to conceive their network as a black box aiming at carrying packets from sources to destinations [MAB+08b, JKM+13, NSBT14]. The network thus becomes a single entity that the operator can program instead of a bunch of devices to configure. This is achieved in OpenFlow thanks to a logically centralized controller that fetches information from the network, computes appropriate routes according to the operator wills and network conditions, and then transparently pushes the corresponding forwarding rules into the switches.

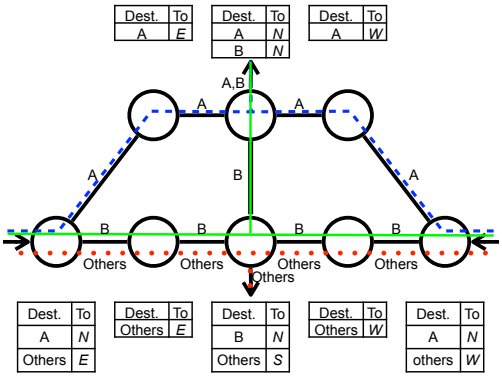
We illustrate the gain from relaxing routing policy in Fig. 4.1 that shows a symmetric network of 8 switches with two ingress links (East and West) and two egress links (North and South). In this example, the endpoint policy stipulates that destinations `A` and `B` must be reached by the North egress link while any other destination must be reached by the South egress link. With the shortest path routing policy (Fig. 4.1a), every destination is reached in 3 hops and for a total of 15 routing entries. With a policy minimizing the number of routing entries (Fig. 4.1b), the routing table is reduced to 9 entries but the memory reduction comes at the cost of longer paths for `A` and `B` (i.e., 4 hops). However, in practice networks might have bandwidth or memory constraints to be respected. For instance, suppose in our network example that each switch can store 2 routing entries. In this case, the two previous routing policies cannot be applied as they would violate the constraints whereas Fig. 4.1c shows an allocation that respects both the endpoint policy and the switches' constraints.

[1] In this chapter, we use the terms *packet*, *router* and *routing table* in their general sense, making no fundamental distinction between packets and frames, routers and switches, or between routing tables and forwarding tables.



(a) Shortest path routing policy

(b) Minimize table routing policy



(c) Constrained network

Figure 4.1: Example of the routing policy on the path followed by packets

Departing from the flexibility offered by OpenFlow, we present OFFICER, a general algorithm to calculate and implement efficient forwarding rules in switches. OFFICER treats the network as a black box that must satisfy the endpoint policy imposed by the operator and tries to get the maximum from the available resources by adapting the routes followed by the different packets towards their desired egress links. When the network is under-provisioned, least valuable packets are routed through a default slow path designed to minimize resource usages. As suggested in [JKM+13] and [NSBT14], we believe that in most networks, enforcing a particular path is not necessary as long as the endpoint policy is respected. Actually, not relying on strict routing policies allows better utilization of the network capacity, reducing so bandwidth wastage and congestion events [JKM+13]. Relaxing routing policy is particularly useful in case of scarce network resources as shown in Fig. 4.1 and in [NSBT14].

The remaining of this chapter presents our algorithm OFFICER to allocate forwarding rules in OpenFlow networks. This algorithm is the result of a general ILP optimization model formulated in Sec. 4.2, where the OpenFlow network is modeled as a directed graph interconnecting switches and the rules to install on switches are to be found. Our model is general in the sense it can accept any endpoint policies and can accommodate any reward functions (i.e., high-level objective) that the operator aims to maximize. Its novelty can be summarized in two main points: (*i*) modeling the network as a black box respecting the endpoint policy, and (*ii*) getting the maximum from the available resources by relaxing routing policy, the rest of the traffic that cannot be installed is routed on a default path. As to be discussed in the related work section (Sec. 4.5.4), we are the first to propose a solution making such abstraction of an OpenFlow network, with a clear gain in terms of the volume of traffic that can be correctly assigned to its desired egress point. To illustrate the flexibility of our proposition, we study the particular case of network that is missing memory to account for all forwarding rules in Sec. 4.3. This is a growing problem in networks because of the increase in size of routing tables but also due to the trend to keep outdated routers in operation [BFCW09]. This problem can even be exacerbated with OpenFlow as it enables very fine granularity on forwarding decisions. In Sec. 4.4 we numerically evaluate the costs and benefits of relaxing routing policy on ISP and data-center topologies and present different heuristics that approximate the optimal algorithm in polynomial time. We open some discussion in Sec. 3.5 to finally conclude in Sec. 4.6.

4.2 General Model to Allocate Rules in OpenFlow

In this section, we formalize a general optimization model for OpenFlow rule allocation and endpoint policy enforcement. The goal of the optimization is to find an allocation of forwarding rules in an OpenFlow network such that the high-level objectives of the operator are respected and network constraints are satisfied. However, depending on the high-level objectives and the network constraints, it may

not be possible to satisfy the endpoint policy for every flow and packets of flows that cannot respect the endpoint policy are then forwarded on an arbitrary *default path*. In the context of OpenFlow, we assume the existence of: (1) a centralized controller that can be reached from every switch in the network and (2) a default path used in every switch to forward packets that do not match any forwarding rule to the controller. [2]

Based on these assumptions, our optimization model is expressed as an Integer Linear Program (ILP) with constraints and the goal is to maximize an objective function that abstracts the high-level objectives of the operator. Without loss of generality, we assume that one forwarding rule is used for at most one flow. This assumption is also used in [JLG+14] to keep the core simple with exact matching rules and easy to manage flows (e.g., rate limitation, accounting). Moreover this has the advantage of keeping our model linear (see 4.5.2).

In the following, we define a *flow* $f \in F$ as a set of packets matching a pattern, starting from one *ingress link* $l_f \in I$ and targeting one of the *egress links* $e_l \in E(f)$. We mean by F the network workload, I the set of ingress links of the network, $E(f) \subseteq E$ is the set of all possible egress links and p_f is the packet rate of flow f.

The optimization builds an $|F|$-by-$|L|$ Boolean allocation matrix denoted by $A = (a_{f,l})$, where $a_{f,l}$ indicates whether flow f passes through the directional link $l = (u, v); u, v \in S^+$ from node u to node v or not. We refer to Table 4.1 for the definition of the different notations used along this paper.

Our optimization model is twofold. One part implements the high-level objectives and the other defines the constraints imposed by the network. For the first part, and without loss of generality, the optimization of the high-level objectives can be written as the maximization of an *objective function* $\mathbb{F}(A, \dots)$. Additional constraints can be added to account for the real network conditions and to limit the space of possible solutions.

The second part of the model consists of a set of constraints on the allocation matrix A to ensure that network limitations and the endpoint policy are respected. Constraints related to the network are defined so to avoid forwarding loops, bandwidth overload, or memory overflow while endpoint policy constraints ensure that packets can only be delivered to valid egress links.

Network constraints:

$$\forall f \in F, \forall l \in L^+ : a_{f,l} \in \{0, 1\} \tag{4.1}$$

$$\forall f \in F, \forall s \in S : \sum_{v \in N^{\rightarrow}(s)} a_{f,(v,s)} = \sum_{v \in N^{\leftarrow}(s)} a_{f,(s,v)} \tag{4.2}$$

$$\forall f \in F : a_{f,l} = \begin{cases} 0 & \text{if } l \in I \setminus \{l_f\} \\ 1 & \text{if } l = l_f \end{cases} \tag{4.3}$$

Constraint (4.1) verifies that $a_{f,l}$ is a binary variable. To avoid forwarding loops, acceptable solutions must satisfy flow conservation constraints (4.2) that ensure that

[2] Our model supports multiple controllers.

Table 4.1: Notations used for the Optimization model.

Notation	Description
F	Set of flows.
S	Set of OpenFlow switches composing the network.
S_e	Set of external nodes directly connected to the network but not part of the network to be optimized (e.g., hosts, provider or customer switches, controllers, blackholes).
S^+	Set of all nodes ($S^+ = S \cup S_e$).
L	Set of directed links, defined by $(s, d) \in S \times S$, where s is the origin of the link and d is its termination.
I	Set of directed ingress links that connect external nodes to OpenFlow switches, defined by $(s, d) \in S_e \times S$. The particular ingress link of a flow $f \in F$ is written l_f by abuse of notation.
E	Set of directed egress links that connect the OpenFlow switches to external nodes, defined by $(s, d) \in S \times S_e$.
L^+	Set of all directed links (i.e., $L^+ = L \cup I \cup E$).
$N^{\rightarrow}(s) \subseteq S^+$	set of incoming neighboring nodes of switch $s \in S$ (i.e., neighbors from which s can receive packets).
$N^{\leftarrow}(s) \subseteq S^+$	Set of outgoing neighboring nodes of switch $s \in S$ (i.e., neighbors towards which s can send packets).
$E(f) \subseteq E$	Set of valid egress links for flow $f \in F$ according to the endpoint policy.
$E^*(f) \subseteq E$	$E^*(f) = E(f) \cup *$, where $*$ denotes the set of links attached to the controller.
$def(s) \in S^+$	Next hop toward the controller from switch $s \in S$.
M	Total switch memory limitation.
C_s	Memory limitation of switch $s \in S$.
B_l	Capacity of link $l \in L^+$.
p_f	Packet rate of flow $f \in F$.

the traffic entering a switch always leaves the switch. Constraint (4.3) is a sanity constraint. It indicates that among all ingress links, packets of the flow can only traverse the ingress link of f.

Bandwidth Constraints:

$$\forall l \in L^+ : \sum_{f \in F} p_f a_{f,l} \leq B_l \tag{4.4}$$

Constraint (4.4) accounts for bandwidth limitation and ensures that the sum of the rates of the flows crossing a link l does not exceed its capacity.[3]

[3]The capacity of a link corresponds to the minimum capacity reserved for delivering packets

Memory Constraints:

$$\forall s \in S : \sum_{v \in N^{\leftarrow}(s) \setminus \{def(s)\}} \sum_{f \in F} a_{f,(s,v)} \leq C_s \tag{4.5}$$

$$\sum_{s \in S} \sum_{v \in N^{\leftarrow}(s) \setminus \{def(s)\}} \sum_{f \in F} a_{f,(s,v)} \leq M \tag{4.6}$$

Constraint (4.5) accounts for when the memory of each switch is known in advance. On the contrary, when the memory to be allocated on a switch is flexible (e.g., in a Network-as-a-Service context or in virtual private networks where the memory is divided between multiple tenants), the operator may see the memory as a total budget that can be freely divided between switches which is accounted by constraint (4.6).

To route a flow f via a directed link $l = (s, d)$, a rule must be installed on switch s. However, if the next hop dictated by the forwarding rule is the same as the one of the default action of the switch, it is unnecessary to install the rule. This simple aggregation of forwarding rules is taken into account in constraints (4.5) and (4.6). We refer to Sec. 4.5.2 for a discussion about rule aggregation.

Endpoint policy constraints:

$$\forall f \in F, \forall l \in E \setminus E^*(f) : a_{f,l} = 0 \tag{4.7}$$

$$\forall f \in F : \sum_{l \in E^*(f)} a_{f,l} = 1 \tag{4.8}$$

Flows need to satisfy the endpoint policy, i.e., packets of flow f should exit the network at one of the egress points predefined in $E(f)$. However, it may not be possible to allocate each single flow and thus, some will be diverted to the controller instead of their preferred egress point. Constraint (4.7) and (4.8) ensure that the endpoint policy is respected by imposing that packets of a flow either exit at one valid egress link or at the controller.

The allocation matrix is a source of information for an operator as it provides at the same time the forwarding table, switch memory occupation, and link usage for a given high-level objective and endpoint policy. It is also important to notice that while a problem may have several equivalent solutions, it may also be unsolvable, depending on the objective function and the constraints. In addition, the general problem is NP-hard as Sec. 4.3.2 demonstrates.

4.3 Rule Allocation Under Memory Constraints

Considering the network as a black box offers flexibility but may lead to the creation of a potentially very large set of forwarding rules to be installed in the network [KLRW13, NSBT14, KHK13]. With current switch technologies, this large

of flows satisfying the endpoint policy. If the link may be used to forward packets of flows not satisfying the endpoint policy, capabilities must be set up to reserve a capacity of at least B_l on the link for flows satisfying the endpoint policy, independently of the total traffic carried by the link.

volume of rules poses a memory scaling problem. Such a problem can be approached in two different ways: either the memory capacity of switches is not known and the problem is then to minimize the overall memory usage to reduce the cost, or the memory capacity is known and the problem becomes the one of finding an allocation matrix that satisfies as much as possible high-level objectives of the operator and the endpoint policy.

In Sec. 4.3.1, we show how to use our model to address the memory minimization problem while in Sec. 4.3.2 we use our model to maximize the traffic satisfaction in case of constrained switch memory. Unfortunately, finding the optimal solution in all circumstances is NP-hard, so we propose a computationally tractable heuristic in Sec. 4.3.3 and evaluate different allocation schemes over representative topologies in Sec. 4.4.

4.3.1 Minimizing Memory Usage

A first application of our model is to minimize the overall amount of memory used in the network to store forwarding rules. This objective is shared by Palette [KHK13] and OneBigSwitch [KLRW13], with always the possibility in our case to relax the routing policy and view the network as a black box. To do so, one has to define the objective function so as to count the number of assigned entries in the allocation matrix as detailed in Eq. (4.9).

$$\mathbb{F}(A, S, N^{\leftarrow}, F) = -\sum_{s \in S} \sum_{v \in N^{\leftarrow}(s) \setminus \{def(s)\}} \sum_{f \in F} a_{f,(s,v)} \tag{4.9}$$

Constraint (4.10), derived from constraint (4.8), is added to prevent packets to always be diverted to the controller (which would effectively minimize memory usage).

$$\forall f \in F : \sum_{l \in *} a_{f,l} = 0 \tag{4.10}$$

Parameters $C_s, \forall s \in S$ and M used by constraints (4.5) and (4.6) should be set to ∞. However, if for technical or economical reasons the individual memory of switches cannot exceed a given value, then C_s must be set accordingly.

4.3.2 Maximizing Traffic Satisfaction

When the topology and switch memory are fixed in advance, the problem transforms into finding a rule allocation that satisfies the endpoint policy for the maximum percentage of traffic.[4] The definition given in Sec. 4.3.1 is sufficient to this end. It must however be complemented with a new objective function, that models the reward from respecting the endpoint policy where a flow that does not see its

[4]This objective is equivalent to minimize the percentage of traffic processed by default devices, a.k.a the default load.

endpoint policy satisfied is supposed not to bring any reward. A possible objective function for this problem is:

$$\mathbb{F}(A, F, E) = \sum_{f \in F} \sum_{l \in E(f)} w_{f,l}\, a_{f,l} \tag{4.11}$$

where $w_{f,l} \in \mathbb{R}_+$ is the normalized gain from flow $f \in F$ if forwarded on link $l \in E(f)$. In other words, $w_{f,l}$ rewards the choice of a particular egress link. In the typical case where the goal is to maximize the volume of traffic leaving the network via an egress point satisfying the endpoint policy, we have $\forall f \in F, \forall l \in E(f) : w_{f,l} = p_f$.

Theorem 1. *The rule allocation problem defined to maximize traffic satisfaction is NP-hard.*

Proof. Let us consider an instance of the problem defined with the objective function (4.11), with the topology consisting of one OpenFlow switch, one ingress link, and one egress link e for all flows. Then, let us assume that the switch memory is larger than the number of flows and thus the limitation only comes from the available bandwidth at the egress link e. The problem then becomes how to allocate rules so as to maximize the gain from the traffic exiting the network at egress link e (the rest of the traffic is forwarded to the controller over the default path). For this instance, we can simplify the problem as follows:

$$maximize \sum_{f \in F} w_{f,e} a_{f,e} \tag{4.12}$$

$$\forall f \in F : a_{f,e} \in \{0, 1\} \tag{4.13}$$

$$\sum_{f \in F} p_f a_{f,e} \leq B_e \tag{4.14}$$

This is exactly the 0-1 Knapsack problem, which is known as NP-hard. In consequence, the rule allocation problem defined with the objective function (4.11) and from which this instance derives is NP-hard. □

4.3.3 Heuristic

Finding a rule allocation that maximizes the value of the traffic correctly forwarded in the network when switch memory is predefined is not tractable (see Theorem 1). Therefore, an optimal solution can only be computed for small networks with a few number of flows. Consequently, we propose in this section a heuristic to find nearly optimal rule allocations in tractable time. The general idea of the heuristic is described in Sec. 4.3.3.1 and the exact algorithm and the study of its complexity is given in Sec. 4.3.3.2.

4.3.3.1 Deflection technique

The number of paths between any pair of nodes exponentially increases with the size of the network. It is therefore impractical to try them all. To reduce the space

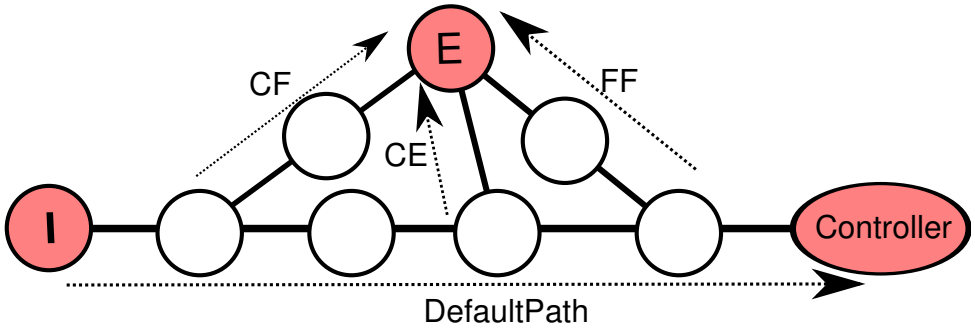


Figure 4.2: Deflection techniques illustrated with 3 deflection strategies.

to explore, we leverage the existence of the default path. Our idea is to forward packets of a flow on the shortest path between the egress point of the flow and one of the nodes on the default path. Consequently, packets of a flow are first forwarded according to the default action and follow the default path without consuming any specific memory entry, then are deflected from the default path (consuming so memory entries) to eventually reach an egress point. That way, we keep tractable the number of paths to try while keeping enough choices to benefit of path diversity in the network. The decision of using the shortest path between default paths and egress points is motivated by the fact that the shorter a path is, the least the number of memory entries to be installed is, letting room for other flows to be installed as well.

To implement this concept, for every flow, switches on the default path are ranked and the algorithm tries each of the switches (starting from the best ranked ones) until an allocation respecting all the constraints is found. If such an allocation exists, a forwarding rule for the flow is installed on each switch of the shortest path from the selected switch on the default path to the egress point. The rank associated to each switch on a default path is computed according to a user-defined strategy. Three possible strategies are:

- **Closest first (CF)**: as close as possible to the ingress link of the flow.
- **Farthest first (FF)**: as close as possible to the controller.
- **Closest to edge first (CE)**: as close as possible to the egress link.

In CF (resp. FF) the weight of a switch on the path is then the number of hops between the ingress link (resp. controller) and the switch. On the contrary, the weight of a switch with CE is the number of hops separating it from the egress point. The deflection techniques and the three strategies are summarized in Fig. 4.2.

4.3.3.2 Greedy algorithm

Algorithm 1 gives the pseudo-code of our heuristic, called OFFICER, constructed around the deflection technique described in Sec. 4.3.3.1. The algorithm is built

upon the objective function in (4.11) that aims at maximizing the overall weight of flows eventually leaving the network at their preferred egress point. The algorithm is greedy in the sense that it tries to install flows with the highest weight first and fill the remaining resources with less valuable flows. The rationale being that the flows with the highest weight account the most for the total reward of the network according to Eq. (4.11).

Algorithm 1: OFFICER

1 **INPUT:** flow weights collection $W : F \times E \to \mathbb{R}_+$, set of network switches S, set of links L^+, set of default path for flows $DefaultPath$, a default path is a set of switches, annotated with a rank, on the path towards the controller.

2 **OUTPUT:** A, a $|F|$-by-$|L^+|$ binary matrix

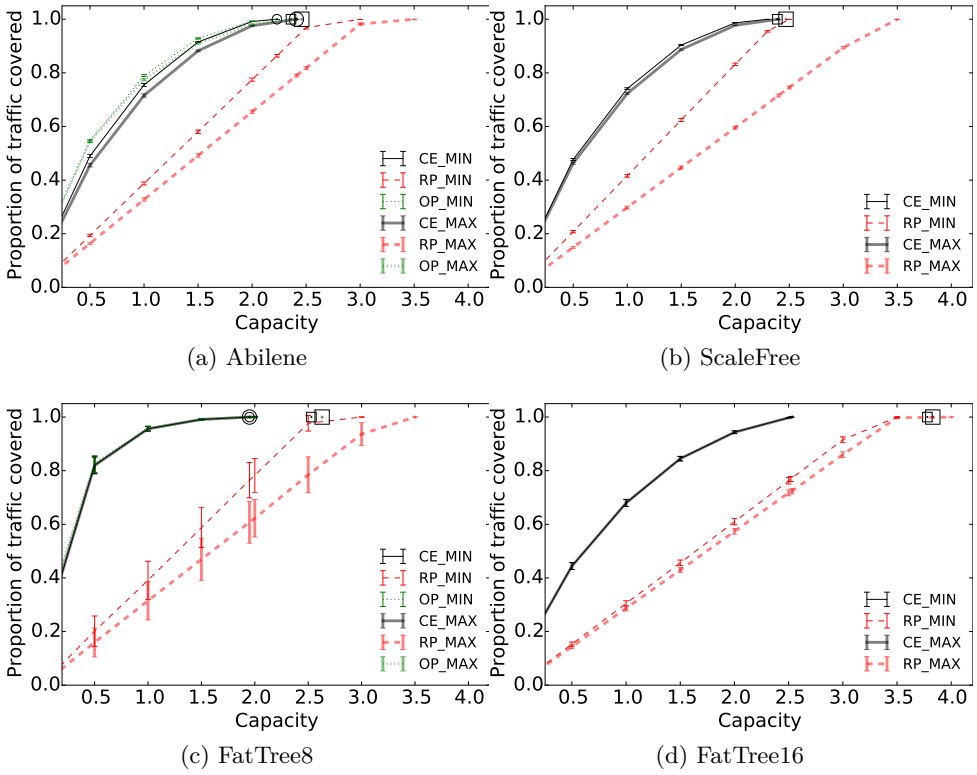
```
1: A ← [0]_{F.L+}
2: M ← sort(W, descending)
3: for all (f, e) ∈ M do
4:    sequence ← sort(DefaultPath(f), ascending)
5:    for all s ∈ sequence do
6:       if canAllocate(A, f, e, s) then
7:          allocate(A, f, e, s)
8:          break
```

Line 2 constructs an order between the flows and their associated egress points according to their weights such that the greedy placement starts with the most valuable flow-egress option. Line 4 determines the sequence of switches along the default path that the algorithm will follow to greedily determine from which switch the flow is diverted from the default path to eventually reach the selected egress point.

The `canAllocate`(A, f, e, s) function determines whether or not flow f can be deflected to egress point e at switch s according to memory, links, and routing constraints. Thanks to constraint (4.8), the `canAllocate` function ensures that a flow is not delivered to several egress points. Finally, the `allocate`(A, f, e, s) function installs rules on the switches towards the egress point by setting $a_{f,l} = 1$ for all l on the shortest path from the deflection point to the egress point. If there are many possible shortest paths, the allocate function selects the path with minimum average load over all links on that path.

When the number of flows is very large w.r.t. the number of switches and the number of links, which is the common case, the asymptotic time complexity[5] of the greedy algorithm is driven by Line 2 and is hence $\mathcal{O}(|F| \cdot log(|F|))$. Unfortunately, even with the polynomial time heuristic, computing an allocation matrix may be challenging, since this matrix is the direct product of the number of flows and links. For example, in data-center networks both the number of links and flows can be very

[5]It is worth to notice that we assume that the algorithm to construct the $DefaultPath$ input is $\mathcal{O}(|F|)$ when the number of flows is large.



(a) Abilene (b) ScaleFree

(c) FatTree8 (d) FatTree16

Figure 4.3: Proportion of traffic covered

large ([BAM10]). With thousands of servers, if flows are defined by their TCP/IP 4-tuple, the matrix can be composed of tens of millions of entries. A way to reduce the size of the allocation matrix is to ignore the small flows that, even if they are numerous, do not account for a large amount of traffic and can hence be treated by the controller.

4.4 Evaluation

In this section, we evaluate our model and heuristic for the particular case of memory constrained networks as defined in Sec. 4.3, for Internet Service Provider (ISP) and Data Center (DC) networks. We selected these two particular deployment scenarios of OpenFlow for their antagonism. On the one hand, ISP networks tend to be built organically and follow the evolution of their customers [SMW02]. On the other hand, DC networks are methodically structured and often present a high degree of symmetry [AFLV08]. Moreover, while workload in both cases is heavy-tailed with a few flows accounting for most of the traffic, DCs exhibit more locality dependency in their traffic with most of communications remaining confined between servers of the same rack [BAM10].

4.4.1 Methodology

We use numerical simulations to evaluate the costs and benefits of relaxing routing policy in a memory constrained OpenFlow network. There are four main factors that can influence the allocation matrix: the topology, the traffic workload, the controller placement, and the allocation algorithm.

4.4.1.1 Topologies

For both ISP and DC cases we consider two topologies, a small one and a large one. As an example of small topology for ISP we use the Abilene [Abi] network with 100 servers attached randomly (labeled `Abilene` in the remaining of the paper). For the large one we use a synthetic scale-free topology composed of 100 switches with 1000 servers attached randomly (labeled `ScaleFree`).

The topologies for DC consist of a synthetic fat tree with 8 pods and 128 servers (labeled `FatTree8`) for the small one, and a synthetic fat tree with 16 pods and 1024 servers (labeled `FatTree16`) for the large one. Both synthetic topologies are randomly produced by the generator proposed by Saino et al. in [SCP13]. Details of the topologies are summarized in Table 4.2. To concentrate on the effect of memory on the allocation matrix, we consider infinite bandwidth links in all four topologies.

Table 4.2: Topology description

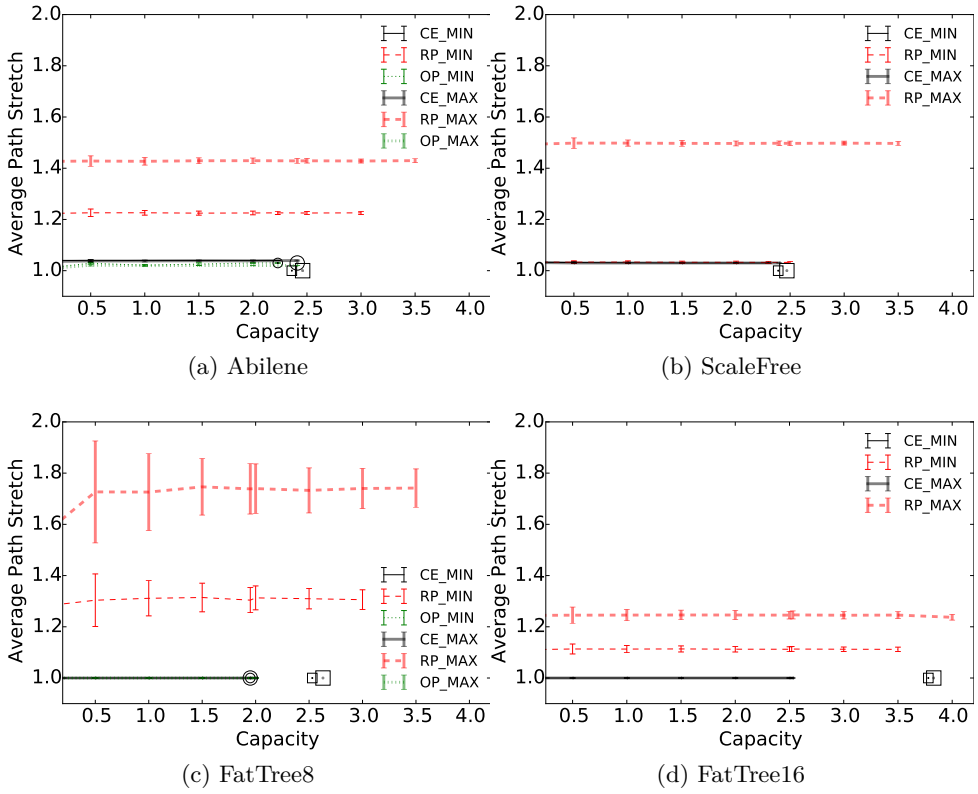
Topology Name	Type	$\|S\|$	$\|L\|$	$\|H\|$	$\|F\|$
Abilene	Small ISP	12	30	100	$\mathcal{O}(10^4)$
ScaleFree	Large ISP	100	292	1000	$\mathcal{O}(10^6)$
FatTree8	Small DC	80	512	128	$\mathcal{O}(10^4)$
FatTree16	Large DC	320	4096	1024	$\mathcal{O}(10^6)$

4.4.1.2 Workloads

For each topology, we randomly produce 24 workloads using publicly available workload generators [SCP13, WSW+14], each representing the traffic in one hour. For each workload, we extract the set F of origin-destination flows together with their assigned source and destination servers. We then use the volume of a flow as its normalized value for the objective function (4.11) (i.e., $\forall f \in F, \forall l \in E(f) : w_{f,l} = p_f$). A flow $f \in F$ starts from the ingress link of the source server and asks to exit at the egress link of the destination server.

4.4.1.3 Controller placement

The controller placement and the default path towards it are two major factors influencing the allocation matrix. In the evaluation, we consider two extreme controller positions in the topology: the most centralized position (i.e., the node that has minimum total distance to other nodes, denoted by `MIN`), and least centralized



(a) Abilene (b) ScaleFree

(c) FatTree8 (d) FatTree16

Figure 4.4: Average path stretch of deflected flows

position (i.e., the node that has maximum total distance to other nodes, denoted by `MAX`). In all cases, the default path is constituted by the minimum shortest path tree from all ingress links to the controller. The most centralized position limits the default path's length and hence the number of possible deflection points. On the contrary, the least centralized position allows a longer default path and more choices for the deflection point.

4.4.1.4 Allocation algorithms

To evaluate the quality of the heuristic defined in Sec. 4.3.3, we compare it with the following two allocation algorithms:

- **Random Placement (RP)**: It is a variant of OFFICER where flow sets are randomly ranked and deflection points are randomly selected.
- **Optimum (OP)**: The allocation matrix corresponds to the optimal one as defined in Sec. 4.3.2 and is computed using CPLEX. [6] Unfortunately, as computing the optimum is NP-hard, it is impossible to apply it to large ISP and large DC topologies.

Because of room constraints, we only present results for the CE strategy to choose the deflection point. Nevertheless, with extensive evaluations, we observed that this strategy outperforms the two others by consuming less memory resources.

4.4.2 Results

In this section, we compare rule allocation obtained with OFFICER with the optimal allocation and random allocation. We also study the impact of the controller placement on the allocation. The benefit of OFFICER is identified as the amount of traffic able to strictly respect the endpoint policy while the drawback is expressed with the path stretch. We also link the number of flows passing through nodes with their topological location.

In Fig. 4.3 and Fig. 4.4, the x-axis gives the normalized total memory capacity computed as the ratio of the total number of forwarding entries to install in the network divided by the number of flows (e.g., a capacity of 2 means that on average flows consume two forwarding entries). Thin curves refer to results obtained with the controller placed at the most centralized location (i.e., MIN) while the thick curves refer to results for the least centralized location (i.e., MAX). The y-axis indicates the average value and standard deviation over the 24 workloads for the metric of interest. Curves are labeled by the concatenation of their allocation algorithm acronym (i.e., CE, RP, and OP) and their controller location (i.e., MIN and MAX).

Reference points indicate the value of the metric of interest if all flows are delivered to their egress link when (i) strictly following the shortest path and denoted with a square and (ii), if ever computable, when minimizing memory usage

[6] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

as formulated in Sec. 4.3.1 and denoted with a circle. For a fair comparison with OFFICER, we also use the aggregation with the default path for these reference points. It is worth noting that the squares are on the right of the circles confirming so that by relaxing routing policy it is possible to deliver all the flows with less memory capacity.

Fig. 4.3 evaluates the proportion of the volume of traffic that can be delivered to an egress point that satisfies the endpoint policy as a function of the capacity. In all situations, OFFICER is able to satisfy 100% of the traffic with less capacity than with a strict shortest routing policy. In addition, when the optimal can be computed, we note that OFFICER is nearly optimal and is even able to satisfy 100% of the traffic with the optimal minimum capacity. This happens because there are no link bandwidth nor per-switch memory limitations and that in our two examples flows never cross twice the default path. On the contrary, the random allocation behaves poorly in all situations and requires up to 150% more memory than OFFICER to cover the same proportion of traffic.

Also, with only 50% of the minimal memory capacity required to satisfy 100% of the traffic, OFFICER satisfies from 75% to 95% of the traffic. The marginal gain of increasing the memory is hence limited and the choice of the memory to put in a network is a tradeoff between memory costs and the lost of revenues induced by using the default path.

Relaxing routing policy permits to deliver more traffic as path diversity is increased but comes at the cost of longer paths. Fig. 4.4 depicts the average path stretch (compared to shortest path in case of infinite memory) as a function of the capacity. Fig. 4.4 shows that the path stretch induced by the optimal placement is negligible in all type of topologies and is kept small for OFFICER using the CE strategy (i.e., less than 5%). On the contrary, the random placement significantly increases path length. In DC topologies, the average path stretch is virtually equal to 1 (Fig. 4.4c and Fig 4.4d). The reason is that in DC networks there is a high diversity of shortest path between node pairs, so it is more likely to find a shortest path satisfying all constraints than in ISPs topologies. It also worth noting that in DCs, there are many in-rack communications that consume less memory than out-rack communications, thus the risk of overloading memory of inter-rack switches is reduced. Interestingly, even though there is a path stretch, the overall memory consumption is reduced, indicating that it is compensated by the aggregation with the default rule.

For ISP networks, when the optimal allocation is computed or approximated with OFFICER, there is a high correlation (i.e., over 0.9) between the memory required on a switch and its topological location (e.g., betweeness centrality and node degree). On the contrary, no significant correlation is observed in DCs where there are much more in-racks communication than out-racks communication [WSW+14]. This suggests to put switches with the highest memory capacity at the most central locations in ISPs and within racks in DCs.

Even though the controller placement is important in OFFICER as it leverages the default path, Fig. 4.3 and Fig. 4.4 do not exhibit a significant impact of the

location of the controller. Nevertheless, no strong conclusion can be drawn from our evaluation. Actually, there are so many factors that drive the placement of the controller [HSM12] that we believe it is better to consider controller placement as an input of the rule allocation problem and we let its full study for future work.

4.5 Discussion

With this section we provide a broad discussion on the model presented in Sec. 4.2 as well as the assumptions that drove it.

4.5.1 Routing Policy

Relaxing routing policy allows better usage of the network but comes with the expense of potential high path stretch. Nevertheless, nothing prevents to add constraints in our model to account for a particular routing policy. For example, the constraint $\forall f \in F : \sum_{l \in L^+} a_{f,l} \leq \alpha(f)$ can be added to control the maximum path length of each flow. This constraint binds the path length to an arbitrary value pre-computed by the operator, with $\alpha(f) : F \to \mathbb{R}$. For example, $\alpha(f) = h \cdot shortest\_path\_length(f)$ to authorize a maximum path stretch h (e.g., $h = 1.5$ authorizes paths to be up to 50% longer than the corresponding shortest paths).

4.5.2 Rule Aggregation

To aggregate two rules having the same forwarding action into one single rule, a common matching pattern must be found between the two rules. Constraints (4.5) and (4.6) provide a first step towards rules aggregation: on a switch, if the forwarding decision for a flow is the same as the default action, the rule for the flow does not need to be installed. However, a problem occurs when the common matching pattern also matches for another rule that has a different action. The latter rule should not be covered by the aggregating rule as that could create loop events or incorrect forwarding. Consequently, the construction of the minimal set of rules in a switch by using aggregation requires the knowledge of the allocation matrix that, in turn, will be affected by the aggregation. This risk of non-linearity is a reason why we assume that one forwarding rule is used for at most one flow and why we limit aggregation to the default rule only.

4.5.3 Multipath

The model presented in Sec. 4.2 assigns one forwarding path per flow. As a result, all the packets of a flow follow the same path to the egress link, which ensures that packet arrival order is maintained. Nevertheless, our model does not prevent multipath routing. To do so, the pattern matching of a flow to be forwarded on several paths must be redefined from the one used in case of one forwarding path. From a network point of view, the flow will then be seen as multiple flows, one per matching pattern. Consequently, the optimizer might give different forwarding

paths for packets initially belonging to the same flow. For example, one can assign a label to packets when they enter the network and then use labels to decide to which rule the packet matches. This may increase significantly the number of rules to be installed in the network and the gain of having several such paths must be compared to the cost of having them. In most situations, multipath routing at the flow level might not be necessary as we are not enforcing any routing policy in our model, which limits the risk of having the traffic matching one rule to be enough to saturate one link.

4.5.4 Related Work

Rule allocation in OpenFlow has been largely covered over the last years. Part of the related work proceeds by local optimization on switches to increase their efficiency in handling the installed rules. The other part, which is more relevant to our work, solves the problem network-wide and produces a set of compressed rules together with their placement. Our present research builds upon this rich research area and presents an original model, together with its solution, for the rule allocation problem where the routing can be relaxed for the only objective of placing as many as rules as possible that respect the predefined endpoint policy.

For the first part, several mechanisms based on wildcard rules have been proposed to minimize the rule space consumption on switches as well as to limit the signaling overhead between switches and controller. DevoFlow [CMT+11] uses wildcard rules to handle short flows locally on switches. DomainFlow [NHL+13] divides the network into one domain using wildcard rules and another domain using exact matching rules. SwitchReduce [IMS13] proposes to compress all rules that have the same actions into a wildcard rule with the exception of the first hop switch.

To reduce further memory usage, latest versions of OpenFlow support pipelining and multi-level flow tables [Ope15b]. Consequently, the large forwarding table is split in a hierarchy of smaller tables that can be combined to build complex forwarding rules with less entries. However, even though these techniques improve memory usage, they do not remove the exponential growth of state with the number of flows and nodes in the network.

As for the second part, some works suggest to use special devices to perform rule placement. DIFANE [YRFW10] places the most important rules at some additional devices, called authority switches. Then, ingress switches redirect unmatching packets towards these specific devices, which enables reducing load on the controller and, at the same time, decreasing the number of rules required to be stored on ingress switches. vCRIB [MYSG12] installs rules on both hypervisors and switches to increase performance while limiting resource usage. Other works optimize rule allocation on switches themselves. Palette [KHK13] and OneBigSwitch [KLRW13] produce the aggregated rule sets that satisfy the endpoint policy and place them on switches while respecting the routing policy and minimizing the resources. However both Palette and OneBigSwitch cannot be used in scenarios where resources are missing to satisfy the endpoint policy. In [GMP14], the rule allocation is modeled as a

constrained optimization problem focusing on the minimization of the overall energy consumption of switches. Finally, the authors in [NSBT14] propose a network-wide optimization to place as many rules as possible under memory and link capacity constraints.

While the related works presented above focus on particular aspects of the rule allocation problem in OpenFlow, with OFFICER we propose an original and general solution that is able to cope with endpoint and routing policies, network constraints, and high-level operational objectives.

4.6 Conclusion

In this chapter, we present a new algorithm called OFFICER for offline OpenFlow Rules Placement Problem. Starting from a set of endpoint policies to satisfy, OFFICER respects as many of these policies as possible within the limit of available network resources both on switches and links. The originality of OFFICER lies in its capacity to relax the routing policy inside the network for the objective of obtaining the maximum in terms of endpoint policies. OFFICER is based on an Integer Linear Programming model and a set of heuristics to approximate the optimal allocation in polynomial time. The gain from OFFICER was shown by numerical simulations over realistic network topologies and traffic traces.

Chapter 5

Adaptive Rules Placement in OpenFlow Networks

Contents

5.1 Introduction 57
5.2 aOFFICER: Adaptive OpenFlow Rules Placement 59
5.2.1 Objectives 59
5.2.2 Design 60
5.2.3 Adaptive Threshold 61
5.3 Evaluation 66
5.3.1 Setup 66
5.3.2 Adaptive Threshold 69
5.3.3 Adaptive Timeout and Deflection Technique 73
5.4 Conclusion 77

In Chapter 4, we study the *offline* OpenFlow Rules Placement Problem (ORPP), in which the set of traffic flows is known (e.g., Source-Destination Flows) and stable in a period. To solve that problem, we propose an Integer Linear Programming and a greedy-based heuristic called OFFICER, that can decide the most profitable rules satisfying policies and network constraints.

However, flows are often not known in advance and are unpredictable, because of measurement errors. Therefore, the proposed solutions can not be directly applied in this case. In this chapter, we study and find solutions for the *online* ORPP, in which the set of flows is unknown and varies over time.

5.1 Introduction

As discussed in Chapter 3, there are two main approaches to place rules: *Proactive*, i.e., placing rules in advance, before the first packet of a new flow arrives [KHK13, KLRW13, ZIL+14, KARW14], or *Reactive*, i.e., placing rules on demand [GKP+08, Flo15, Eri13, Ryu15, Ope15a].

The *proactive* approach nullifies the setup delay, reduces the signaling overhead, and is commonly used to implement access control applications [KHK13, KLRW13, ZIL+14, KARW14], as access control rules are predefined independently of the traffic.

For other applications, forwarding rules can also be pre-installed, but flows must be known in advance [GMP14, LLG14, HLGY14, WWJ+15].

In practical situations, flows are unknown and unpredictable [BAAZ11]. With *reactive* approach, rules are populated on demand and continuously updated to cope with traffic fluctuations. For example, new coming flows require new forwarding rules, whereas a down link event requires updating rules to reroute all the affected flows. Using this approach, controller platforms [GKP+08, Flo15, Eri13, Ryu15, Ope15a] can install rules to respond a wide range of network events. However, they have several limitations, as we explain in the following.

First, these controllers treat all flows equally, and install rules for all incoming flows, from the first packet seen. However, this method causes high setup delay, high signaling overhead because a large number of new flows may arrive (e.g., 100k new flows/s in a cluster [KSG+09]). As consequences, the controller and the switch buffer can be overloaded, and packets are dropped. Furthermore, the largest flows (i.e., flow that sends many packets, often called elephant flows) may not be installed because the switch memory is occupied by other flows. Managing large flows by dedicated OpenFlow rules is important for traffic engineering goals [CMT+11], since they account for a large part in the total traffic load.

Second, these controllers install rules along the shortest paths from the source to the destination (e.g., computed using OSPF [Moy89], ECMP [Hop00]). In some cases, this approach is necessary to obtain the required performance (e.g., throughput, latency). However, resources on other paths are not leveraged, as we explained and illustrated in Chapter 4.

We argue that when resources (e.g. switch memory) are limited, only rules for important, large flows should be installed. On one hand, it is not necessary to install rules for small flows, because rule placement decisions for small flows contribute little to the global performance. On the other hand, managing the largest flows is important for traffic engineering (e.g., rate limitation, accounting), since they account for most of traffic load. Based on this observation, we propose an adaptive rules placement framework, called aOFFICER, that can detect the important, large flows and install rules for them on appropriate paths. Furthermore, aOFFICER reuses the deflection technique proposed in Chapter 4, to select the paths consuming less switch memory than other paths. We then conduct packet level simulations for aOFFICER in realistic scenarios. Simulation results show that aOFFICER can reduce signaling overhead significantly compared to existing solutions, and it can adapt the parameter to cope with traffic fluctuations.

The content of this chapter is organized as follows. In Sec. 5.2, we describe the objectives and the design of aOFFICER. In Sec. 5.3, we evaluate, compare aOFFICER to other solutions, and discuss the obtained results. Finally, in Sec. 5.4, we draw conclusion remarks.

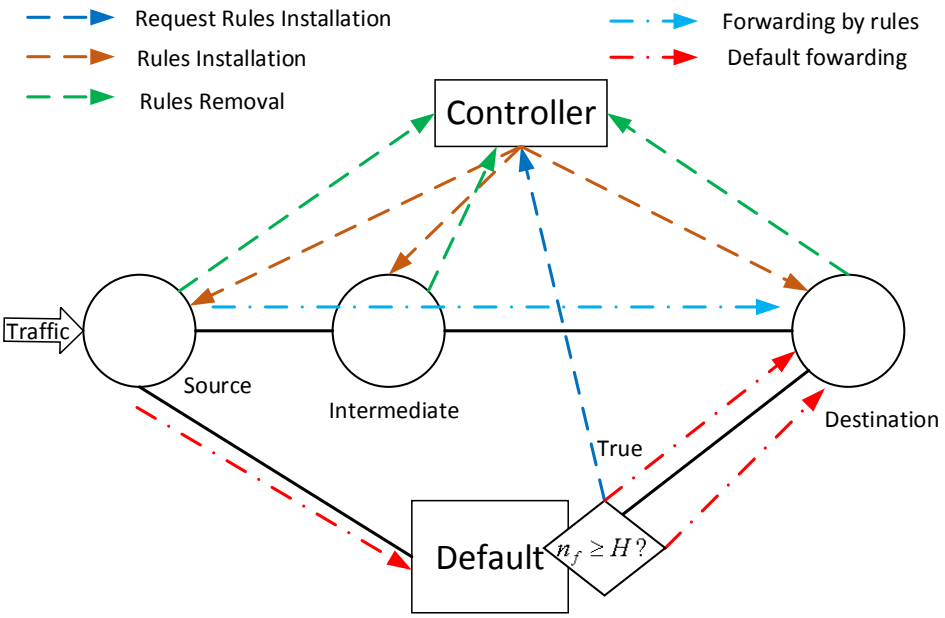


Figure 5.1: Problem Statement

5.2 aOFFICER: Adaptive OpenFlow Rules Placement

5.2.1 Objectives

In Fig. 5.1, we abstract the scenario that aOFFICER aims to solve. The network is composed of three kinds of components: controllers, commodity switches (e.g., with TCAMs), default devices (e.g., software switches with large non-TCAM memory). The network traffic is represented as flows, and each flow consists a sequence of packets matching a pattern. A flow requires rules on intermediate switches to reach its destination. On new flow arrival, switches send *request* messages to the controller, and rules are installed by the controller using *rules installation* messages. On rule removal (due to rule timeout), the switch send back a *rules removal* message.

Because of the memory limitation [SCF+12], some flows may not have matching rules and they are forwarded by default rules, towards default devices for further processing (i.e., *default forwarding*). A flow is called *installed* if it is *not* processed by default devices. Normally, packets processed by default devices suffer from performance penalties, e.g., high lookup latency in non-TCAM memory, so it is necessary to minimize the default devices' load.

aOFFICER is implemented as a plug-in on the controller platforms (e.g., OpenDayLight [Ope15a]). Basically, aOFFICER supports the controller to find rules satisfying the following requirements:

1. Enforcing policies from operators such as endpoint policy, which defines where

the packets should go, e.g., peering links, gateways, firewalls.

2. Enforcing network constraints, such as switch memory, link capacity, controller capacity.

3. the default load (i.e., load on default devices) and the signaling load (i.e., load on the controller) are minimized. On one hand, default devices (e.g., software switches) usually cause high lookup delay, because of using the non-TCAM memory. Therefore, the traffic volume processing by default devices needs to be minimized. On the other hand, minimizing the signaling load increases the network scalability (e.g., more nodes and flows can be managed).

5.2.2 Design

Since flows are not known in advance and varies over time, aOFFICER uses the reactive approach to place rules enforcing policies and constraints. aOFFICER involves in two important decisions, including how rules are chosen and how rules are installed.

First, some flows are important and worth to install than others, e.g., flows from delay sensitive applications (e.g., Voice over IP, Video on Demand), or flows from premium users. Large flows (i.e., flow that sends many packets or bytes) are also important, since they account for most of the traffic load. These flows need to be managed using OpenFlow rules (e.g., rate limitation, accounting) and forwarded on appropriate paths. For example, the controller installs rules to forward them on the least congested links and to avoid default devices. In the rest of the chapter, we focus in a challenging case, in which flows are unpredictable, so aOFFICER must detect which flow is potentially large and worth to be installed.

Installing rules for a flow from the first packet seen, like implemented in [Flo15, Eri13, GKP+08], is not efficient because most of the flows send just few packets, according to traffic measurement studies [BAM10, KSG+09]. Departing from the state of the art, aOFFICER only installs rules for a flow if that flow is potential large, by using traditional large flow detection techniques, such as packet sampling. By using a reasonable number of samples, large flows can be identified quickly with very low error [CPZ04]. In aOFFICER, at the beginning, all flows must use default rules to reach default devices. Default devices act as center hubs for all flows, and they use large flow detection mechanisms (e.g., using the mechanism proposed in [CMT+11]) to detect potential large flows and to notify the controller. Large flow detection can be implemented on ingress switches, however, the memory space limits the number of flows it can monitor.

A flow f is labeled as "worth to install", if it sends H packets to a default device. $H \in \mathbb{Z}^+$ is a configurable parameter called the *threshold*, and can be modified by aOFFICER, e.g., through the OFConfig protocol [Fou]. Furthermore, in aOFFICER, H is adjusted according to traffic fluctuations. For example, when traffic demands are high, H is large, so only very large flows can be installed.

After f is marked as "worth to install", aOFFICER verifies available resources (e.g., switch memory and link capacity) to check if f can be installed. If yes, rules for f are installed along the shortest paths from the source to the destination of f, or along the paths selected by deflection techniques (Chapter 4). The subsequent packets of f are forwarded by these rules, without passing via default devices. If it is not possible to install rules for f, aOFFICER ignores the request. Therefore, packets of f continue to be processed by default devices, and default devices continue sending requests for each new packet of f. In this manner, the large flow will eventually be installed because of many requests.

The advantage of using aOFFICER is multifold. First, the signaling load is reduced, because the controller receives less requests than other approaches. Second, large flows have a higher installation probability than others, because more requests are sent. Third, the penalty caused by default devices can be reduced, as only mice flows with few packets are processed by default devices. Forth, aOFFICER automatically adjust its parameter H according to traffic fluctuations, to use resources more efficiently. A drawback of using aOFFICER is that flows need to wait longer before getting installed than using traditional approaches. Another drawback is that flows may follow a long path, because of the deflection technique.

In the following section, we explain the adaptive threshold mechanism used in aOFFICER to adjust H according to traffic fluctuations. We present a formalization for the problem of selecting the optimal H (Sec. 5.2.3.1), and an adaptive algorithm to adjust H (Sec. 5.2.3.2).

5.2.3 Adaptive Threshold

5.2.3.1 Problem formalization

As motivated in the previous section, H should be adapted according to traffic fluctuations. To that goal, we first formalize the problem of selecting H to minimize the cost, using the notations in Table 5.1.

Table 5.1: Notations

Notation	Meaning
H	Threshold determined when to notify the controller ($H \in \mathbb{Z}^+$)
T	Interval (s)
F	Set of flows f in T
n_f	Number of packets of flows f in T
r	Success installation probability ($0 \leq r \leq 1$)
a_f	Number of packets of f sending to the default device
b_f	Number of request packets of f sending to the controller
d_f	Number of rejected packets of f
p_f	Installation probability of f
β_f	Number of signaling messages needed to install f

The network traffic is modeled as a set of flows F, each flow $f \in F$ sends n_f packets in the period T. The threshold $H \in \mathbb{Z}^+$ determines when the request is sent to the controller. If the request can be satisfied, the controller installs rules for f. For sake of simplicity, we assume that once the flow f is installed, rules for f remains in the rest of the interval T.

Because of resource limitations, some requests can not be satisfied. Therefore, we denote $0 \leq r \leq 1$ as the average success installation probability during T. We adapt H according to r, since r reflects the current traffic load and can easily be measured by the controller.

aOFFICER is designed to minimize two metrics, which are the the default load (i.e., the load on the default devices) and the signaling load (i.e., the load on the controller). Both metrics are expressed in terms of number of packets. There is a trade-off between two metrics. For example, if more flows are installed, the default load reduces but the signaling load increases. Therefore, we aim to minimize the total of two metrics, called total cost, to find the best trade-off between them.

To formalize the cost as a function of (r, H), for each flow, we estimate the number of packets seen at the default devices (denoted as a_f), the number of request packets (denoted as b_f), the installation probability (denoted as p_f), and the number of rejected packets (denoted as d_f).

For the flow f with $n_f < H$, all packets of f arrive to the default devices and there is no request for f, and therefore, $a_f = n_f, b_f = 0, d_f = 0$.

For the flow f with $n_f \geq H$, $H \leq a_f \leq n_f$. To estimate a_f, we compute the probability of the event $a_f = k$, denoted as $\mathbb{P}(a_f = k)$ $(H \leq k \leq n_f)$. Basically, $a_f = k$ $(H \leq k < n_f)$ means that there are $k - H + 1$ requests for f, in which requests with id from $(0 \rightarrow k - H - 1)$ are not satisfied, and the request with id $k - H$ is satisfied. The success installation probability of each request is r. Therefore, we derive the following:

$$\mathbb{P}(a_f = k) = (1 - r)^{k-H} \cdot r \qquad (H \leq k < n_f) \tag{5.1}$$

For the case $a_f = n_f$, there are two possibilities: the last request for f is satisfied, or all requests for f are not satisfied, therefore:

$$\mathbb{P}(a_f = n_f) = (1 - r)^{n_f - H} r + (1 - r)^{n_f - H + 1} = (1 - r)^{n_f - H} \tag{5.2}$$

By using the law of the total probability:

$$a_f = \sum_{H}^{n_f} \mathbb{P}(a_f = k) \times k \tag{5.3}$$

In summary, a_f is expressed as the following:

$$a_f = \begin{cases} n_f & ; n_f < H \\ \sum_{i=H}^{n_f - 1} i \cdot r \cdot (1 - r)^{i-H} + (1 - r)^{n_f - H} \cdot n_f & ; n_f \geq H \end{cases} \tag{5.4}$$

As designed in aOFFICER, after sending $H-1$ packet, a request for f is sent for each new packet of f, until f is installed, so b_f is expressed as the following:

$$b_f = \begin{cases} 0 & ; n_f < H \\ a_f - H + 1 & ; n_f \geq H \end{cases} \tag{5.5}$$

A flow f with $n_f \geq H$ will not be installed if all $(n_f - H + 1)$ requests for f are rejected, the probability of that possibility is $(1-r)^{n_f - H + 1}$. Therefore, the installation probability of f is $p_f = 1 - (1-r)^{n_f - H + 1}$. To summarize:

$$p_f = \begin{cases} 0 & ; n_f < H \\ 1 - (1-r)^{n_f - H + 1} & ; n_f \geq H \end{cases} \tag{5.6}$$

d_f represents the number or rejected requests, is computed based from b_f. If the flow f is installed, $d_f = b_f - 1$. Otherwise, $d_f = b_f$. Using the law of total probability, we derive that:

$$d_f = (1 - p_f) b_f + p_f (b_f - 1) = b_f - p_f \tag{5.7}$$

The total cost C includes the default load C_d and the signaling load C_s. The default load C_d is computed based on a_f:

$$C_d = \sum_{f \in F} a_f \tag{5.8}$$

To compute the signaling load C_s, we denote β_f as the number of signaling messages needed to install for flow f. For example, assuming that the installation path length for f is n. The controller need to process 1 request message, to generate n rule installation messages, and to process n removal messages (because of rules timeout), thus $\beta_f = (1 + 2n)$. Therefore, for each flow f, the controller needs to process $d_f + \beta_f p_f$ messages.

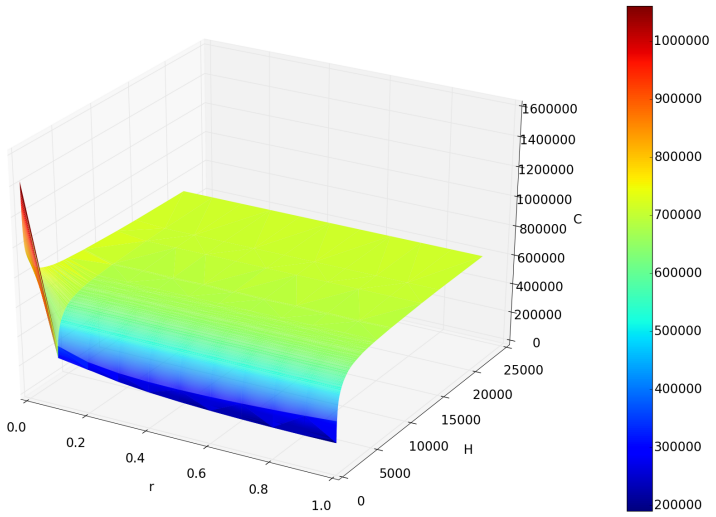
Finally, the total cost can be written as:

$$C = C_d + C_s \tag{5.9}$$

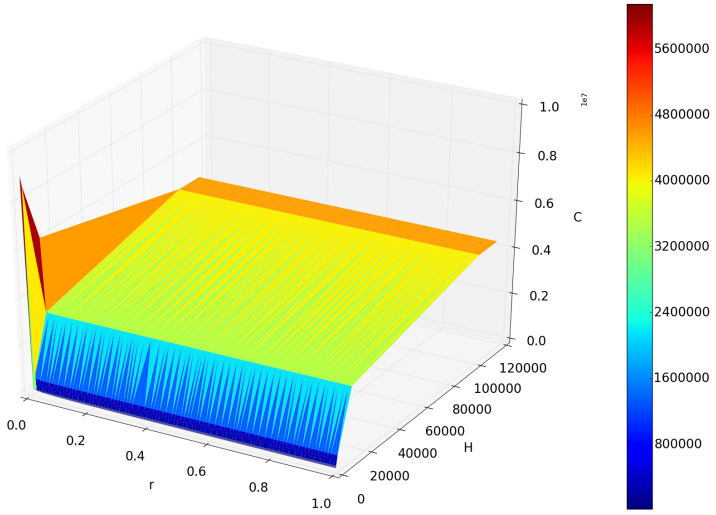
$$C = \sum_{f \in F} (a_f + \beta_f p_f + d_f) \tag{5.10}$$

By replacing the values of a_f, b_f, d_f as functions of (r, H) into Eq. 5.10, we establish the relationship between C and (r, H). Trivial results can be derived from these above formulations. From the equation 5.4, when $r \to 0$ (e.g., high demands), $a_f = n_f, \forall f \in F$. When $r \to 1$, $a_f = H, \forall f$ with $n_f \geq H$. From Eq. 5.6, if H increases, less number of flows is installed, and the flow f with larger n_f has a higher installation probability.

To better understand the relationship between r and H, we compute C with different values of r, H, using an example value $\beta_f = 5$ and flow size distributions n_f of two data center traces EDU1 and Hadoop (Table 5.2 and Fig. 5.3). We will discuss these traces later in Sec. 5.3.1.



(a) EDU1



(b) Hadoop

Figure 5.2: Total cost with different values of r and H

The numerical results are represented on Fig. 5.2. In both cases, there is a high cost zone to be avoided, which is $(r \to 0, H \to 1)$. This zone reflects the fact that when r is small, installing rules for all flows must be avoided, because it incurs a high signaling load, while most of the requests are rejected. Therefore, in that case, H must be adjusted to high values, in order to reduce the number of requests. In case where $r \to 1$, H should be decreased *exponentially*, so more flows can be installed, to reduce the default load. These numerical results motivate the need to use the adaptive control for H based on r.

5.2.3.2 Adaptive Threshold Algorithm

In the previous section, we model the cost C as the function of (r, H). In practice, the cost function is much more complex, due to timeout effects and the traffic fluctuations. However, the high cost zone $(r \to 0, H \to 1)$ still exists and must be avoided, since the controller receives many requests, but it cannot satisfy all of them.

Basically, r reflects the traffic load, and depends on many factors, such as number of new flows, available resources, timeout effects. To escape the high cost zone, it is important to maintain r at high values (e.g., $r = 0.9$), by adjusting H. To that goal, we propose an adaptive algorithm, inspired from TCP Congestion Avoidance [APB09], that adjust H to avoid the high cost zone.

First, we use Exponentially Weighted Moving Average (EWMA) model [KO90] to estimate r_n, which is the average success installation probability at step n, every time when controller receives a request. EWMA is used because it is proved useful to predict the long term trend of a time series, and it accounts for the weight of old values into current values. Basically, r_n is computed recursively as follows:

$$r_n = \alpha r_{n-1} + (1 - \alpha) z_n \tag{5.11}$$

In Eq.5.11, z_n is the result of the installation trials at the controller (1 means that the request is satisfied, 0 means that the request is not rejected); α is the parameter from EWMA model that indicates the important of history values in the expected value $(0 < \alpha \leq 1)$ [KO90]; r_{n-1} is the average success installation probability at step $n-1$; r_0 is the desired expected value of r (e.g., 90%).

Second, to adjust H, we implement the Multiplicative Increase/Multiplicative Decrease (MIMD) algorithm (Algo. 2), a feedback control algorithm that is well known for its simplicity and efficiency, especially in TCP congestion control [APB09]. In Algo. 2, I, D are the increasing and decreasing factors (e.g., $I = 2, D = 2$), $round()$ is the function that rounds a real number to the nearest integer. We use MIMD because it quickly converges in our scenarios, as we will show in Sec. 5.3.

When $r < r_0$, H increases exponentially (Line 2), to reduce the number of requests and to avoid from the high cost zone. As a result, the controller only needs to satisfy a lower number of requests with the same amount of resources, so r is improved. On the contrary, when $r > r_0$, H decreases exponentially to $round(H/D)$, so more flows can be installed (Line 4), and the default load can be reduced further. Moreover, H is bounded by H_{min} (Line 6).

Algorithm 2: Adaptive Threshold

Require: $H_{old} \in \mathbb{Z}^+$: current value of H

1: **if** $r < r_0$ **then**
2: $H \leftarrow H_{old} * I$
3: **else**
4: $H \leftarrow round(H_{old}/D)$
5: **if** $H < H_{min}$ **then**
6: $H \leftarrow H_{min}$
7: **return** H

Note that there is a case where H remains at very high value, so no request is sent and flows can not installed. To avoid this case, we implement an auto-decrement mechanism for H. If there is no request within the *interval*, H decreases to $round(H/D)$, to allow more requests are sent and more flows can be installed.

Parameters (I, D, H_{min}) are selected empirically with traces, and they affect to the convergence speed, size of oscillations and the possible values of H. In Sec. 5.3, we will perform simulations for different values of (I, D, H_{min}).

5.3 Evaluation

In this section, we evaluate and compare aOFFICER to existing algorithms [Flo15, GKP+08, Eri13] and OFFICER (Chapter 4) in realistic scenarios.

5.3.1 Setup

Our scenarios are machine-to-machine communications in data centers. An operator wants to deploy the forwarding rules for his data center to manage flows between servers, such that endpoint policies and network constraints are satisfied. Basically, a flow is a sequence of packets having the same five tuples (source/destination IP address, source/destination port, the protocol number).

In this data center, all switches support OpenFlow protocol [Ope15b], and each switch can store at most 1000 rules [SCF+12]. As usual, on each switch, there is a default rule that directs non-matching packets towards a default device. The default device is a software OpenFlow switch that can store a large number of rules, but it has a higher processing delay than commodity switches. The default device is attached to the most centralized switch (i.e., the average distance from this switch to all other switches is the minimum), as setup in Chapter 4. The OpenFlow controller controls all switches including the software switch, via a dedicate control network, so the traffic network is not influenced.

To evaluate and compare aOFFICER to different algorithms, we implement a Python-based simulator that replays the packet trace, and simulates behaviors of OpenFlow switches, threshold-based detection, and timeout mechanisms. We are aware of existing network simulators, such as NS-3 [HLR+08] but at this moment, it

Table 5.2: Packet traces summary

Trace	Duration (s)	# Packets	# Unique flows	Topology
EDU1	300	714159	55543	fat tree, 22 switches
Hadoop	300	4563784	2341	fat tree, 80 switches

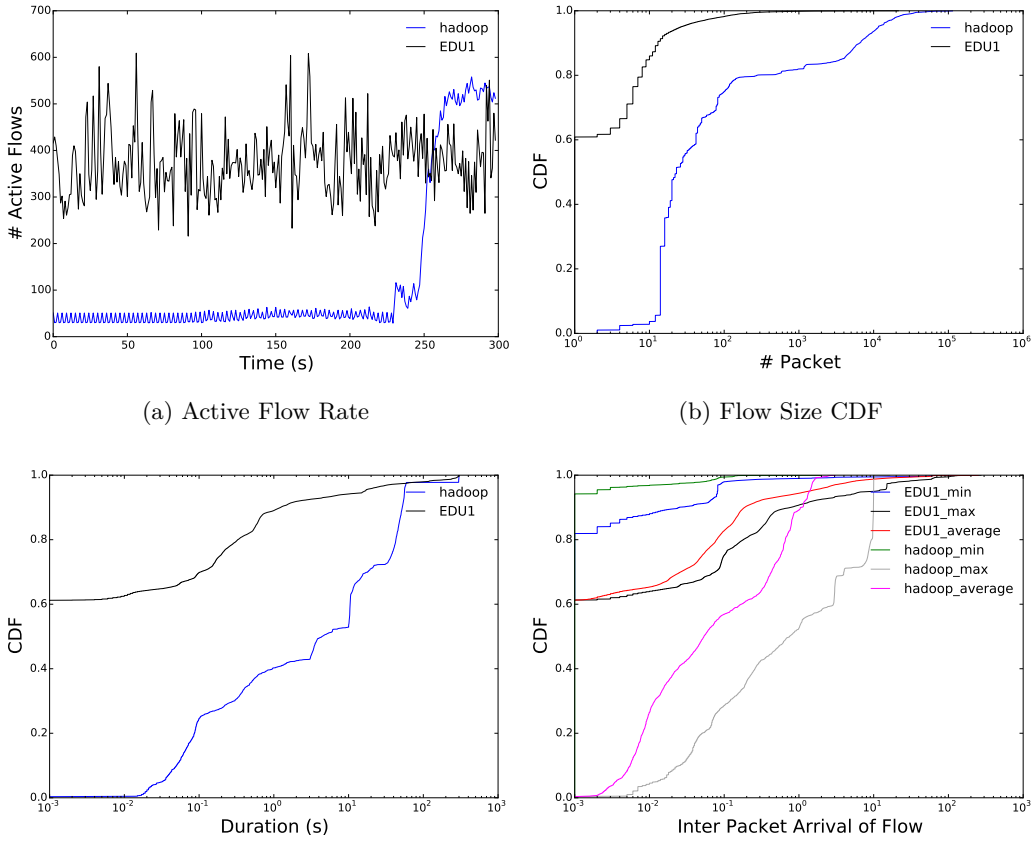
does not fully support the OpenFlow protocol. For the sake of simplicity, we assume that all links are over-provisioned, so the link constraints can be omitted and there is no packet loss. Also, in OpenFlow networks, the memory constraints are often stricter than bandwidth constraints. All simulations are performed on the INRIA NEF Cluster [INR15].

We use two datasets: (*i*) EDU1: a packet trace and a fat tree topology from a public university data center [BAM10]; (*ii*) Hadoop: a trace and a fat tree topology from our emulated data center running Hadoop[Apa15]. To generate this dataset, we emulate a data center on Grid5000 [gri15] using our tool DiG [Son15], deploy Hadoop applications, and capture the packet traces from all nodes. Due to a large number of packets in Hadoop, we use flow sampling technique with rate 10% to reduce the size of the trace, while preserving flow size distribution. Two datasets give us more insights on the performance of different algorithms. For the Hadoop trace, we know exactly which IP prefix is attached to which edge switch. For the trace EDU1, this information is not given, so each IP prefix in the trace is assigned randomly to an edge switch.

The trace details are summarized in Table 5.2 and Fig. 5.3. Generally, two traces have same duration (300s), but they are different in characteristics. EDU1 has more unique flows, but less number of packets than in Hadoop. The number of active flows per second (i.e., flows that send packets within that second) in EDU1 varies in range $[200 - 600]$, while in Hadoop the number of active flows is stable at the beginning and then increases significantly around $t = 250s$ (Fig. 5.3a). As observed in Fig. 5.3b, 80% of flows in EDU1 send less than 10 packets, while in Hadoop, only 2% of flows send less than 10 packets. Moreover, most of the flows in EDU1 are short live (e.g., lifetimes of 80% of flows are less than $1s$), as shown in Fig. 5.3c. Fig. 5.3d shows the CDF function for min, average, and max inter packet arrival (i.e., the difference in timestamps of two successive packets in a flow) of all flows. In EDU1, the average inter-packet arrival of 80% flows is less than 0.1s, while in Hadoop, the average inter-packet arrival of 80% flows is less than 1s. Therefore, in general flows in EDU1 need a smaller timeout than flows in Hadoop.

The following algorithms are implemented on the simulator:

- NAIVE: the algorithm used in [Flo15, GKP+08, Eri13], in which rules are reactively installed with timeout $t = 5$(s) along the shortest path, from the first packet seen.
- aOFFICER: Rules are reactively installed for flows that send more H packets. H is adjusted using Algo. 2 (labels $H = a(I, D, H_{min})$). In all cases, $r_0 = 0.9$, and $\alpha = 0.9$ (Eq. 5.11).



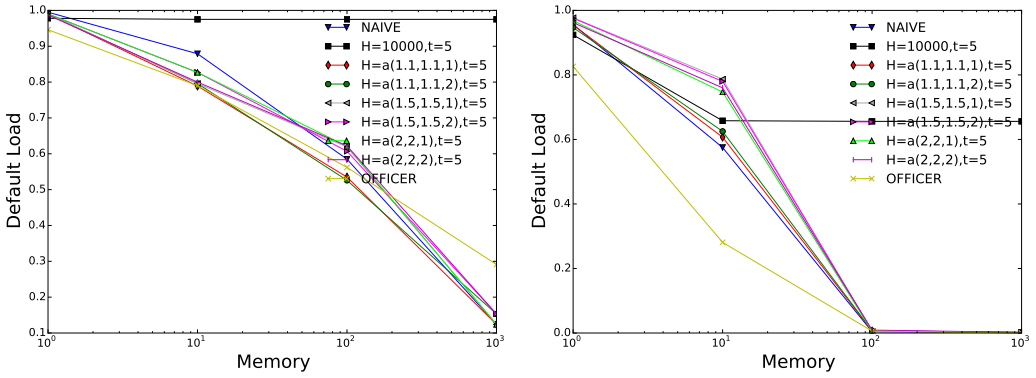
(a) Active Flow Rate

(b) Flow Size CDF

(c) Flow Duration CDF

(d) Inter Packet Arrival CDF

Figure 5.3: Trace Analysis



(a) EDU1 (b) Hadoop

Figure 5.4: Simulated Default Load of different rule placement algorithms

- OFFICER: Rules are pre-computed and placed using OFFICER (Chapter 4), with the assumption of knowing all flows in advance.

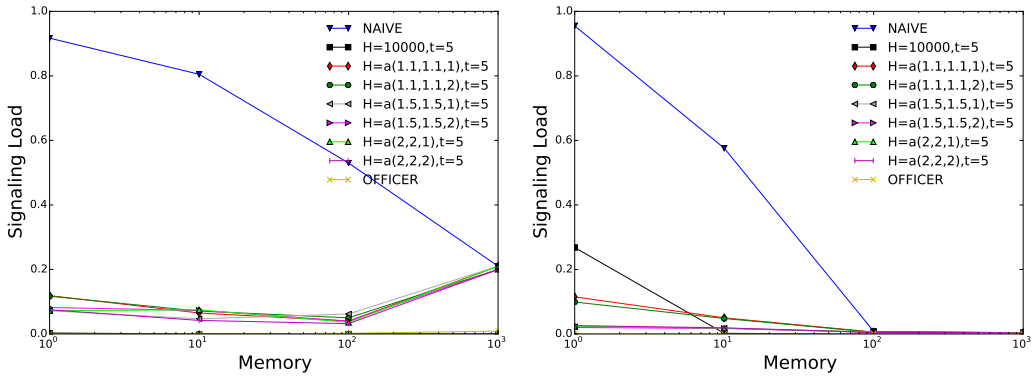
During the experiments, we are interested in the following metrics:

- Signaling load C_s: the number of signaling packets exchanging between the default devices, commodity switches and the controller. Signaling load includes rule request messages, rule installation messages, and rule timeout messages.
- Default load C_d: the number of packets processed by the default devices.
- Total load $C = C_s + C_d$: the sum of the default load and the signaling load, which we aim to reduce.
- Average Path Stretch for all packets: For each packet, we have the path stretch, which is the fraction of the actual path length and the shortest path length of that packet. In aOFFICER, packet might follow a longer path than the shortest path, because of the deflection technique proposed in Chapter 4. This metrics measures how long the path is on average.

5.3.2 Adaptive Threshold

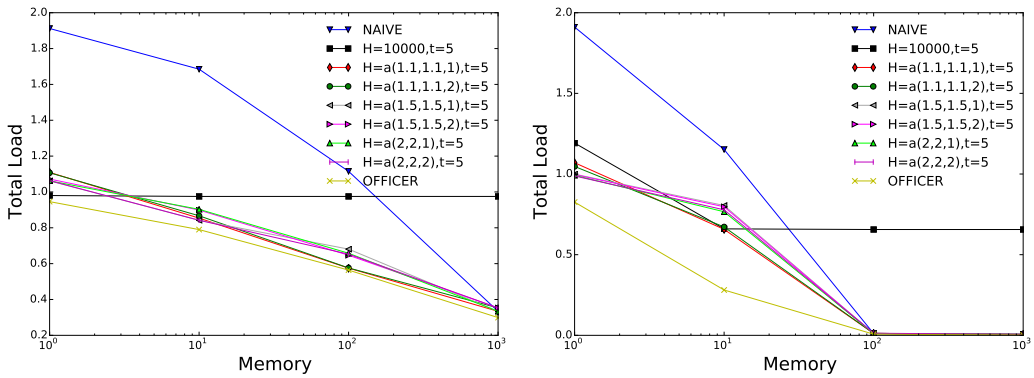
In this section, we evaluate the gain of the adaptive threshold mechanism used in aOFFICER. To focus on the impact of the adaptive threshold mechanism, we use the idle timeout $t = 5s$ and the shortest path routing (ECMP) in aOFFICER, as implemented in popular controller platforms [Flo15, GKP+08, Eri13].

In Fig. 5.4, 5.5 and 5.6, we present the simulated default load C_d, the signaling load C_s, and the total load C of different rule placement algorithms, for the traces EDU1 and Hadoop. In all figures, the *x-axis* represents the memory capacity of each switch, while the *y-axis* shows the load in terms of number of packets, and normalized



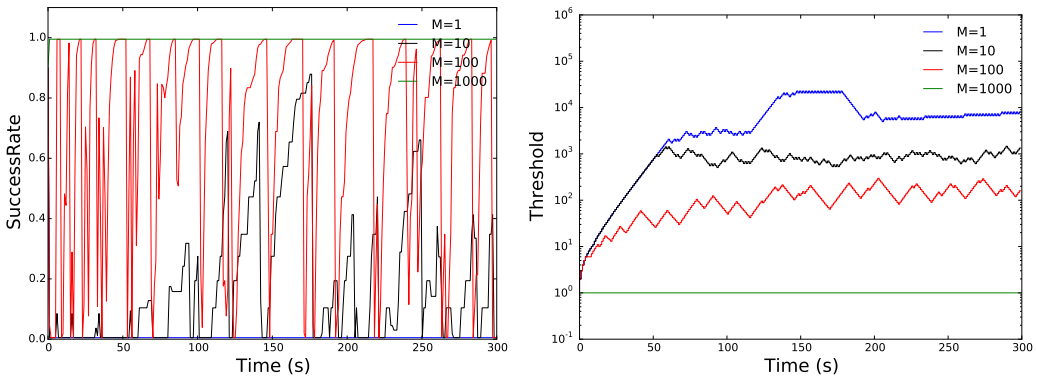
(a) EDU1 (b) Hadoop

Figure 5.5: Simulated Signaling Load of different rule placement algorithms



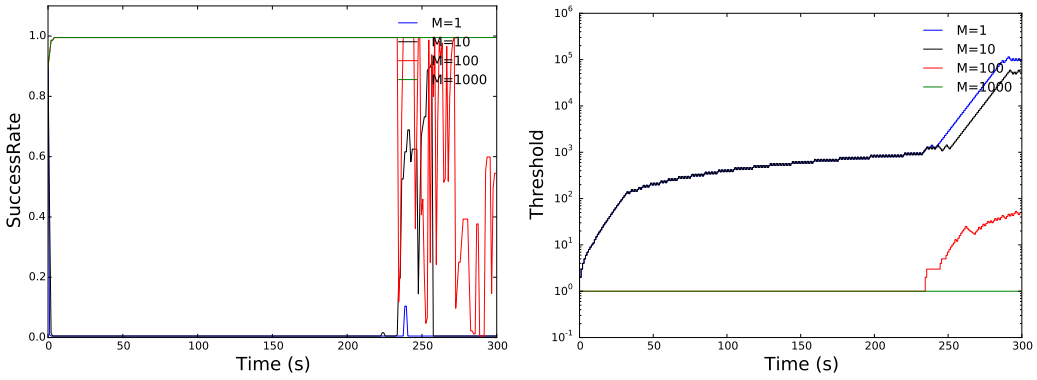
(a) EDU1 (b) Hadoop

Figure 5.6: Simulated Total Load of different rule placement algorithms



(a) r convergence (b) H convergence

Figure 5.7: Convergence of r and H for $H = a(1.1, 1.1, 1)$ with EDU1 trace



(a) Success Rate r convergence (b) Threshold H convergence

Figure 5.8: Convergence of r and H for $H = a(1.1, 1.1, 1)$ with Hadoop trace

with the traffic load. For example, the default load equals to 1 means that all packets are processed by the default device. A curve (e.g., NAIVE) represents the load for the corresponding algorithms under different memory capacity ($M = 1, 10, 100, 1000$).

Using the proactive approach, OFFICER reduces significantly the signaling load (no timeout messages, no request messages), compared to NAIVE and aOFFICER. However, to use OFFICER, flows must be known in advance.

When the switch memory is scarce ($M < 1000$), NAIVE causes a high signaling load (Fig. 5.5a and Fig. 5.5b). Moreover, the default load caused by NAIVE is not significantly different with others (Fig. 5.4a and Fig. 5.4b). This observation confirms that installing rules for all flows from the first packet is not a good strategy, as it incurs a high signaling load. More precisely, many requests are sent, while most of them are not satisfied due to memory limitations. By setting $H = 10000$, only very large flows can send requests, therefore the signaling load can be reduced greatly, but the default load is not reduced (Fig. 5.4a).

aOFFICER reduces the signaling load significantly compared to NAIVE (Fig. 5.5), by adapting H according to traffic fluctuations. In the case EDU1, aOFFICER is comparable with OFFICER when there is few large flows, in terms of the total load (Fig. 5.6a). In the case Hadoop, the total load of aOFFICER is smaller than NAIVE's, but is higher than OFFICER's (Fig. 5.6b). The reason is that there are many large flows in Hadoop trace, so aOFFICER cannot decide easily which flows should be installed.

Different parameters (I, D, H_{min}) affect to the convergence for H and the total load caused by aOFFICER, but the difference is not really significant among our selected parameters. When the switch memory is very scarce ($M = 1$), a very large H is required so only very large flows can be installed. In this case, large values of (I, D) can increase H faster, and therefore using large values (I, D) is better than using small values, as confirmed in Fig. 5.6a. However when the switch memory is bigger ($M = 10, 100$), small values I, D allow to scan more values for H, so aOFFICER can find a better H and outperform others (Fig. 5.6a and Fig. 5.6b).

Fig. 5.7 and Fig. 5.8 show the convergence of r and H over time under different memory capacities, for a particular case $H = a(1.1, 1.1, 1)$. In these figures, the *x-axis* represents the simulation time, while the *y-axis* shows the value of r (Fig. 5.7a) and H (Fig. 5.7b). For example, a curve ($M = 100$) shows the convergence of r and H under the memory capacity $M = 100$.

As observed in Fig. 5.7a and Fig. 5.7b, aOFFICER tries to improve $r \to 1$, so all requests are satisfied. Under a higher active flow rate, more requests are sent but rejected because of memory limitations, so r decreases.

When the switch memory is scarce ($M < 1000$), H rapidly increases by MI phase of Algo. 2, and then H oscillates around a value depending on memory capacity (Fig. 5.7b). With a high memory capacity, H converges and oscillates around a lower value.

When $M = 1000$, H is stable at $H_{min} = 1$, so all flows can be installed. For the trace Hadoop, at the beginning ($t < 250$), $M = 100$ is enough for aOFFICER to keep $r = 1$. From $t = 250$, many flows appear or become active, so r decreases fast.

To cope with this increasing demand, aOFFICER reacts by adjusting H to a higher value, as observed in Fig. 5.8b.

Fig. 5.9 provides a different view about these algorithms. In this figure, the $x - axis$ represents the flow rank by size (from large to small) in log scale, while the $y - axis$ shows the success installation rate per flow r_f (e.g., number of success installation/number of requests). Higher switch memory leads to higher values of r_f, because more requests can be satisfied.

In NAIVE, large flows do not be treated differently than other flows, and many large flows are not installed ($r_f = 0$). For OFFICER, because flows are known in advance, so largest flows are identified correctly and installed ($r_f = 1$). However, at the case *EDU1-M=1000* (Fig. 5.9e), OFFICER does not install rules for some mice flows, while all flows are installed in cases of NAIVE and aOFFICER. The reason is that OFFICER does not use timeout mechanisms, while aOFFICER and NAIVE use idle timeout mechanisms to recover occupied switch memory. However, these flows are mice and does not significantly affect to the total load.

For aOFFICER, most of top 10 largest flows are installed ($r_f > 0$), which conforms with our motivation. In the Hadoop case, there are many large flows competing for resources, so some large flows are not installed as expected.

5.3.3 Adaptive Timeout and Deflection Technique

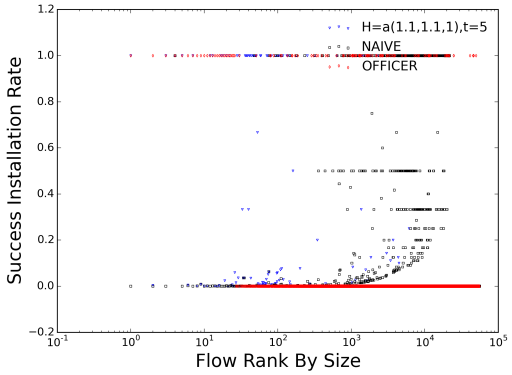
The adaptive threshold mechanism used in aOFFICER can reduce the total load, as shown in Sec. 5.3.2. In this section, we try to improve the performance of aOFFICER by enabling complementary features, such as the *adaptive timeout* and the *deflection technique*. We explain the need for these features in the following.

First, popular controllers are using fixed timeout mechanism ($t = 5s$) when installing rules. However, in many workloads, most of the flows are short-live and only need a smaller timeout value, as shown in Fig. 5.3d. Therefore, using $t = 5s$ is not memory efficient, as the switch memory may be occupied by inactive flows. To cope with this problem, studies [VPMB14, XZZ+14] propose to adapt timeout value for each flow, e.g., using the SmartTime algorithm [VPMB14], which is implemented in our simulator.

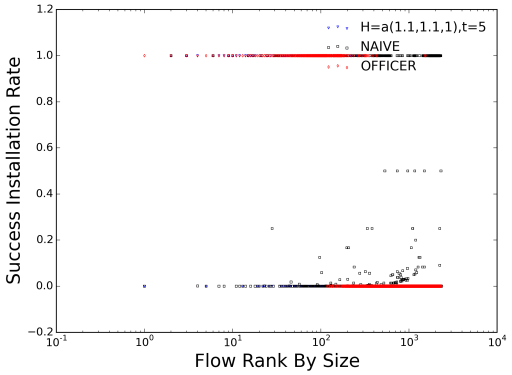
Second, as shown in Chapter 4, the shortest paths may not be optimal in terms of memory usages. Therefore, we propose the deflection technique that selects path consuming less switch memory. In this section, we consider the strategy *Close to the Edges* (CE) for the deflection technique. More details about this strategy can be found in Chapter 4.

To extend the discussion, we perform simulations using the trace EDU1 on the topology ScaleFree22, and the trace Hadoop on the topology ScaleFree80. Both new topologies, generated using NetworkX [Net15], are Scale-Free topologies and they have the same number of nodes with the original Fattree topologies. Basically, these topologies represent the evolution of the network in practice, and their node degree distributions follow a power law.

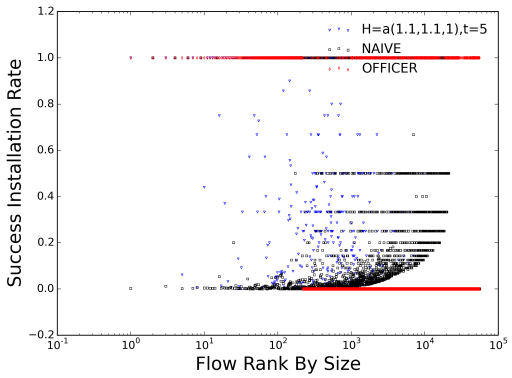
We perform simulations for aOFFICER including features such as adaptive



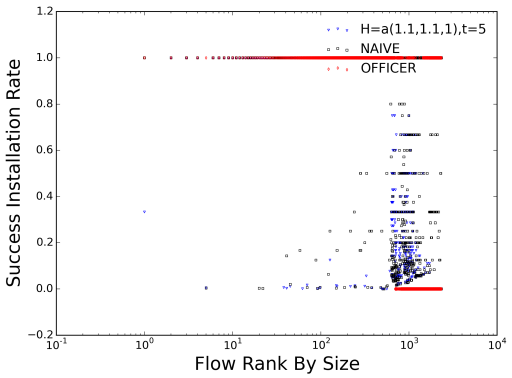
(a) EDU1-M=10



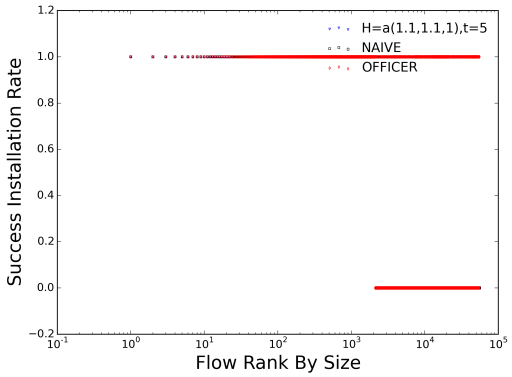
(b) Hadoop-M=10



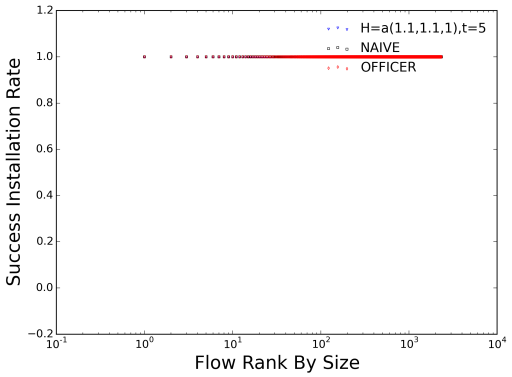
(c) EDU1-M=100



(d) Hadoop-M=100

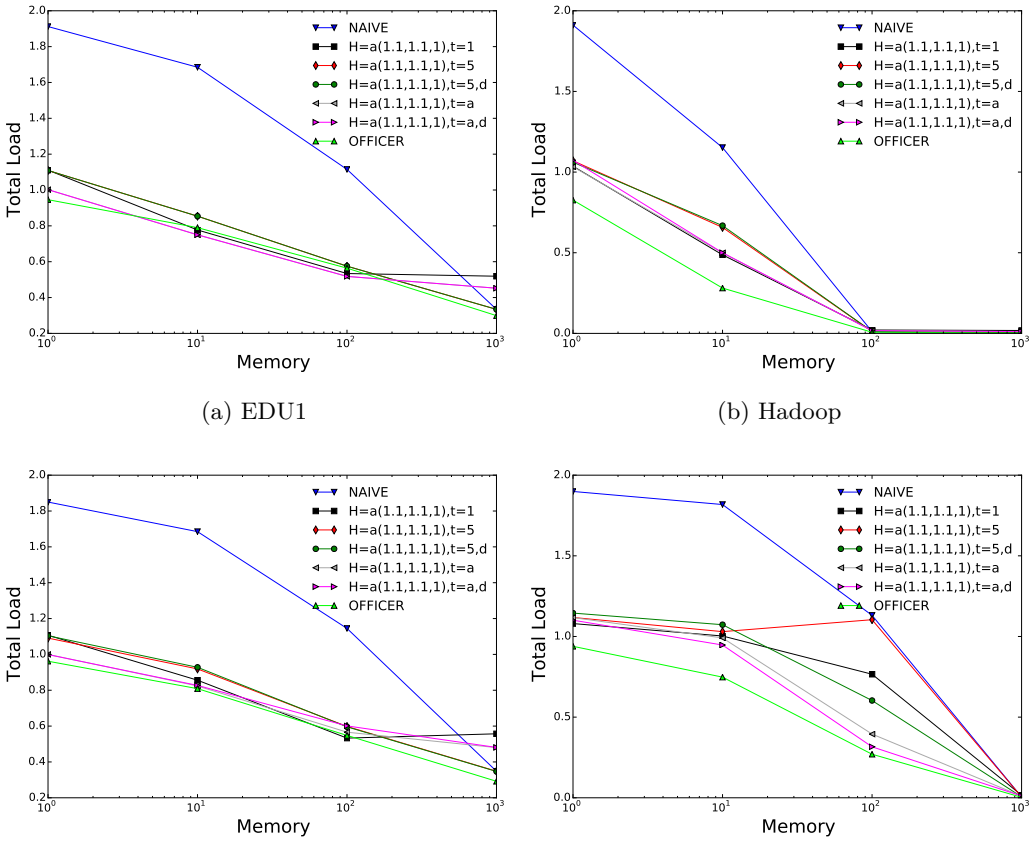


(e) EDU1-M=1000



(f) Hadoop-M=1000

Figure 5.9: Success installation rate per flow



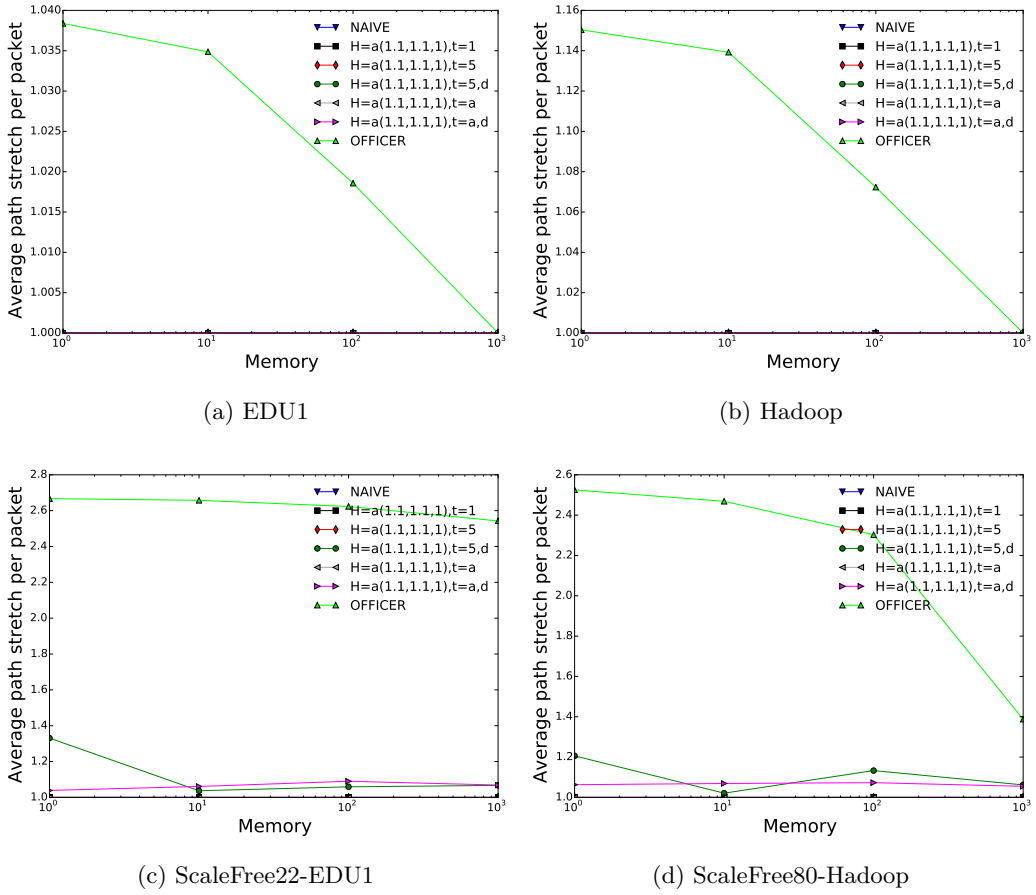
(a) EDU1

(b) Hadoop

(c) ScaleFree22-EDU1

(d) ScaleFree80-Hadoop

Figure 5.10: Simulated Total Load caused by different algorithms



(a) EDU1

(b) Hadoop

(c) ScaleFree22-EDU1

(d) ScaleFree80-Hadoop

Figure 5.11: Average path stretch per packet caused by aOFFICER

timeout (using SmartTime, denoted as $t = a$ in figure legends), and the deflection technique (using CE, denoted as d in figure legends).

Fig. 5.10 shows the total load caused by different algorithms for all datasets. In figure legends, the adaptive timeout strategy (i.e. SmartTime) is denoted as $t = a$, and the deflection technique is denoted as d. With new topologies, NAIVE still causes a high total load while aOFFICER can reduce total load significantly compared to NAIVE, in cases where the switch memory is scarce ($M = 1, 10, 100$). Furthermore, aOFFICER is comparable with OFFICER in case of EDU1, where there is few large flows. By using the adaptive timeout strategy ($t = a$), the switch memory are used better than using the fix timeout strategy and therefore, the total load is further reduced. However, when memory is not limited ($M = 1000$), both strategies can install rules for all flows, but using a fixed timeout strategy causes less total load, as timeout messages are sent less frequently.

For data center topologies, enabling the deflection technique does not reduce the total load further, compared to using the shortest path routing policy. These data center topologies have short average default path length and many shortest paths

between two nodes. Therefore, the deflection technique often selects the shortest paths to place rules. Fig. 5.11a and 5.11b confirm these observations, since the average path stretch of using aOFFICER with the deflection technique is 1.

In ScaleFree80-Hadoop (Fig. 5.10d), the average default path length is larger than in Hadoop topology. Therefore, the gain in total load by enabling the deflection technique is visible, for example, from $(H = a, t = a)$ to $(H = a, t = a, d)$, and from $(H = a, t = 5)$ to $(H = a, t = 5, d)$ at memory $M = 100$. For both ScaleFree22 and ScaleFree80 cases, aOFFICER also does not incur a high average path stretch (Fig. 5.11c and 5.11d).

OFFICER outperforms other algorithms in reducing total load, but it also causes the highest average path stretch, as observed in Fig. 5.11, even the same deflection technique is used. In OFFICER, the paths are predefined at the computation moment, and no memory recovery mechanism is used. In aOFFICER, paths are decided at the request moment, and memory resources are also be recovered (by timeout mechanisms).

In summary, our results show the efficiency and the adaptability of aOFFICER for different scenarios, in presence of constrained switch memory. First, aOFFICER can identify large flows and install rules for them, to reduce the signaling load compared to NAIVE. Second, enabling the adaptive timeout strategy (e.g., SmartTime) can reduce the total load further, compared to using the fix timeout strategy. Third, the deflection technique should be enabled if the topology has a large average default path (e.g., in ScaleFree topologies). For the data center topology, using the shortest path routing policy is enough. Forth, if the workload is known or predictable with high accuracy, OFFICER is more efficient to find rules placement, because it helps reducing the signaling load significantly. However, it may cause a high average path stretch, as shown in cases of ScaleFree topologies. Finally, when the switch memory is not limited, using NAIVE is good enough.

5.4 Conclusion

In this chapter, we study the *online* OpenFlow Rules Placement Problem, where the set of flows is not known in advance and varies over time. To satisfy the endpoint policies, existing controller platforms install rules reactively for all the flows, from the first packet seen. However, this approach is not optimal, because of a large signaling overhead, and the fact that large flows may not be installed.

We present a novel rules placement framework, called aOFFICER, that can find rules satisfying endpoint policies, network constraints, and reduce the signaling load. The novelty of aOFFICER comes from its ability to install rules for important, large flows on efficient paths, and to adjust its parameters according to traffic fluctuations. Our simulation results in realistic scenarios have confirmed the efficiency and adaptability of aOFFICER, in presence of constrained switch memory.

Chapter 6

Use Case: Improving Content Delivery in LTEs

Contents

6.1 Introduction 79
6.2 Background 81
6.3 LTE In-network Caching Architecture 81
6.3.1 Multi-level Caching Scheme 82
6.3.2 Enabling Backhaul Caching with OpenFlow 83
6.4 Content Allocation Model 83
6.4.1 Content Allocation Problem Approximation 86
6.5 Evaluation 87
6.5.1 Simulation Setup 87
6.5.2 Benefits of Caching at Different Levels 89
6.5.3 Impact of Several Levels Caching 92
6.5.4 Advantages of Using Opportunistic Caching for Networks with Loss 93
6.6 Related Work 94
6.7 Conclusion 95

In chapters 4 and 5, we presented OFFICER and aOFFICER, which are basic blocks to implement the black box abstraction. In this chapter, we study a use case, where such abstraction is leveraged to improve the performance of content delivery services in cellular networks.

The content of this chapter is the result of a collaboration with the colleagues at Aalto University (Finland) and is also reported in [KNS+15].

6.1 Introduction

Video streaming and multimedia contents distribution have taken a paramount importance in the Internet ecosystem and generate enormous amount of traffic [Cis15], and the ubiquity of mobile devices is not foreign to this trend. Nowadays, mobile traffic exceeds desktop traffic [Cis15], thanks to the ubiquitous of mobile devices and high speed mobile networks. It is evident that such traffic will continue to grow

in the coming years with the generalization of Ultra High Definition [UHD] and the impossibility to distribute it with traditional physical medias (e.g., Blu-ray).

While the traditional approach to face traffic growth is to increase link capacity, it is uncertain that just upgrading link capacity will still be economically viable for mobile carriers that must provide better services at very low prices. To overcome this major issue, the only solution is to reduce the overall usage of links. To that aim, we can leverage temporal and spatial locality of traffic with caches [WJP+13] that supply contents to consumers on behalf of the content producers. In this chapter, we thus propose Arothron, a versatile caching framework for mobile carrier networks. With Arothron, we introduce the concept of *hybrid caching* that combines opportunistic (non-collaborative) and preset (collaborative) caching. On the one hand, opportunistic caching is used to cope with short term content demand, which is bursty and very dynamic by essence [AAG+10]. On the other hand, we use preset caching to cache the most valuable contents. To implement hybrid caching, Arothron splits each storage unit into two logical storage units: one dedicated for the opportunistic caching and the other used for preset caching. The rational behind this duality in the caches is to combine both their advantages to reduce overall network loads. Using only opportunistic caching is not sufficient because of high redundancy, while using only preset caching is impractical to provide high hit ratios for highly dynamic demands.

Unfortunately, making caching ubiquitous in mobile networks is not straightforward, because of the following reason. In LTE networks, for each user equipment, a GTP tunnel is established between its eNodeB and a gateway of the core network (i.e., the Evolved Packet Core) [3GP]. GTP tunnels are used to separate the radio part from the data part in the network, in order to facilitate the handover process and mobility. As a result, data packets are not seen by intermediate equipments in the network, reducing the ability to perform caching. To overcome this limitation, Arothron replaces the usage of GTP tunnels with native forwarding thanks to OpenFlow [MAB+08b] and proposals [CR14, CRKMK14]. Thus, it becomes possible to forward the content demands directly to the most appropriate preset caches and to perform opportunistic caching on the way.

In this chapter, we present the Arothron, an OpenFlow based hybrid caching architecture, that reduce the overall usages of links and improve user experience. We first formalize a Mixed Integer Linear Programming (MILP) that decides the contents to be cached in the preset caches. As the problem is NP-hard, we propose a polynomial time greedy heuristic to find a content allocation solution in a tractable way. With extensive simulations of a typical LTE network, we study the impact of cache locations and the ratio between preset and opportunistic storage on backhaul and provider-edge link usage, as well as the gain in terms of delay.

The remaining of the chapter is organized as follows. Section 6.2 presents the background about LTE networks. Section 6.3 presents the architecture of Arothron, and the different types of caches used. Section 6.4 introduces the MILP and the heuristic used to place the contents. Section 6.5 presents the simulation results and their analysis. Finally, Section 6.6 presents some related works.

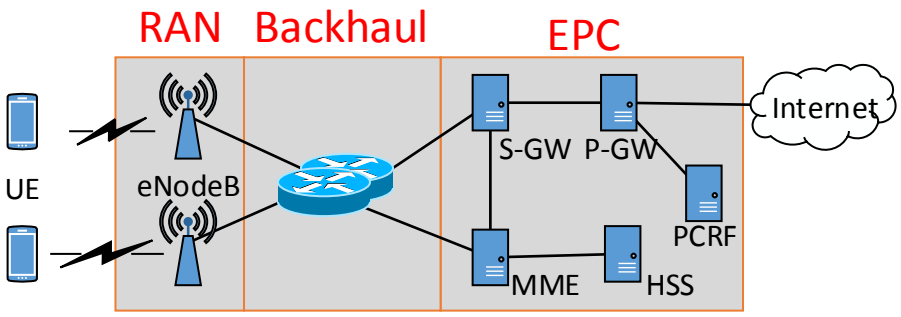


Figure 6.1: LTE architecture

6.2 Background

LTE, standing for Long-Term Evolution, is a telecommunication standard for broadband cellular networks [3GP]. Fig. 6.1 shows the overall architecture of LTE networks. The *Radio Access Network* (RAN) consists of base stations called *evolved Node B* (eNodeBs) that provide the radio access for *User Equipments* (UE). The RAN is connected to the backhaul network that consists of switches aggregating the traffic from RAN and that provides connectivity towards the network core, called the *Evolve Packet Core* (EPC).

The main components in the EPC are: (*i*) the *Mobility Management Entity* (MME) which tracks UEs and manages UE handover between eNodeBs; (*ii*) *Packet Data Network Gateways* (P-GW) which are the gateway routers to IP networks; (*iii*) *Serving Gateways* (S-GW), which act as the mobility anchors for UE handovers and forward packets to P-GWs; (*iv*) the *Home Subscriber Server* (HSS) that contains all the information about UE subscribers, such as QoS profiles; (*v*) the *Policy Control and Charging Rule Function* (PCRF), which controls flow-based charging functionalities in P-GWs and ensure that data flows are treated according to user profiles.

Mobility is the critical functionality in mobile networks, and new technologies need to provide proper reliability and low latency for handover processes. The common handover solution is to use the *GPRS Tunneling Protocol* (GTP) to establish tunnels between eNodeBs and S/P-GWs.

6.3 LTE In-network Caching Architecture

In this section, we detail Arothron, a versatile caching architecture for LTE networks. First, in sec. 6.3.1 we present our multi-level caching scheme for LTE networks that combines opportunistic and preset caching. Then in sec. 6.3.2, we show how to

implement this caching scheme in practice in LTE networks.

6.3.1 Multi-level Caching Scheme

The numerous studies on Information-Centric Networking have demonstrated the advantages of in-network caching [ZLL13] and two trends in the way to manage caches are emerging: *opportunistic caching* and *preset caching*. The idea behind opportunistic caching is storing contents when they pass through the cache, generally using the LRU cache eviction policy, to leverage the spatial and temporal locality of demands over short periods of time. On the contrary, preset caching reasons in terms of average behaviour of the system and leverages the power law characteristics of content demand over long periods of time. Furthermore, optimization techniques are used to decide if and where a content should be stored in preset caches.

In LTE networks, customer terminals are inherently mobile, roaming between eNodeBs to which they are connected over lossy radio links. Furthermore eNodeBs are geographically spread and usually connected to the network core over long-distance backhaul links of limited capacity. For LTE networks, it is thus desirable to have a caching system that in addition to reducing the overall link usage, prevents loss events at the radio access network to propagate in the core through the backhaul. To that aim, we propose, Arothron, a versatile caching architecture for LTE networks that combines opportunistic and preset caching.

In Arothron, any node in the LTE network can be equipped with a storage unit. The LTE network therefore constitutes a network of caches. However, as the system must handle at the same time highly dynamic demand caused by mobility and losses and stable overall demand, each physical storage unit is split in two logical caching units, one is dedicated to opportunistic caching while the other is used for preset caching. An opportunistic cache stores any content that transits through it and the storage space is managed according to the LRU eviction algorithm in order to absorb short term bursty demands. On the contrary, the contents to be put in a preset cache are determined in advance based on expected content demand. The most beneficial contents that are cached according to the Mixed Integer Linear Programming (MILP) formalized in Sec. 6.4 are then pre-fetched in the preset caches. The MILP takes into account the network capacity, the diversity of paths to decide which is the best preset cache unit to put the contents so that it maximizes the amount of content delivered by the caching system while minimizing the network load. The preset cache is complementary to the opportunistic cache as it favors content with an overall high demand, regardless of their dynamics.

The ratio of storage space dedicated to the opportunistic cache on each storage unit is a key factor in the design of Arothron as well as where to deploy the different caches. For example, caches located between the radio access network and the backhaul links aim to alleviate the impact of losses while caches in the core network aim at reducing the traffic towards the Internet. We extensively evaluate these two aspects with a simulator in Sec. 6.5.

6.3.2 Enabling Backhaul Caching with OpenFlow

In LTE networks, the IP traffic between RAN and EPC is encapsulated in GTP tunnels so that the backhaul network is agnostic of the handover process. This architecture makes caching inside the backhaul network impossible as backhaul network elements only see the tunnels and not the traffic carried by them. To overcome this limitation, we leverage the fact that the backhaul typically operates in level 2 with Ethernet or MPLS and use OpenFlow [MAB+08b] to replace the GTP tunnels by strict hop-by-hop switching meaning that traffic will be visible for any equipment in the backhaul network instead of being hidden in a tunnel.

The Arothron architecture consists of three elements: the *analyzers*, the *caches* and the *controller*. The analyzers are implemented directly at the eNodeB level while caches are spread in the network and the controller is joined to the MME in a centralized location. The analyzer intercepts all requests for contents from the customer equipments connected to the eNodeB, extracts the content name from the request and queries the controller for the location of preset cache where the content is available.[1] After that, the request is then forwarded directly to that location. If the location is the eNodeB itself, then the request is treated locally. Otherwise, the request is sent natively in the backhaul network. If the cache is located in the backhaul network or in the EPC, the packet containing the request is tagged with the VLAN tag corresponding to the caching equipment.[2] It is worth noting that our solution does not require to modify the backhaul network equipment to work as packets are forwarded natively in the technology of the backhaul. As packets are never encapsulated, any intermediate network device can intercept the requests which makes opportunistic caching anywhere in the LTE network possible with our Arothron architecture.

Continuously updating the preset caches would cause high signaling traffic. Therefore, in Arothron, the controller proceeds to periodic relocation operations. During those operations, the controller gathers information from the analyzers and caches about which contents were requested and about their requests frequency. Based on the pool of requests created and using prediction algorithms, the controller computes the optimal content allocation, if any. The content allocation solution is sent to all the caches that will download the missing contents from other caches or from the origin servers. As relocation generates extra traffic, this step is performed during off-peak hours.

6.4 Content Allocation Model

In this section, we formalize a Mixed Integer Linear Programming (MILP) for optimal content allocation in preset caches, based on the notations defined in table 6.1 and 6.2.

[1] The controller response is cached for faster processing for the subsequent requests for the same content.

[2] In case of an MPLS backhaul, MPLS labels are used instead.

Table 6.1: Parameters

Parameter	Description
V	set of nodes
$*$	set of servers
V^+	nodes and servers $(V \cup *)$
E	set of directed links $(i, j) \in V^+ \times V^+$
C_i	cache size of node i (MB)
B_l	link capacity of $l \in E$ (Mbps)
M	set of contents
s^m	content size (MB)
s_r^m	size of the request for content m (MB)
u^m, r^m	upstream, downstream bit rate of content m (Mbps)
$P_{i,j}$	sequence of links on the path from node i to j
a_i^m	total number of requests for content m at node $i \in V$ over T

The network is modeled as a finite directed graph $G = (V, E)$ where V is the set of nodes, E is the set of directed links between nodes. M is the set of contents (e.g., videos). Each content has a size s^m, downstream rate r^m and upstream rate u^m (e.g., for video streaming services). Each node has a preset cache with capacity C_i. Servers $s \in *$ hold all the contents.

A mobile user sends a request for content m to a node i, if m does not exist in the cache i, node i forwards the request to one of the neighbor caching the content, and the content will be fetched from that node. If the content is not cached, the request is forwarded to one of the servers. a_j^m stands for the total number of requests for content m at node j in the whole modeling period T.

These parameters are used to find which content is placed at which node (x_i^m) and the probability of forwarding requests to neighbors ($y_{i,j}^m$). Therefore, we formalize a MILP for content placement $(1) - (8)$.

The main goal of using caching is to reduce the overall traffic load in the network, as represented in the objective function (6.1). Constraint (6.2) stands for the memory capacity constraint, the total size of cached contents on i does not exceed the cache capacity. Constraint (6.3) makes sure that all requests from a node will be satisfied, locally or from other nodes. Constraint (6.4) and (6.5) define the range values for x and y. Constraint (6.6) captures the fact that node i can serve content m if and only if the content m is cached on i. Constraint (6.7) makes sure that the load on the link does not exceed the link capacity. More precisely, the load on the link $l = (i, j)$ consists of the upstream volume from i to j (first component) and the downstream volume from i to j (latter component). Constraint (6.8) indicates that servers hold all the contents.

Table 6.2: Variables

Variable	Meaning
x_i^m	binary, to indicate if content m is placed at node i
$y_{i,j}^m$	float in $[0,1]$, to indicate the percentage of requests from node j to i

$$min \sum_{m \in M} \sum_{i,j \in V^+} (s^m a_j^m y_{i,j}^m + s_r^m a_i^m y_{j,i}^m) \tag{6.1}$$

$$\forall i \in V : \sum_{m \in M} s^m x_i^m \leq C_i \tag{6.2}$$

$$\forall (j,m) \in V \times M : \sum_{i \in V^+} y_{i,j}^m = 1 \tag{6.3}$$

$$\forall (i,m) \in V \times M : x_i^m \in \{0,1\} \tag{6.4}$$

$$\forall (m,i,j) \in M \times V^+ \times V : 0 \leq y_{i,j}^m \leq 1 \tag{6.5}$$

$$\forall (m,i,j) \in M \times V^+ \times V^+ : y_{i,j}^m \leq x_i^m \tag{6.6}$$

$$\forall l \in E :$$
$$\sum_{m \in M} \sum_{i,j \in V^+ | l \in P_{i,j}} (u^m a_j^m y_{j,i}^m + r^m a_j^m y_{i,j}^m) \leq B_l \tag{6.7}$$

$$\forall (m,i) \in M \times * : x_i^m = 1 \tag{6.8}$$

A similar content placement model can be found in [AAG+10], however, our MILP model accounts for upstream traffic, and the fact that requests might be forwarded to different servers.

Corollary 1. *The content allocation problem is NP-hard.*

Proof. Consider an instance of the problem, with one node of cache size C, one server, and one link l between them. Assuming that the upstream rate and size of request messages are negligible ($u^m = 0, s_r^m = 0, \forall m \in M$), the objective function is to minimize the load on link l:

$$min \sum_{m \in M} s^m a^m (1 - x^m) \cong max \sum_{m \in M} (s^m a^m) x^m \tag{6.9}$$

where x^m is subjected to memory constraint:

$$\sum_{m \in M} s^m x^m \leq C \tag{6.10}$$

This is exactly the form of the Knapsack problem, which is known as NP-hard. So the original problem is also NP-hard. □

6.4.1 Content Allocation Problem Approximation

Since the content allocation problem is intractable, we propose a polynomial time greedy heuristic (Algo. 3) to approximate the content allocation solution in a practical way. As content popularity usually follows an heavy tailed distribution, a greedy approach usually provides good results [AL06].

The main idea is to cache contents with the highest demand by placing them in the most suitable nodes that are as close as possible to the UEs. So first, the demands, for each content at each edge node, are sorted based on their volume (line 7).

For each neighbor distance d of node i, we define a cost function (6.14) that includes memory and bandwidth cost. The memory cost (6.11) indicates the percentage of memory which will be occupied. Similarly, the bandwidth cost (6.12) and (6.13) indicates the percentage of upstream and downstream bandwidth that will be consumed. If node j does not have enough memory or bandwidth ($c_{m,up,down} > 1$), the total cost is ∞. If the content exists on this node ($x[m,j] = 1$), the total cost includes the bandwidth cost only.

Among the neighbors, the one with minimum cost will be selected to place content m, the variable X and Y are set to corresponding values, then the next demand is processed.

For each set of demands in an eNodeB (m,i), content m will be placed in at most one node, to avoid exceeding cache size.

$$c_m[m,i,j] = \frac{s^m}{C_{available}[j]} \tag{6.11}$$

$$c_{up}[m,i,j] = \frac{u^m f_i^m}{min(B_{available}[l] | l \in P[i,j])} \tag{6.12}$$

$$c_{down}[m,i,j] = \frac{r^m f_i^m}{min(B_{available}[l] | l \in P[j,i])} \tag{6.13}$$

$$c[m,i,j] = \begin{cases} \infty & ; c_m > 1 \vee c_{up} > 1 \vee c_{down} > 1 \\ c_{memory}(1 - x[m,j]) + c_{up} + c_{down} ; otherwise \end{cases} \tag{6.14}$$

The temporal complexity of our greedy heuristic is $\mathcal{O}(|V|^3 + |M||V|log(|M||V|) + |M||V|^2|DIAMETER|)$. In usual scenarios $|M| >> |V|$, so the complexity can thus be rewritten as $\mathcal{O}(|M||V|log(|M||V|))$.

The complexity of the Greedy is driven by the three following components:

- (line 3-6) Finding all pairs of shortest paths to construct the neighbor set of each node. This is achieved using the Floyd–Warshall algorithm [Flo62], which is $\mathcal{O}(V^3)$.
- (line 7) Sorting the demand, which is $\mathcal{O}(|M||V|log(|M||V|))$.
- (line 8-14) Finding the neighbor with minimum cost, which includes computing of bandwidth cost and verifying neighbors is of complexity $\mathcal{O}(|M||V|^2|DIAMETER|)$.

Algorithm 3: GREEDY

$X \leftarrow [0]_{M \times V}$; $X \leftarrow [1]_{M \times (*)}$
$Y \leftarrow [0]_{M \times V^+ \times V}$
$P \leftarrow [shortest\_path[i,j]]_{(i,j) \in V \times V^+}$
for $i \in V$ **do**
  for $d = 0$ **to** $DIAMETER$ **do**
    $NEIGHBOR[i,d] \leftarrow (j \in V^+ | P[i,j] = d)$
$DEMAND \leftarrow \texttt{sort}((m,i) \in M \times V | (f_i^m \times (s^m + u^m)) \downarrow)$
for $(m,i) \in DEMAND$ **do**
  for $d = 0$ **to** $DIAMETER$ **do**
    $c[m,i,n] \leftarrow min(c[m,i,j] | j \in NEIGHBOR[i,d])$
    if $c[m,i,n] < \infty$ **then**
      $x[m,n] = 1$
      $y[m,n,i] = 1$
      break
return X, Y

6.5 Evaluation

6.5.1 Simulation Setup

This section describes the simulations environment we setup to evaluate Arothron.

6.5.1.1 Simulated Network

The simulated network come from a French telecom operator, which consists of 16 eNodeBs, 16 edge switches, 8 aggregation switches, 4 core routers. A data center is at the other edge of the network, and hosts all the functions of the EPC, except the P-GW. The 16 eNodeBs are equally distributed over the simulation area and all have the same circular coverage area, so that all the simulation area is covered. The strength of the signal is assumed to be sufficient to provide the highest download speed of contents from the pool used for the simulation on any point of the coverage area. Fig. 6.2 shows a schematic of the simulated network.

In order to apply our caching system, all the edge switches support OpenFlow and the other switches support VLANs and consider 3 levels of caches. The first level of caches includes all the caches that are located at the 16 eNodeBs. The second level of caches is located at the 16 edge switches. Finally, the third level caches is located in the aggregation switches. Each cache unit is split between opportunistic and preset spaces and we explore the impact of the ratio of the space for opportunistic cache to the space for preset cache as well as the impact of the overall cache size on the performance of Arothron.

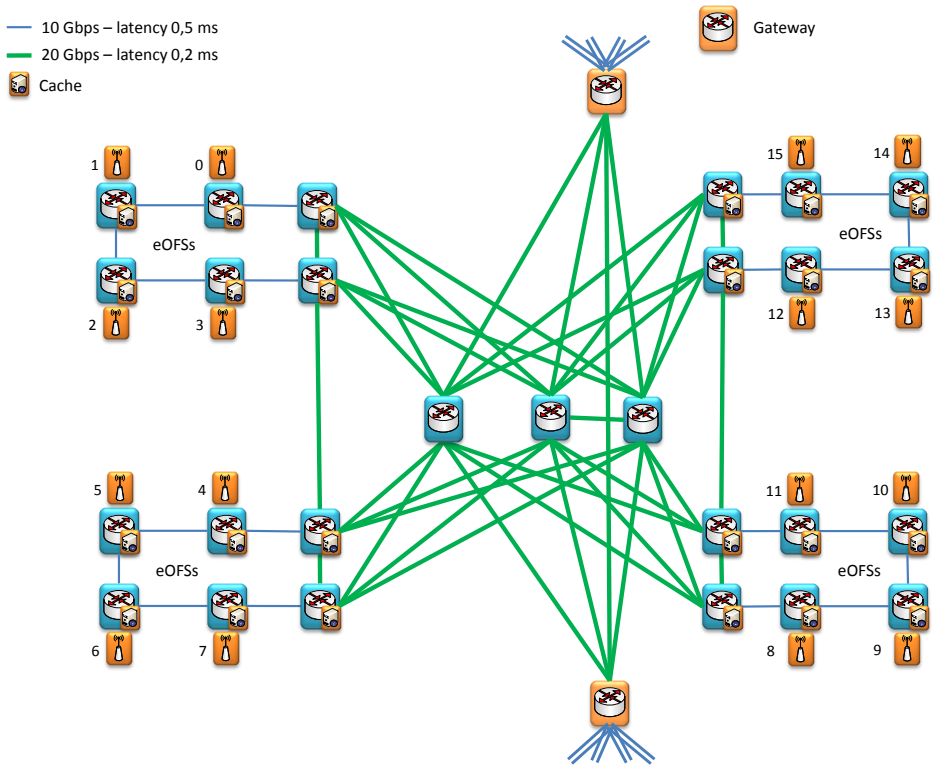


Figure 6.2: Simulation network

6.5.1.2 Simulation of user behaviour

We simulate the customer terminals movements with a Gauss-Markov mobility model [Meg10] where the simulation area is a semi-urban square of 4km sides. The maximum speed is set to 80 km/h for any user as the simulation is in a semi-urban area. The average speed of the users is distributed to reflect their activities, most of them are rather immobile, some of them use low-speed urban transportation, and a few ones have high average speed to simulate peri-urban mobility.

The users request contents following a Zipf distribution with a decay parameter $\alpha = 0.9$ [BCF+99].[3] A user can download a single content at once. Once a download is completed, the user waits before starting a new download by a waiting time drawn from a Poisson distribution. In all simulations, we consider 1,000 independent customers consuming contents from a catalog of 100,000 entries of a mean size of 9 MB that is representative of user-generated videos [CDL07]. For each customer, the average download rate is 5 Mbps and the average upload rate is 50 kbps. Simulations have been repeated 3 times and the average values are given (the deviation is negligible).

[3] Simulations with other decay parameters have been performed and confirm the findings with $\alpha = 0.9$.

6.5.1.3 Evaluation metrics

We consider the latency that represents user satisfaction (i.e., the lower the latency the better) and the network load that summaries the operator costs (i.e., the lowest the load the cheaper operation cost). We also evaluate impact of the location of caches, their size, and the ratio between opportunistic and preset caches, to these metrics.

The advantage in terms of latency is evaluated with the *latency reduction* metric. This metric is computed as the average of the relative delta between the latency induced by the network when Arothron is used and when no caching is used. The latency reduction is an upper bound as we intentionally neglect the latency induced by the external network when contents are fetched through the gateway.

The advantage in terms of network load is studied with the *load reduction* metric. The load of a set of links is defined as the sum of the loads on all the links in the set (both up and down links). The load reduction of a set of links is thus the average of the relative delta between the load of the set of links when Arothron is used and when no caching is used. We distinguish two different cases. First, when all the links of the network are taken into account (except radio links), the load reduction is called the *network load reduction*. Second, when only the peering and exchange point links are taken into consideration, the load reduction is called the *gateway load reduction*.

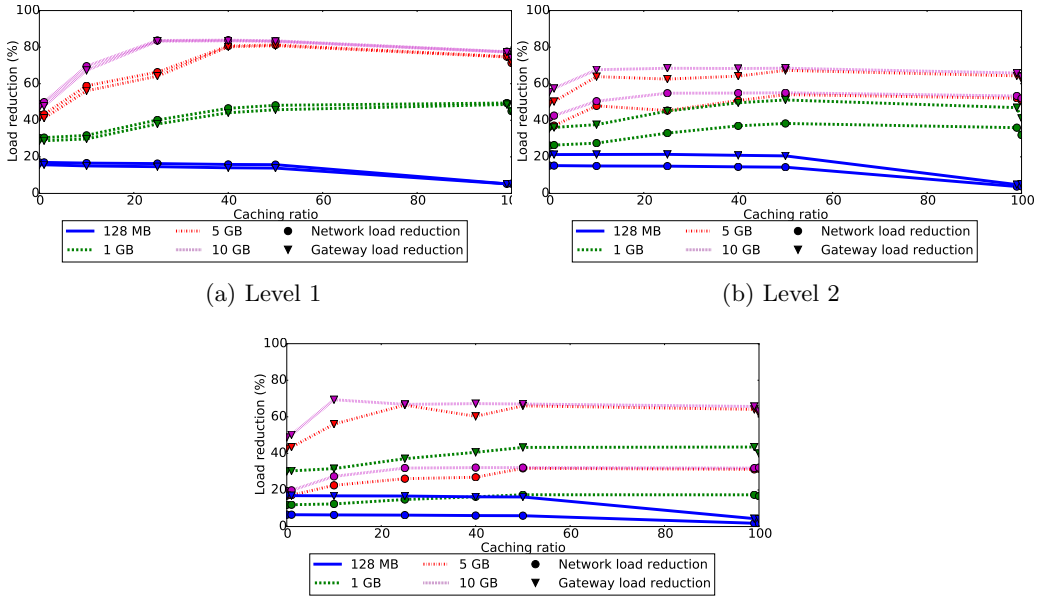
Several parameters of Arothron have a direct impact on the performances of the system. In Sec. 6.5.2 we study the impact of the parameters on the performances when caches are deployed only at one level. In Sec. 6.5.3 we determine the performance of the system with caches deployed at all the levels. Finally, in Sec. 6.5.4 we see how caching reacts in presence of packet losses.

6.5.2 Benefits of Caching at Different Levels

A major strength of our architecture is its ability to place caching units anywhere in LTE networks in a multilevel way. To assess the benefit of this flexibility, this section studies the respective advantages of deploying caches at a particular level of the network while Sec. 6.5.3 studies how combining caches at several levels improves network performance.

We consider 3 scenarios: (*i*) caches are deployed only in the eNodeBs (i.e., level 1); (*ii*) caches are deployed only in the backhaul (i.e., level 2); and (*iii*) caches are deployed only in the EPC (i.e., level 3). In all scenarios, each storage unit is split in an opportunistic part and a preset part and we vary the *ratio* of storage allocated to the two parts. When the ratio is 0%, the caches are performing only preset caching and when the ratio is 100%, the caching is only opportunistic. In addition to the impact of this ratio, we study the performances of the system using four different node storage capacities: 128 MB, 1 GB, 5 GB, and 10 GB.

The performance of the system is shown in Fig. 6.3 Fig. 6.4. We can see that, as the intuition suggests, the location of caches in the network has a direct impact



(a) Level 1

(b) Level 2

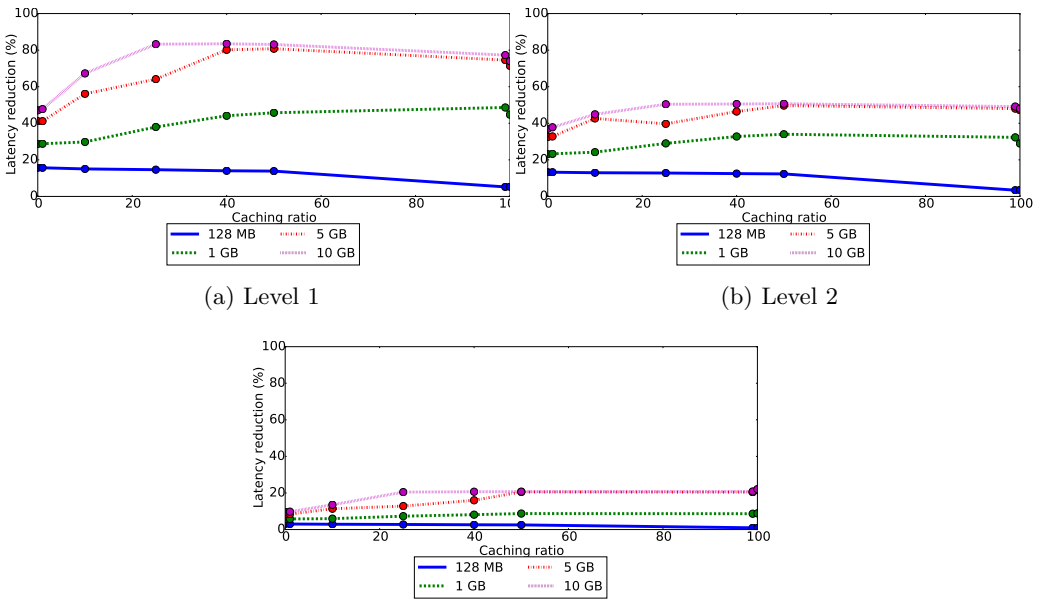
(c) Level 3

Figure 6.3: Reduction of the network and gateway load depending on the level of caches and the caching ratio (x-axis)

on the latency with a clear advantage for deployment at level 1 as it reduces the necessity to use long-distance backhaul links that are causing most of the delay in LTE networks. On the contrary, the location of caches has no significant impact on the network or gateway load reduction.

Transferring small shares of capacity from a pure preset cache to its opportunistic counterpart induces a clear increase in the network and gateway load reduction as in the latency reduction. Indeed, adding some opportunistic caching permits reducing the traffic related to the dynamic part of the requests since the LRU policy captures this dynamism. However, by increasing the ratio of space dedicated to opportunistic caching, we observe that the performance gain saturates before dropping. The reason is that by giving more shares to opportunistic caches, there is less space for storing the most popular contents in the preset caches that participate the most to the reduction of load and latency. This space reduction is not compensated by the space provided to opportunistic caches as their lack of cooperation causes redundancy and thus for the same total storage capacity, less contents can be stored.

Fig. 6.3 and Fig. 6.4 show that the size of caches has a clear impact on the performance: when caches are too small the benefit is limited but using very large caches doesn't bring much benefits either. Moreover, the storage size impacts the way performances evolve with the caching ratio. For 128 MB of storage capacity, the ideal is to strictly use preset caches. This is because such small cache size only gives space to store a limited number of contents (i.e., less than 15 on average with our



(a) Level 1

(b) Level 2

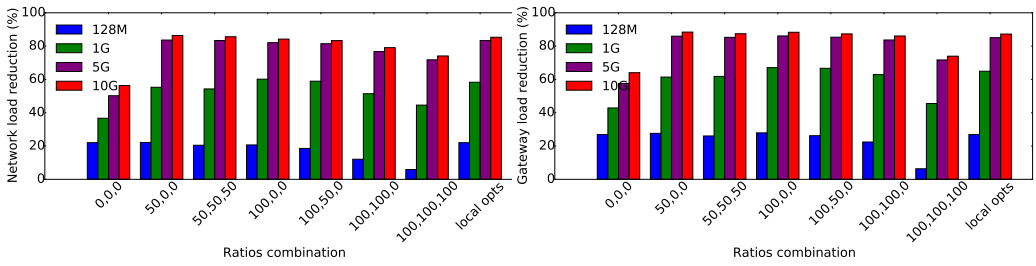
(c) Level 3

Figure 6.4: Reduction of user perceived latency depending on the level of caches and the caching ratio

setup), meaning that the number of contents that can be hold by one opportunistic cache is too small and it is unlikely that two subsequent requests for the same content will cause a hit in the cache. A small opportunistic cache only works with highly skewed popularity distributions where some contents are so popular that they are very likely to be requested every few requests passing the cache.

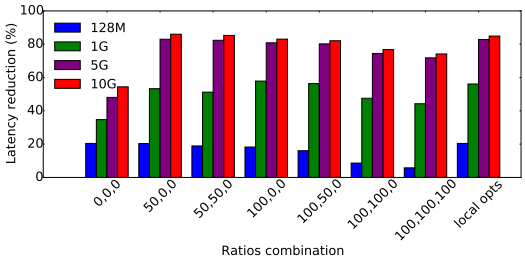
On the contrary, with larger cache sizes, the performance benefit from setting opportunistic caching. Pure preset caching performance is half of pure opportunistic caching, while the optimal is achieved with a combination of both in all the levels. In both cases, using a single type of caching does not provide the highest performance. Interestingly, the ideal ratio to adopt between opportunistic and preset caching depends on the size of the caches, it is closer to pure preset for big caches and closer to pure opportunistic for smaller caches. The reason is that because of LRU, opportunistic caches must be of a minimal size, which mostly depends on the inter-arrival times of requests for the same content. With large caches a small ratio permits to achieve this minimal size while for small caches, it requires a higher ratio to obtain it.

This section shows that to improve network performances, it is ideal to combine rather small opportunistic caches and relatively large preset caches. It also shows that if the main objective is to reduce the user perceived latency, caches should be deployed in the eNodeBs. However, as deploying caches in the eNodeB may cause high OPEX, if the main objective of caching is to reduce the overall network load,



(a) Network load reduction

(b) Gateway load reduction



(c) Latency reduction

Figure 6.5: Network performance in case with multiple levels of caches

deploying caches in the backhaul or in the EPC is more appropriate.

6.5.3 Impact of Several Levels Caching

We determined the optimal ratios for each possible level of caching separately in Sec. 6.5.2 and in this section, we analyze how these ratios impact network performances when caches are deployed simultaneously at all the levels of the network. For that we simulate cases with various combinations of ratios and compare their performances with the ones obtained if the ideal per-level ratios found in Sec. 6.5.2 are used.

The network performances obtained for the most representative combinations are displayed in histograms in Fig. 6.5. Each group of bars represents a particular ratio combination where a distinctive bar is used per tested storage capacity. The label of the group is a triplet a, b, c that gives the ratio used at each level (a corresponding to level 1, b to level 2, and c to level 3). The simulations made with the optimal ratios found in Sec. 6.5.2 is labeled **local opts**.

Fig. 6.5 highlights that even though the combination of local optima does not ensure the global optimum when multiple levels of caches are used, it consistently provides performances close, if not equal to the optimum. In an interesting manner, the $100, 0, 0$ ratios combination (i.e., pure opportunistic at the eNodeBs and pure preset in the network) also consistently offers among the best performances. It demonstrates the benefits of combining opportunistic and preset caching in a single architecture. However, in Sec. 6.5.2 we saw that when only one level of caches is

used, it is preferable to use opportunistic and preset caches together on each cache, but if several levels of caches are used, the impact of the ratio on the third level is nearly none as then most of the caching is done on the first two levels. The ratio is very important on the first level, where opportunistic caching is required to achieve good performances. In the core, reducing the share of preset caching reduces the performance of the system. Thus, preset caching should be performed in the core to achieve the best results.

6.5.4 Advantages of Using Opportunistic Caching for Networks with Loss

As demonstrated in previous sections, caching at the level 1 (i.e., at the eNodeB) significantly reduces latency as well as network and gateway load. However, using different caching techniques leads to different performances of the caches in case of retransmissions due to losses. We thus analyze the performances of the network when caches are installed only at level 1 and are either fully opportunistic (i.e., ratio = 100%) or fully preset (i.e., ratio = 0%) for three typical network loss rates: (*i*) 0% for a fixed network, (*ii*) 0.03% for a LTE network, and (*iii*) 0.25% for a WiFi network [CTN+12]. We suppose that each cache is large and that there is no loss on the upload as the requests are considerably smaller than the responses and that the loss occurs on the RAN.

Fig. 6.6 shows a clear advantage of opportunistic caching over preset caching in all situations, but the advantage is boosted in case of high loss rate. The reason is that when a loss occurs, the request is retransmitted within a short period of time and in case of opportunistic caching, the content is likely to be in the cache as the presence of a content in the cache only depends on the last time it has been requested, not its popularity, which tends to favor caching of content for requests seen within a short time scale. On the contrary, preset caches are biased towards popular contents and less popular contents cannot benefit from the cache, even upon retransmission. Indeed, preset caching is not able to cope with this dynamic and variable demand as the set of contents stored is fixed between two relocations.

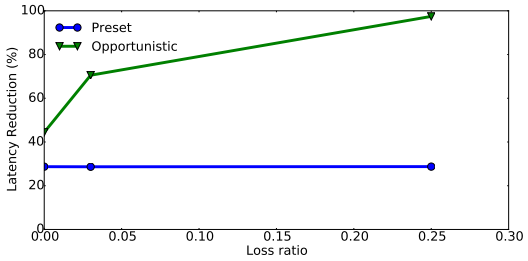


Figure 6.6: Reduction of the user perceived latency depending on the type of caching and the loss rate

6.6 Related Work

Overcoming the cellular mobile traffic explosion in cellular networks with limited link capacities [XWF+15] can be achieved through four emerging caching techniques deployed at the different layers of LTE cellular networks (See Fig. 6.1): Evolved Packet Core (EPC) caching, Radio Access Network (RAN) caching, Device-to-Device (D2D) caching, and Information-Centric Network (ICN [JST+09])-based caching.

Woo et al. [WJP+13] show that up to 58 % of traffic from Korean cellular networks consists of redundant data. Ramanan et al. [RDH+13] analyze the gain of HTTP content caching at S-GW using LTE network traces and show that the advantages of deploying caches at EPC nodes: (*i*) easy management of both cache servers and EPC nodes; (*ii*) high hit ratio. However, other studies ([AD12, GSD+12]) suggest to deploy caches within the radio access network (RAN). Ahlehagh et al. [AD12] demonstrate the feasibility of performing video caching in RAN and propose two new caching policies based on user preference profiles to improve hit ratio at eNodesBs. Golrezaei et al. [GSD+12] introduce "helpers" base stations, with large storage capacity and caching functionalities. They formalize the content allocation for helpers, prove its hardness, and propose approximation algorithms. The advantages of caching at RAN are: (*i*) reducing traffic on backhaul links and (*ii*) improving user quality of experience. However, in practice, storage capacity is small on eNodeBs [WCT+14] and the number of users served by each eNodeB is small, leading to small cache hit ratios.

Device-to-Device (D2D) caching is another approach to reduce overall bandwidth usage [GDM14, APVG14], by exploiting unused storage space available in mobile devices to form a distributed cache system. Golrezaei et al. [GDM14] introduce a D2D architecture and express the optimal collaboration distance as a function of the exponential parameter of the content demand distribution. Moreover, they show that a central coordinator is not needed. Altiari et al. [APVG14] formalize a geometric model for distributed D2D caching, which can be used to estimate how many requests could be served. D2D caching is suitable for synchronous streaming of live events that can potentially improve throughput and energy efficiency, and reduce traffic load for eNodeBs. However, D2D caching requires solving important implementation challenges, such as interference management and resource allocation [AWM14].

The similarity in the shift of traffic towards data consumption allows to apply the new Information-Centric Networking (ICN) communication paradigm [ZLL13]. In ICN, each node has caching capabilities, and unique resource identifiers (URI) are used for routing and caching decisions. As a proof of concept for cellular networks, Han et al. [HWC+13] implemented an ICN architecture on a commercial WiMAX base station.

ICN makes caching become more ubiquitous, flexible and can be used for different type of contents. There are enormous work on ICN covering different aspects of caching. Several studies point out that the cache space should be allocated proportional to node degree [PCP12] and more for edge nodes rather than core nodes [RR12] to obtain high performance. Eum et al. [ENM+12] shows that a

simple random replacement achieves similar performance with LRU and complex replacement algorithms are not suitable for ICN. Other studies propose caching decision algorithms, such as based on probabilistic [RR12] or content popularity [CLP+12].

Nevertheless, Fayazbakhsh et al. [FLT+13] point out ubiquitous caching like ICN might not be necessary because the caching benefit comes from sa mall number of caches, and the difference of performance between simple edge caching and ubiquitous caching is relatively small. Moreover, ICN is a clean slate solution and there are challenges needed to be addressed before deployment in production networks [ZLL13].

Departing from the state of the art, our work takes into account the fact that caches can be placed in access networks of LTEs, using our proposals [CR14, CRKMK14, NST13a, NST13b].

6.7 Conclusion

OpenFlow enables new opportunities to simplify network management and reduce costs. In Chapter 4 and Chapter 5, we propose OFFICER and aOFFICER, which are basic blocks to implement the black box abstraction. With the black box abstraction, operators can focus on defining optimal high level endpoint policies and do not need to care about how to transform them into low level OpenFlow rules. In this chapter we present a use case of OpenFlow blackbox abstraction for a high level application: improving content delivery services in LTE.

The tremendous growth of mobile Internet traffic puts cellular network operators under pressure to find solutions in order to reduce latency and bandwidth usage. To cope with this problem, in-network caching seems to be a promising solution. However, it is not possible to deploy caches everywhere in an LTE network, as user equipment and the network core rely on IP, whereas the backhaul network does not and uses tunnels to transport the traffic of customers.

In this chapter, we propose Arothron, a versatile caching framework for LTE networks. Wee replace the usual tunneling techniques used in LTE to handle mobility and handover by native packet switching by virtue of OpenFlow. With this approach, it is then possible to deploy caches anywhere in an LTE network and construct a multilevel caching infrastructure. In Arothron, each storage unit is split in two logical caching units: one opportunistic part and one preset part. The opportunistic cache is implemented using LRU to absorb the dynamic part of traffic while the preset cache uses a collaborative approach to store the optimal contents. As each level of the network has its own particularity, the storage shares given to the opportunistic and preset caches can be adapted for each level.

We develop the Arothron architecture and formalize a Mixed Linear Integer Programming to allocate contents in preset caches. As the content allocation problem is NP-hard, we propose a polynomial complexity greedy heuristic to solve it. With extensive simulations, we show that the network overall performances are better if

each storage unit combines opportunistic and preset caching instead of using only one caching approach. The reason is that opportunistic caching is able to deal with the instantaneous dynamics of traffic, for example due to losses and mobility, but is not efficient at handling long term content popularity trends. On the contrary, preset caches are efficient for long term traffic demand pattern, but are unable to cope with sudden demand changes. Thus, based on observations, we determine the optimal ratio for each level of caching separately and found out that the optimal ratio is not the same when caches are deployed at multiple levels.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Nowadays, a massive number of connected devices and an ever increasing traffic volume are huge burdens for the network infrastructure. To keep pace, the network is often upgraded and reconfigured. However, managing and reconfiguring many diverse devices (e.g., switches, routers, middleboxes) using vendor-specific commands are complicated, time consuming and error prone. The main reason is that the network devices are closed boxes running proprietary softwares, which are hard for operators to innovate and to customize.

By separating the network control logic from forwarding devices and providing the programmability, Software-Defined Networking (SDN) promises to simplify network management greatly. To implement the concepts of SDN, the OpenFlow protocol has been proposed. In OpenFlow, control logic functions are logically centralized in controllers, and compiled into matching-actions rules on OpenFlow switches. Many applications (e.g., access control, traffic engineering) can be implemented by installing appropriate OpenFlow rules. However, existing OpenFlow controllers still force operators to handle the problem of selecting and populating rules to satisfy policies and network constraints, which we refer as the OpenFlow Rules Placement Problem (ORPP). This problem is not trivial, because many rules are available for selection, while only a limited number of rules can be installed on OpenFlow switches. Furthermore, installing inappropriate rules may degrade the network performance and violate the policies.

We believe that SDN and OpenFlow will be widely deployed and be important components of future network architectures, e.g., 5G cellular networks. Our vision is that with OpenFlow, the network can be abstracted as a *black box*, which hides the complexity of network management and exposes high level application programming interfaces. Using the blackbox abstraction, high-level policies (e.g., endpoint policies) defined by operators are compiled into low-level rules and installed on corresponding OpenFlow switches. Moreover, rules can be automatically updated to cope with demand fluctuations or network failures. In this thesis, we present several studies aiming to realize the black box abstraction using OpenFlow.

In Chapter 3, we first formalize the ORPP and identify its challenges, including resource limitations and the signaling overhead. Second, we classify and discuss the pros and cons of existing solutions focusing on the ORPP. We find out that most of the existing solutions enforce the shortest path routing policies (e.g., OSPF, ECMP) when installing OpenFlow rules, which is not efficient in terms of switch memory

usage. We believe that operators do not need to care about routing policies, and should delegate the controller to select efficient paths for rules installation.

In Chapter 4, we present a new approach for the ORPP: trading routing for better resource efficiency. More precisely, a relax routing policy is used to leverage resources on all available paths when installing rules. Using this approach comes with the challenge of path selection and the fact that flow may follow a long path. Therefore, we propose a path heuristic, called *deflection*, that can identify paths consuming less switch memory than other paths and with low path stretch.

We then apply the above approach to address the offline ORPP, in which the set of flows is assumed known and stable in a period, and that allows to apply optimization techniques. We first formalize the offline ORPP as an Integer Linear Programming (ILP), and prove that the offline ORPP is NP-Hard by reducing it to the Knapsack problem. The ILP can be implemented on LP solvers (e.g., CPLEX) to find optimal rules satisfying policies and network constraints, while minimizing the costs, in terms of percentage of traffic processed by the default device. Second, we propose a Greedy-based heuristic, called OFFICER, to find rules in polynomial time complexity for large problem instances (e.g., large number of flows). Our simulation results in realistic scenarios show that OFFICER outperforms a random placement algorithm and closes to the optimum, without using long paths for flows.

In practice, flows are often not known in advance, and unpredictable due to measurement errors. In Chapter 5, we study the online ORPP, in which the set of flows is unknown and varies over time. To address this problem, existing controller platforms install rules reactively for all flows, from the first packet seen. However, this approach is not optimal, due to a large signaling overhead and that large flows may not be installed. Departing from the state of the art, we propose aOFFICER, an adaptive rules placement framework, that finds rules satisfying policies and constraints, while reducing the signaling overhead. The novelty of aOFFICER comes from its ability to detect and install rules for large flows on efficient paths, and to adjust its parameter according to traffic fluctuations. Our packet level simulation results, obtained with realistic traces and topologies, have confirmed the efficiency and the adaptability of aOFFICER, compared to existing solutions.

OFFICER and aOFFICER are basic blocks to realize the blackbox abstraction that allows to deploy high level, flexible endpoint policies using OpenFlow. In Chapter 6, we study a use case, in which the blackbox abstraction is leveraged to improve content delivery services in cellular networks. First, we argue that with OpenFlow, switches can support in-network caching functionalities (e.g., using our technical solution [NST13b]), and therefore, caches can be deployed everywhere to reduce network load and improve user experience. Second, we propose a versatile hybrid caching framework, called Arothron. The main idea of Arothon is to split cache space on each node into two parts: one for opportunistic caching and the other for preset caching. One one hand, the preset caches store the most popular contents to satisfy long term demands with high cache hit ratios, and the content placements are decided by solving a Mixed Integer Linear Programming (MILP) or using a greedy heuristic. On the other hand, the opportunistic caches store and replace

contents using LRU mechanism to absorb short term, bursty demands (e.g., flash crowd effects). The numerical simulation results reveal that combining opportunistic and preset caches can reduce the total link usage and the total latency further, compared to using preset caches only or opportunistic caches only. Moreover, the optimal ratio between the opportunistic and the preset cache storage depends on the node location.

7.2 Future Work

We believe the ORPP is still a challenging research area, and there remain interesting aspects to be studied and improved. In the following, we open some questions worth for future research.

7.2.1 Robust and Fault Tolerant Rules Placement

Normally, installed rules need to be updated to adapt with varying network conditions (e.g., changes in policies, network topology, user mobility). Most of proposed solutions recompute the rules placement to maximize or to minimize some performance metrics (e.g., total number of rules), which may come with the cost of computation overhead, signaling overhead, and setup delay.

Robustness and fault tolerance are also important factors, and should be taken into account when finding rules. On this purpose, robust optimization techniques [BBC11] might be a promising approach. Such techniques can model the uncertainty of inputs and produce robust rule placement solutions.

7.2.2 Impact of Default Devices

Recently, default devices, such as controllers and software switches, are used to offload the memory burden of OpenFlow switches (see Sec. 3.3.3.2). In such a case, default rules are often configured to redirect flow table miss from OpenFlow switches towards these devices [MYSG12, KARW14] for further processing. An important goal is to minimize the percentage of traffic processing by default devices, to avoid performance penalty (e.g., high processing delay).

The impact of the default devices' locations, and default rules' construction on the efficiency of rules placement are not well understood. It would be interesting to combine the controller placement [HSM12] and the OpenFlow Rules Placement Problem to overall optimize the network performance.

7.2.3 Multilevel Table Rules Placement

OpenFlow 1.0 uses a flat table model that cannot handle rule explosions. To address this issue, OpenFlow 1.1+ [Ope15b] supports multi-level flow tables and pipeline processing. Consequently, a large flow table on a switch can be split into smaller flow tables with less rules. Furthermore, some rules require to be placed in TCAM tables

while the remaining rules are placed in non-TCAM tables. The multi-tables feature has been implemented in commercial devices, such as NoviSwitch 1248 [Nov13].

How to benefit from multi-level architectures, how to leverage pipeline processing ability, which rules to place in which sub-tables should be taken into account for future research.

7.2.4 Network Function Virtualization

Networks today consist of a large number of various middleboxes and hardware appliances (e.g., Firewall, Deep Packet Inspections). Usually, launching a new service requires new hardware-based appliances, which increases the cost of investments, energy, integration, operation, and maintenance.

Network Function Virtualization (NFV) [Chi12] is a network architecture concept that advocates replacing network equipments by software-based functions on high volume servers, switches, and storage devices. Basically, NFV requires dynamic instantiations and placement of network functions, which can be provided by OpenFlow. For example, some L2, L3 network functions (e.g., routing, firewall) could be implemented directly by installing appropriate rules on OpenFlow switches. Furthermore, forwarding rules can be installed on OpenFlow switches to redirect flows through different network functions to realize service chaining in NFV.

How and where to place network functions, how traffic is routed through them are key challenges towards the deployment of NFV. Indeed, NFV is a fascinating use case and new rules placement solutions must be designed to implement NFV.

Bibliography

[3GP] 3GPP, *LTE architecture*, http://www.3gpp.org/LTE. (Cited in pages 80 and 81.)

[AAG+10] David Applegate, Aaron Archer, Vijay Gopalakrishnan, Seungjoon Lee, and K. K. Ramakrishnan, *Optimal content placement for a large-scale vod system*, Proceedings of the 6th International COnference (New York, NY, USA), Co-NEXT '10, ACM, 2010, pp. 4:1–4:12. (Cited in pages 80 and 85.)

[Abi] Abilene, *Abilene*, http://http://sndlib.zib.de. (Cited in page 49.)

[ACJ+07] David A. Applegate, Gruia Calinescu, David S. Johnson, Howard Karloff, Katrina Ligett, and Jia Wang, *Compressing Rectilinear Pictures and Minimizing Access Control Lists*, Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), SODA '07, Society for Industrial and Applied Mathematics, 2007, pp. 1066–1075. (Cited in page 25.)

[AD12] H. AhleHagh and S. Dey, *Video caching in radio access network: Impact on delay and capacity*, Wireless Communications and Networking Conference (WCNC), 2012 IEEE, Apr 2012, pp. 2276–2281. (Cited in page 94.)

[ADRC14] Kanak Agarwal, Colin Dixon, Eric Rozner, and John Carter, *Shadow macs: Scalable label-switching for commodity ethernet*, Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (New York, NY, USA), HotSDN '14, ACM, 2014, pp. 157–162. (Cited in pages 21, 27, 29 and 32.)

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat, *A scalable, commodity data center network architecture*, ACM SIGCOMM (2008), 63. (Cited in page 48.)

[AL06] Z. Abrams and Jie Liu, *Greedy is good: On service tree placement for in-network stream processing*, Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on, 2006, pp. 72–72. (Cited in page 86.)

[Apa15] Apache, *Hadoop*, 2015, https://hadoop.apache.org. (Cited in page 67.)

[APB09] Mark Allman, Vern Paxson, and Ethan Blanton, *Tcp congestion control*, Tech. report, RFC, 2009. (Cited in pages 14 and 65.)

[APVG14] A. Altieri, P. Piantanida, L.R. Vega, and C.G. Galarza, *A stochastic geometry approach to distributed caching in large wireless networks*, Wireless Communications Systems (ISWCS), 2014 11th International Symposium on, Aug 2014, pp. 863–867. (Cited in page 94.)

[AWM14] A. Asadi, Qing Wang, and V. Mancuso, *A survey on device-to-device communication in cellular networks*, Communications Surveys Tutorials, IEEE **16** (2014), no. 4, 1801–1819. (Cited in page 94.)

[BAAZ11] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang, *MicroTE: Fine Grained Traffic Engineering for Data Centers*, Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies (New York, NY, USA), CoNEXT '11, ACM, 2011, pp. 8:1–8:12. (Cited in pages 8, 34 and 58.)

[BAM10] Theophilus Benson, Aditya Akella, and David A. Maltz, *Network Traffic Characteristics of Data Centers in the Wild*, Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (New York, NY, USA), IMC '10, ACM, 2010, pp. 267–280. (Cited in pages 21, 23, 24, 27, 48, 60 and 67.)

[BBC11] Dimitris Bertsimas, David B. Brown, and Constantine Caramanis, *Theory and Applications of Robust Optimization*, SIAM Review **53** (2011), no. 3, 464–501. (Cited in page 99.)

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, *Web caching and zipf-like distributions: Evidence and implications*, INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 1, IEEE, 1999, pp. 126–134. (Cited in page 88.)

[Bel58] Richard Bellman, *On a Routing Problem*, Quarterly of Applied Mathematics **16** (1958), 87–90. (Cited in page 38.)

[BFCW09] Hitesh Ballani, Paul Francis, Tuan Cao, and Jia Wang, *Making routers last longer with viaggre*, USENIX NSDI (Berkeley, CA, USA), 2009, pp. 453–466. (Cited in page 40.)

[BM14] W. Braun and M. Menth, *Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-Based Software-Defined Networking*, Software Defined Networks (EWSDN), 2014 Third European Workshop on, Sept 2014, pp. 25–30. (Cited in pages 26 and 32.)

[CCK+10] James J. Cochran, Louis A. Cox, Pinar Keskinocak, Jeffrey P. Kharoufeh, and J. Cole Smith, *Milp software*, John Wiley and Sons, Inc., 2010. (Cited in page 12.)

[CDL07] Xu Cheng, Cameron Dale, and Jiangchuan Liu, *Understanding the characteristics of internet short video sharing: Youtube as a case study*, CoRR **abs/0707.3670** (2007), 1–9. (Cited in page 88.)

[Chi12] Margaret Chiosi, *Network Function Virtualisation White Paper*, https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012, accessed 19-Aug-2015. (Cited in page 100.)

[Chv83] V. Chvatal, *Linear programming*, Series of books in the mathematical sciences, W. H. Freeman, 1983. (Cited in pages 11 and 12.)

[Cis15] Cisco, *Cisco Visual Networking Index: Forecast and Methodology, 2014-2019*, 2015, http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html, accessed 21-Oct-2015. (Cited in pages 1 and 79.)

[CLP+12] Kideok Cho, Munyoung Lee, Kunwoo Park, T.T. Kwon, Yanghee Choi, and Sangheon Pack, *Wave: Popularity-based and collaborative in-network caching for content-oriented networks*, Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on, March 2012, pp. 316–321. (Cited in page 95.)

[CMT+11] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee, *DevoFlow: scaling flow management for high-performance networks*, SIGCOMM Comput. Commun. Rev. **41** (2011), no. 4, 254–265. (Cited in pages 20, 23, 27, 32, 34, 35, 54, 58 and 60.)

[CPZ04] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang, *Adaptive packet sampling for accurate and scalable flow measurement*, Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE, vol. 3, Nov 2004, pp. 1448–1452 Vol.3. (Cited in page 60.)

[CR14] J. Costa-Requena, *SDN integration in LTE mobile backhaul networks*, Information Networking (ICOIN), 2014 International Conference on, Feb 2014, pp. 264–269. (Cited in pages 80 and 95.)

[CRKMK14] J. Costa-Requena, M. Kimmerlin, J. Manner, and R. Kantola, *SDN optimized caching in LTE mobile networks*, Information and Communication Technology Convergence (ICTC), 2014 International Conference on, Oct 2014, pp. 128–132. (Cited in pages 80 and 95.)

[CSS10] Yasunobu Chiba, Yusuke Shinohara, and Hideyuki Shimonishi, *Source Flow: Handling Millions of Flows on Flow-based Nodes*, Proceedings of the ACM SIGCOMM 2010 Conference (New York, NY, USA), SIGCOMM '10, ACM, 2010, pp. 465–466. (Cited in pages 26, 27 and 32.)

[CTN+12] Yung-Chih Chen, Don Towsley, Erich M Nahum, Richard J Gibbens, and Yeon-sup Lim, *Characterizing 4g and 3g networks: Supporting mobility with multipath tcp*, Tech. report, UMass, 2012. (Cited in page 93.)

[DKVZ99] R.P. Draves, C. King, S. Venkatachary, and B.D. Zill, *Constructing optimal IP routing tables*, INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 1, Mar 1999, pp. 88–97 vol.1. (Cited in page 24.)

[DLOX15] Mianxiong Dong, He Li, Kaoru Ota, and Jiang Xiao, *Rule caching in SDN-enabled mobile access networks*, Network, IEEE **29** (2015), no. 4, 40–45. (Cited in page 30.)

[ENM+12] Suyong Eum, Kiyohide Nakauchi, Masayuki Murata, Yozo Shoji, and Nozomu Nishinaga, *Catt: Potential based routing with content caching for icn*, Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking (New York, NY, USA), ICN '12, ACM, 2012, pp. 49–54. (Cited in page 94.)

[Eri13] David Erickson, *The Beacon Openflow Controller*, Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (New York, NY, USA), HotSDN '13, ACM, 2013, pp. 13–18. (Cited in pages 4, 8, 57, 58, 60, 66, 67 and 69.)

[Fac] Facebook, *Facebook draws 1 billion users in a single day*, `http://www.theverge.com/2015/8/27/9217607/facebook-one-billion-daily-active-users`, accessed 21-Oct-2015. (Cited in page 1.)

[FHF+11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker, *Frenetic: a network programming language*, Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ICFP '11, ACM, 2011, pp. 279–291. (Cited in page 18.)

[Flo62] Robert W. Floyd, *Algorithm 97: Shortest path*, Commun. ACM **5** (1962), no. 6, 345–. (Cited in page 86.)

[Flo15] Floodlight, *Floodlight Controller*, 2015, `http://www.projectfloodlight.org/floodlight/`. (Cited in pages 4, 8, 57, 58, 60, 66, 67 and 69.)

[FLT+13] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, K.C. Ng, Vyas Sekar, and Scott Shenker, *Less pain, most of the gain: Incrementally deployable icn*,

SIGCOMM Comput. Commun. Rev. **43** (2013), no. 4, 147–158. (Cited in page 95.)

[Fou] Open Networking Foundation, *Open networking foundation*, https://www.opennetworking.org/about. (Cited in page 60.)

[Fou13] ———, *Optical transport working group otwg*, Open Networking Foundation ONF, 2013. (Cited in page 2.)

[GDM14] N. Golrezaei, A.G. Dimakis, and A.F. Molisch, *Scaling behavior for device-to-device communications with distributed caching*, Information Theory, IEEE Transactions on **60** (2014), no. 7, 4286–4298. (Cited in page 94.)

[GKP+08] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, *Nox: towards an operating system for networks*, SIGCOMM CCR **38** (2008), no. 3, 105–110. (Cited in pages 4, 8, 23, 57, 58, 60, 66, 67 and 69.)

[GMP14] F. Giroire, J. Moulierac, and T.K. Phan, *Optimizing rule placement in software-defined networks for energy-aware routing*, Global Communications Conference (GLOBECOM), 2014 IEEE, Dec 2014, pp. 2523–2529. (Cited in pages 20, 28, 29, 30, 32, 34, 54 and 58.)

[GNU13] GNU, *GNU Linear Programming Kit*, 2013, `http://www.gnu.org/software/glpk`. (Cited in page 12.)

[gri15] *Grid5000*, 2015, `https://www.grid5000.fr`. (Cited in page 67.)

[GSD+12] N. Golrezaei, K. Shanmugam, A.G. Dimakis, A.F. Molisch, and G. Caire, *Femtocaching: Wireless video content delivery through distributed caching helpers*, INFOCOM, 2012 Proceedings IEEE, Mar 2012, pp. 1107–1115. (Cited in page 94.)

[HCWL15] J. Huang, G. Chang, C. Wang, and C. Lin, *Heterogeneous Flow Table Distribution in Software-defined Networks*, Emerging Topics in Computing, IEEE Transactions on **PP** (2015), no. 99, 1–6. (Cited in pages 29 and 32.)

[HLGY14] Huawei Huang, Peng Li, Song Guo, and Baoliu Ye, *The Joint Optimization of Rules Allocation and Traffic Engineering in Software Defined Network*, May 2014, pp. 141–146. (Cited in pages 20, 26, 29, 30, 32, 34 and 58.)

[HLR+08] T.R. Henderson, M. Lacage, G.F. Riley, C. Dowell, and JB Kopena, *Network simulations with the ns-3 simulator*, SIGCOMM demonstration (2008). (Cited in page 66.)

[Hop00] Christian E Hopps, *Analysis of an equal-cost multi-path algorithm*, Tech. report, RFC, 2000. (Cited in page 58.)

[HSM12] Brandon Heller, Rob Sherwood, and Nick McKeown, *The controller placement problem*, Proceedings of the first workshop on Hot topics in software defined networks (New York, NY, USA), HotSDN '12, ACM, 2012, pp. 7–12. (Cited in pages 53 and 99.)

[HWC+13] Bing Han, Xiaofei Wang, Nakjung Choi, T. Kwon, and Yanghee Choi, *Amvs-ndn: Adaptive mobile video streaming and sharing in wireless named data networking*, Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on, Apr 2013, pp. 375–380. (Cited in page 94.)

[IBM] IBM, *CPLEX Solver*, `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`. (Cited in pages 8 and 12.)

[IMS13] A.S. Iyer, V. Mann, and N.R. Samineni, *SwitchReduce: Reducing switch state and controller involvement in OpenFlow networks*, IFIP Networking Conference, May 2013, pp. 1–9. (Cited in pages 21, 26, 27, 29, 30, 32 and 54.)

[INR15] INRIA, *INRIA NEF Clusters*, 2015, `https://wiki.inria.fr/ClustersSophia/Clusters_Home`. (Cited in pages 7 and 67.)

[JKM+13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al., *B4: Experience with a globally-deployed software defined WAN*, Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM, ACM, 2013, pp. 3–14. (Cited in pages 2, 18, 38 and 40.)

[JLG+14] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer, *Dynamic Scheduling of Network Updates*, SIGCOMM Comput. Commun. Rev. **44** (2014), no. 4, 539–550. (Cited in pages 27 and 41.)

[JST+09] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard, *Networking named content*, Proceedings of the 5th international conference on Emerging networking experiments and technologies (New York, NY, USA), CoNEXT '09, ACM, 2009, pp. 1–12. (Cited in page 94.)

[KARW14] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker, *Infinite CacheFlow in Software-defined Networks*, Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (New York, NY, USA), HotSDN '14, ACM, 2014, pp. 175–180. (Cited in pages 20, 21, 25, 29, 30, 31, 32, 34, 57 and 99.)

[KB14] Kalapriya Kannan and Subhasis Banerjee, *FlowMaster: Early Eviction of Dead Flow on SDN Switches*, Distributed Computing and Networking, Springer Berlin Heidelberg, 2014, pp. 484–498 (English). (Cited in page 23.)

[KHK13] Yossi Kanizo, David Hay, and Isaac Keslassy, *Palette: Distributing tables in software-defined networks*, INFOCOM, Apr. 2013, pp. 545–549. (Cited in pages 8, 25, 28, 29, 30, 32, 34, 43, 44, 54 and 57.)

[KLC+14] Eun-Do Kim, Seung-Ik Lee, Yunchul Choi, Myung-Ki Shin, and Hyoung-Jun Kim, *A flow entry management scheme for reducing controller overhead*, Advanced Communication Technology (ICACT), 2014 16th International Conference on, Feb 2014, pp. 754–757. (Cited in pages 23 and 32.)

[KLRW13] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker, *Optimizing the "one big switch" abstraction in software-defined networks*, Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (New York, NY, USA), CoNEXT '13, ACM, 2013, pp. 13–24. (Cited in pages 4, 8, 19, 20, 25, 28, 29, 30, 32, 34, 38, 43, 44, 54 and 57.)

[KNS+15] Mael Kimmerlin, Xuan Nam Nguyen, Damien Saucez, Jose Costa-Requena, and Thierry Turletti, *Arothron: a Versatile Caching Framework for LTE*, Tech. report, INRIA and Aalto University, 2015, under submission. (Cited in pages 9 and 79.)

[KO90] Sir Maurice Kendall and J. Keith Ord, *Time Series 3rd Edition*, Hodder Arnold, Great Britain, 1990. (Cited in pages 14 and 65.)

[KREV+15] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, *Software-defined networking: A comprehensive survey*, IEEE Communications Surveys & Tutorials **103** (2015), no. 1, 14–76. (Cited in pages 1, 2, 7, 18 and 34.)

[KSG+09] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken, *The Nature of Data Center Traffic: Measurements & Analysis*, Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference (New York, NY, USA), IMC '09, ACM, 2009, pp. 202–208. (Cited in pages 23, 34, 58 and 60.)

[KSP+14] Masayoshi Kobayashi, Srini Seetharaman, Guru Parulkar, Guido Appenzeller, Joseph Little, Johan van Reijendam, Paul Weissmann, and Nick McKeown, *Maturing of OpenFlow and Software-defined Networking through deployments*, Computer Networks **61** (2014), 151–175. (Cited in pages 2 and 20.)

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown, *A network in a laptop: rapid prototyping for software-defined networks*, Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, 2010. (Cited in page 7.)

[LHO+14] Wei Liu, Ao Hong, Liang Ou, Wenchao Ding, and Ge Zhang, *Prediction and correction of traffic matrix in an ip backbone network*, Performance Computing and Communications Conference (IPCCC), 2014 IEEE International, Dec 2014, pp. 1–9. (Cited in page 37.)

[LKA13] Bu Sung Lee, Renuga Kanagavelu, and Khin Mi Mi Aung, *An efficient flow cache algorithm with improved fairness in Software-Defined Data Center Networks*, Proceedings of the 2013 IEEE 2nd International Conference on Cloud Networking, CloudNet 2013 (2013), 18–24. (Cited in page 23.)

[LLG14] He Li, Peng Li, and Song Guo, *Morule: Optimized rule placement for mobile users in sdn-enabled access networks*, Global Communications Conference (GLOBECOM), 2014 IEEE, Dec 2014, pp. 4953–4958. (Cited in pages 20, 29, 30, 32, 34 and 58.)

[LMT10] A.X. Liu, C.R. Meiners, and E. Torng, *TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs*, Networking, IEEE/ACM Transactions on **18** (2010), no. 2, 490–500. (Cited in page 25.)

[LYL14] Shouxi Luo, Hongfang Yu, and Le Min Li, *Fast incremental flow table aggregation in SDN*, Computer Communication and Networks (ICCCN), 2014 23rd International Conference on, Aug 2014, pp. 1–8. (Cited in pages 27, 28 and 32.)

[MAB+08a] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *Openflow: enabling innovation in campus networks*, ACM SIGCOMM Computer Communication Review **38** (2008), no. 2, 69–74. (Cited in pages 2, 4, 17 and 18.)

[MAB+08b] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner, *Openflow: Enabling innovation in campus networks*, SIGCOMM Comput. Commun. Rev. **38** (2008), no. 2, 69–74. (Cited in pages 38, 80 and 83.)

[Mat] Matplotlib, *Matplotlib Python Plotting*, `matplotlib.org/`. (Cited in page 7.)

[MC12] Jeffrey C. Mogul and Paul Congdon, *Hey, You Darned Counters!: Get off My ASIC!*, Proceedings of the First Workshop on Hot Topics in

Software Defined Networks (New York, NY, USA), HotSDN '12, ACM, 2012, pp. 25–30. (Cited in page 35.)

[Meg10] Natarajan Meghanathan, *Impact of the gauss-markov mobility model on network connectivity, lifetime and hop count of routes for mobile ad hoc networks*, Journal of networks **5** (2010), no. 5, 509–516. (Cited in page 88.)

[MLT12] C.R. Meiners, A.X. Liu, and E. Torng, *Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs*, Networking, IEEE/ACM Transactions on **20** (2012), no. 2, 488–500. (Cited in page 25.)

[MNL+10] Daekyeong Moon, Jad Naous, Junda Liu, Kyriakos Zarifis, Martin Casado, Teemu Koponen, Scott Shenker, and Lee Breslau, *Bridging the Software/Hardware Forwarding Divide*, 2010, UC Berkeley. (Cited in page 21.)

[Moy89] John Moy, *Ospf specification*, Tech. report, RFC, 1989. (Cited in page 58.)

[MT13] Bernhard Meindl and Matthias Templ, *Analysis of commercial and free and open source solvers for the cell suppression problem*, Trans. Data Privacy **6** (2013), no. 2, 147–159. (Cited in page 12.)

[MYSG12] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan, *vCRIB: Virtualized Rule Management in the Cloud*, USENIX HotCloud (Berkeley, CA, USA), 2012, pp. 23–23. (Cited in pages 4, 18, 20, 21, 30, 31, 32, 54 and 99.)

[MYSG13] ———, *Scalable Rule Management for Data Centers*, NSDI'13 Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (2013), 157–170. (Cited in page 21.)

[Net15] NetworkX, *NetworkX*, 2015, `http://networkx.lanl.gov/`. (Cited in pages 7 and 73.)

[Nev14] Miguel Cardoso Neves, *On Time-based Strategies for Optimizing Flow Tables in SDN*, Dec 2014, Master Thesis, Department of Computer Science, The Federal University of Rio Grande do Sul, Brazil. (Cited in pages 23 and 24.)

[Ngu12] X.N. Nguyen, *Software Defined Networking in Wireless Mesh Network*, Msc. thesis, INRIA, UNSA, August 2012. (Cited in page 10.)

[NHL+13] Yukihiro Nakagawa, Kazuki Hyoudou, Chunghan Lee, Shinji Kobayashi, Osamu Shiraki, and Takeshi Shimizu, *DomainFlow: Practical Flow Management Method Using Multiple Flow Tables in Com-*

modity Switches, ACM CoNEXT, ACM, 2013, pp. 399–404. (Cited in pages 27, 29, 30, 32, 34 and 54.)

[NMN+14] B.A.A. Nunes, M. Mendonca, Xuan-Nam Nguyen, K. Obraczka, and T. Turletti, *A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks*, Communications Surveys Tutorials, IEEE **16** (2014), no. 3, 1617–1634. (Cited in pages 2, 3, 7, 9 and 18.)

[Nov13] NoviFlow, *NoviSwitch 1248 High Performance Switch*, Jan 2013, http://www.nvc.co.jp/pdf/product/noviflow/NoviSwitch1248Datasheet.pdf. (Cited in page 100.)

[NSBT14] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti, *Optimizing Rules Placement in OpenFlow Networks: Trading Routing for Better Efficiency*, Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (New York, NY, USA), HotSDN '14, ACM, 2014, pp. 127–132. (Cited in pages 10, 30, 37, 38, 40, 43 and 55.)

[NSBT15a] Xuan Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti, *OFFICER: A general Optimization Framework for OpenFlow Rule Allocation and Endpoint Policy Enforcement*, IEEE INFOCOM, April 2015. (Cited in pages 10, 20, 26, 27, 28, 29, 30, 31, 32, 34 and 37.)

[NSBT15b] ———, *Rules Placement Problem in OpenFlow Networks: a Survey*, IEEE Communications Surveys & Tutorials (2015), 1–12. (Cited in pages 9 and 17.)

[NST13a] Xuan-Nam Nguyen, Damien Saucez, and Thierry Turletti, *Efficient caching in Content-Centric Networks using OpenFlow*, INFOCOM 2013 Student Workshop (Turin, Italy), Feb 2013 (Anglais). (Cited in pages 7, 9, 10 and 95.)

[NST13b] ———, *Providing CCN functionalities over OpenFlow switches*, Inria research report 00920554, INRIA, August 2013. (Cited in pages 7, 9, 10, 95 and 98.)

[Ope15a] OpenDaylight, *OpenDaylight Controller*, http://www.opendaylight.org/, 2015, accessed 19-Aug-2015. (Cited in pages 57, 58 and 59.)

[Ope15b] OpenFlow, *OpenFlow Switch Specification*, http://www.opennetworking.org/, 2015, accessed 19-Aug-2015. (Cited in pages 3, 19, 20, 22, 23, 24, 33, 54, 66 and 99.)

[Ope15c] OpenvSwitch, *OpenvSwitch*, http://openvswitch.org/, 2015, accessed 19-Aug-2015. (Cited in pages 21 and 31.)

[Pan] Pandas, *Pandas Data Analysis*, pandas.pydata.org/. (Cited in page 7.)

[PCP12] Ioannis Psaras, Wei Koong Chai, and George Pavlou, *Probabilistic in-network caching for information-centric networks*, Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking (New York, NY, USA), ICN '12, ACM, 2012, pp. 55–60. (Cited in page 94.)

[POB+14] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal, *Fastpass: A Centralized "Zero-Queue" Datacenter Network*, ACM SIGCOMM, August 2014. (Cited in page 38.)

[Pos81] Jon Postel, *Internet protocol*, STD 5, RFC Editor, September 1981, http://www.rfc-editor.org/rfc/rfc791.txt. (Cited in page 27.)

[PRO15] PRONTO, *OpenFlow Switch Pronto 3290*, 2015, http://www4.ncsu.edu/~acbabaog/openflow/openflow_pronto.pdf. (Cited in page 7.)

[RDH+13] B.A. Ramanan, L.M. Drabeck, M. Haner, N. Nithi, T.E. Klein, and C. Sawkar, *Cacheability analysis of http traffic in an operational lte network*, Wireless Telecommunications Symposium (WTS), 2013, Apr 2013, pp. 1–8. (Cited in page 94.)

[RHC+15] M Rifai, N Huin, C Caillouet, F Giroire, D Lopez-Pacheco, J Moulierac, and G Urvoy-Keller, *Too many SDN rules? Compress them with MINNIE*, Global Communications Conference (GLOBECOM), IEEE, Jul 2015, pp. 1–6. (Cited in pages 26 and 32.)

[RR12] D. Rossi and G. Rossini, *On sizing ccn content stores by exploiting topological information*, Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on, March 2012, pp. 280–285. (Cited in pages 94 and 95.)

[Ryu15] Ryu, *Ryu controller*, http://osrg.github.com/ryu/, 2015, accessed 19-Aug-2015. (Cited in pages 57 and 58.)

[SCF+12] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter, *PAST: Scalable Ethernet for Data Centers*, Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (New York, NY, USA), CoNEXT '12, ACM, 2012, pp. 49–60. (Cited in pages 4, 18, 20, 59 and 66.)

[Sch86] Alexander Schrijver, *Theory of linear and integer programming*, John Wiley & Sons, Inc., New York, NY, USA, 1986. (Cited in pages 11, 12 and 13.)

[SCP13] Lorenzo Saino, Cosmin Cocora, and George Pavlou, *A Toolchain for Simplifying Network Simulation Setup*, SIMUTOOLS, 2013. (Cited in pages 7 and 49.)

[SMW02] Neil Spring, Ratul Mahajan, and David Wetherall, *Measuring ISP topologies with Rocketfuel*, SIGCOMM CCR **32** (2002), no. 4, 133–145. (Cited in page 48.)

[Son15] Hardik Soni, *DiG: Data centers in the Grid*, 2015, `https://team.inria.fr/diana/software/dig/`. (Cited in page 67.)

[TSS+97] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden, *A survey of active network research*, Communications Magazine, IEEE **35** (1997), no. 1, 80–86. (Cited in page 2.)

[TZ01] Mikkel Thorup and Uri Zwick, *Compact routing schemes*, SPAA, 2001. (Cited in page 38.)

[UHD] UHD, *Understanding ultra high definition television*, Ericsson white paper, 2015. (Cited in page 80.)

[VCB+15] Stefano Vissicchio, Luca Cittadini, Olivier Bonaventure, Xie Geoffrey, and Laurent Vanbever, *On the Co-Existence of Distributed and Centralized Routing Control-Planes*, IEEE INFOCOM, April 2015. (Cited in pages 18 and 35.)

[VKF12] Andreas Voellmy, Hyojoon Kim, and Nick Feamster, *Procera: a language for high-level reactive network control*, Proceedings of the first workshop on Hot topics in software defined networks (New York, NY, USA), HotSDN '12, ACM, 2012, pp. 43–48. (Cited in page 18.)

[VPMB14] Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya, *Effective switch memory management in OpenFlow networks*, Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14 (2014), 177–188. (Cited in pages 23, 24, 32 and 73.)

[VWY+13] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak, *Maple: Simplifying sdn programming using algorithmic policies*, Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (New York, NY, USA), SIGCOMM '13, ACM, 2013, pp. 87–98. (Cited in page 18.)

[WBR11] Richard Wang, Dana Butnariu, and Jennifer Rexford, *Openflow-based server load balancing gone wild*, Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Berkeley, CA, USA), Hot-ICE'11, USENIX Association, 2011, pp. 12–12. (Cited in page 20.)

[WCT+14] Xiaofei Wang, Min Chen, T. Taleb, A. Ksentini, and V. Leung, *Cache in the air: exploiting content caching and delivery techniques for 5g systems*, Communications Magazine, IEEE **52** (2014), no. 2, 131–139. (Cited in page 94.)

[WJP+13] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park, *Comparison of caching strategies in modern cellular backhaul networks*, Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (New York, NY, USA), MobiSys '13, ACM, 2013, pp. 319–332. (Cited in pages 80 and 94.)

[WNTS16] Michelle Wetterwald, Xuan Nam Nguyen, Thierry Turletti, and Damien Saucez, *SDN for Public Safety Networks*, Tech. report, INRIA, 2016, under submission. (Cited in page 9.)

[WSW+14] Philip Wette, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl, *MaxiNet: Distributed Emulation of Software-Defined Networks*, IFIP Networking Conference, 2014. (Cited in pages 49 and 52.)

[WWJ+15] Xin Wang, Cheng Wang, Changjun Jiang, Lei Yang, Zhong Li, and Xiaobo Zhou, *Rule Optimization for Real-Time Query Service in Software-Defined Internet of Vehicles*, CoRR **abs/1503.05646** (2015), 1–12. (Cited in pages 23, 26, 30, 32, 34 and 58.)

[XWF+15] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, D. Niyato, and Haiyong Xie, *A survey on software-defined networking*, Communications Surveys Tutorials, IEEE **17** (2015), no. 1, 27–51. (Cited in pages 2, 7, 18 and 94.)

[XZZ+14] Liang Xie, Zhifeng Zhao, Yifan Zhou, Gang Wang, Qianlan Ying, and Honggang Zhang, *An adaptive scheme for data forwarding in software Defined Network*, Wireless Communications and Signal Processing (WCSP), 2014 Sixth International Conference on, Oct 2014, pp. 1–5. (Cited in pages 23, 24, 32 and 73.)

[YRFW10] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang, *Scalable flow-based networking with DIFANE*, SIGCOMM CCR **41** (2010), no. 4, 351–362. (Cited in pages 20, 29, 30, 32, 35 and 54.)

[ZFLJ15] Huikang Zhu, Hongbo Fan, Xuan Luo, and Yaohui Jin, *Intelligent timeout master: Dynamic timeout for SDN-based data centers*, Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium, May 2015, pp. 734–737. (Cited in pages 23 and 32.)

[ZGL14] Adam Zarek, Yashar Ganjali, and David Lie, *OpenFlow Timeouts Demystified*, 2014, Master Thesis, Department of Computer Science, University of Toronto, Canada. (Cited in pages 23 and 32.)

[ZIL+14] Shuyuan Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik, *An Adaptable Rule Placement for Software-Defined Networks*, Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 2014, pp. 88–99. (Cited in pages 25, 28, 29, 30, 32, 34 and 57.)

[ZLL13] Guoqiang Zhang, Yang Li, and Tao Lin, *Caching in information centric networking: A survey*, Comput. Netw. **57** (2013), no. 16, 3128–3141. (Cited in pages 82, 94 and 95.)