Technische Universität München
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Kommunikationsnetze

Management of Programmable Control and Data Plane Towards Flexible Softwarized Networks

Mu He, M.Sc.

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr.-Ing. Ulf Schlichtmann
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. Wolfgang Kellerer
	2.	Prof. Dr. Martina Zitterbart

Die Dissertation wurde am 10.06.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 28.10.2020 angenommen.

Management of Programmable Control and Data Plane Towards Flexible Softwarized Networks

Mu He, M.Sc.

28.10.2020

Abstract

Emerging networking applications and varying user demands are challenging today's communication networks, which rely heavily on rigid networking devices and protocol stack and lack the ability to dynamically allocate networking resources on the fly. Novel networking techniques, i.e., Software-Defined Networking (SDN) and Programmable Data Plane (PDP), are proposed to provide higher flexibility in terms of dynamic network service provisioning in the era of network softwarization. In particular, SDN enables the programmability of the control plane, which can assist a fine-grained networking traffic control from a global perspective, whereas PDP enables the programmability of the data plane, which can aid the customization of packet processing functionality. Both techniques promise excellent opportunities to accommodate changing networking conditions and requirements. However, how to manage the control and data plane, especially under dynamic scenarios and with the target of *flexibility*, is still challenging and has not been widely studied in the literature. The goal of this thesis is to close this research gap by addressing the following four challenges.

First, to make a fair comparison between different network designs in terms of flexibility, a rigorous interpretation and a formal definition of a flexibility measure need to be derived. This thesis contributes to an interpretation by unfolding the intrinsic properties of a flexible network, i.e., support of changes in the network requirements in a timely and efficient manner. The network supports the changes either directly without any internal adaptation or by adapting its internal states such as the flow routes and functions' locations. If an adaptation is necessary, the time and cost factors need to be explicitly considered: the former indicates that the adaptation should accomplish under time limitation; the later indicates that the cost to realize the adaptation should also be lower than a threshold. Following the interpretation, this thesis elaborates on a presented flexibility metric and discusses the metric with a detailed survey and comprehensive use cases.

The second challenge is modeling and optimizing the placement opportunities that are offered by the flexibility of the dynamic control plane. SDN leverages the distributed control plane architecture to address the scalability and latency issues and benefits from the flexibility that enables dynamic migration of controllers and reassignment of switches. However, such architecture and flexibility introduce a new problem, i.e., where to place controller instances and assign switches to the controllers towards a specific performance metric, which is algorithmically hard. Average end-to-end flow setup time is the primary performance metric of the placement problem in this thesis, as it reveals the overhead of latency between the control and data plane during configuring flow rules along the path. The challenge here is to formulate the optimization problem based on the modeling of end-to-end flow setup time. Besides, when considering the dynamic case with multiple time-slots, changing from one placement to another placement incurs non-negligible adaptation cost, which should also be modeled in the optimization problem. Efficient algorithms are also proposed accordingly to solve the optimization problems.

The third challenge is the evaluation and optimization of the flexibility of the dynamic control plane. Different design choices, e.g., the number of controllers, have been compared considering classical performance metrics such as control latency and reliability; however, an explicit analysis and comparison in terms of flexibility are still missing in the literature. Therefore, this thesis proposes an evaluation framework that calculates the time that the control plane needs for adaptation, comprising controller migration and switch reassignment. The framework also generates a representative set of demand changes in the form of flow profiles as the input for the comparison between different design choices. Furthermore, the locations of the Data Centers (DCs) to host the controllers also have a critical impact on the highest possible flexibility that the dynamic control plane can have. In this regard, this thesis takes the flexibility analysis a step further: it presents a mathematical programming model that considers the underlying adaptation time and cost and optimally decides the static DC locations over a time horizon of planning.

The last challenge is the design of the programmable data plane towards runtime reconfiguration. Data plane programmability, especially the target-independent programming language P4, enables the customization of packet processing functionality that can be applied to various types of forwarding devices. With programmability, the functionality of each device can be reconfigured at runtime. To leverage these advantages, this thesis proposes an architecture that manages heterogeneous P4 data plane devices and explores the capability of reconfiguration without service interruption. One critical requirement of data plane reconfiguration is to ensure the consistency, i.e., the states within the data plane need to be maintained. The maintenance should be efficient and only target the necessary state variables. Accordingly, this thesis presents a suite of algorithms that analyze P4 programs and locate states that need to be preserved during reconfiguration. Moreover, the performance of the data plane is comprehensively measured to understand the potential overhead of reconfiguration.

Kurzfassung

Neue Netzwerkanwendungen und unterschiedliche Benutzeranforderungen stellen das heutige Kommunikationsnetz vor eine Herausforderung, die durch feste Implementierung von Netzgeräten und Protokollstapeln hervorgeführt wird. Dadurch sind die Netze nicht in der Lage, Ressourcen vollumfänglich dynamisch im laufenden Betrieb zuzuweisen. Neuartige Netztechniken werden vorgeschlagen, d. h. Software-Defined Networking (SDN) und Programmable Data Plane (PDP), um eine höhere Flexibilität bei der Bereitstellung dynamischer Netzdienste im Zeitalter von Network Softwarization zu gewährleisten. Insbesondere ermöglicht SDN die Programmierbarkeit der Steuerebene, was eine fein abgestimmte Netzverkehrssteuerung aus globaler Sicht unterstützen kann, während PDP die Programmierbarkeit der Datenebene ermöglicht, was die Anpassung der Paketverarbeitungsfunktionalität fördern kann. Beide Techniken bieten hervorragende Möglichkeiten, um sich ändernden Netzbedingungen und -anforderungen anzupassen. Die Verwaltung der Steuerungs- und Datenebene, insbesondere unter dynamischen Szenarien und mit dem Ziel der Flexibilität, ist jedoch immer noch eine Herausforderung und wurde in der wissenschaftlichen Literatur des Standes der Technik nicht umfassend untersucht. Ziel dieser Arbeit ist es, diese Forschungslücke zu schließen, indem die folgenden vier Herausforderungen angegangen werden.

Um einen fairen Vergleich zwischen verschiedenen Netzdesigns hinsichtlich der Flexibilität zu ermöglichen, müssen zunächst eine rigorose Interpretation und eine formale Definition des Flexibilitätsmaßes abgeleitet werden. Diese Arbeit trägt zu einer Interpretation bei, indem die intrinsischen Eigenschaften eines flexiblen Netzwerks betrachtet werden, d. h. die rechtzeitige Unterstützung von Änderungen der Netzanforderungen mit geringem Aufwand. Das Netzwerk unterstützt die Änderungen entweder direkt ohne interne Anpassung oder durch Anpassung seiner internen Zustände wie der Flüsse und der Platzierungen der Funktionen. Wenn eine Anpassung erforderlich ist, müssen die Zeit- und Kostenfaktoren explizit berücksichtigt werden: Ersteres gibt an, dass die Anpassung vor einer vordefinierten Frist erfolgen sollte; Letzteres zeigt an, dass die Kosten für die Realisierung der Anpassung ebenfalls unter einem Schwellenwert liegen sollten. Im Anschluss an die Interpretation erörtert diese Arbeit eine vorgestellte Flexibilitätsmetrik und diskutiert sie anhand von Use Cases.

Die zweite Herausforderung ist die Modellierung und Optimierung der Platzierungsmöglichkeiten der dynamischen Steuerebene. SDN nutzt die Architektur der verteilten Steuerebene, um die Skalierbarkeits- und Latenzprobleme zu lösen, und profitiert von der Flexibilität, die die dynamische Migration von Controllern und die Neuzuweisung von Switches ermöglicht. Eine solche Architektur und Flexibilität führt jedoch zu einem neuen Problem, d. h. wo Controller-Instanzen platziert und die Controller Switches für eine bestimmte Leistungsmetrik zugewiesen werden müssen. Die durchschnittliche End-to-End-Flow-Setup-Time ist die primäre Leistungsmetrik für das Platzierungsproblem in dieser Arbeit, da sie den Overhead der Latenz zwischen Steuerung und Datenebene während der Konfiguration der Flow-Regeln

entlang des Pfads aufzeigt. Die Herausforderung besteht darin, das Optimierungsproblem basierend auf der Modellierung der End-to-End-Flow-Setup-Time zu formulieren. Wenn der dynamische Fall mit mehreren Zeitschlitzen betrachtet werden sollte, entstehen beim Wechsel von einer Platzierung zu einer anderen Platzierung nicht zu vernachlässigende Anpassungskosten, die auch im Optimierungsproblem modelliert werden sollten. Die vorgeschlagenen Optimierungsprobleme sind algorithmisch komplex; Daher werden heuristische Algorithmen entsprechend vorgeschlagen.

Die dritte Herausforderung ist die Bewertung und Optimierung der Flexibilität der dynamischen Steuerebene. Verschiedene Entwurfsmöglichkeiten der dynamischen Steuerebene, z. B. die Anzahl der Steuerungen, werden unter Berücksichtigung klassischer Leistungsmetriken wie Steuerungslatenz und Zuverlässigkeit verglichen; Eine explizite Analyse und ein Vergleich hinsichtlich der Flexibilität fehlen jedoch noch in der Literatur. Daher wird in dieser Arbeit ein Bewertungsrahmen vorgeschlagen, der die Zeit berechnet, die die Steuerebene für die Anpassung benötigt, einschließlich Controller-Migration und Switch-Neuzuweisung. Der Bewertungsrahmen generiert eine repräsentative Menge von Anfragen in Form von Flussprofilen als Eingabe für den Vergleich zwischen verschiedenen Entwurfsoptionen. Darüber hinaus ist der Controller eine Software, die in Rechenzentren (DCs) ausgeführt werden muss. Die geografische Entfernung zwischen DCs kann in Wide Area Networks (WANs) beträchtlich sein, was zu einer unerträglich langen Anpassungszeit führt. Somit haben die Positionen der DCs einen kritischen Einfluss auf die höchstmögliche Flexibilität der dynamischen Steuerebene. In diesem Zusammenhang wird in dieser Arbeit ein mathematisches Programmiermodell vorgestellt, das die zugrunde liegende Anpassungszeit und die zugrunde liegenden Kosten berücksichtigt und die statischen DC-Standorte über einen Zeithorizont der Planung optimal festlegt.

Die letzte Herausforderung ist der Entwurf einer programmierbaren Datenebene für die Rekonfiguration der Laufzeit. Die Programmierbarkeit der Datenebene, insbesondere die zielunabhängige Programmiersprache P4, ermöglicht die Anpassung der Paketverarbeitungsfunktionalität, die auf verschiedene Arten von Weiterleitungsgeräten angewendet werden kann. Durch die Programmierbarkeit kann jedes Gerät zur Laufzeit seine eigene Funktionalität neu konfigurieren. Um diese Vorteile zu nutzen, wird in dieser Arbeit eine Architektur vorgeschlagen, die die heterogene P4-Datenebenengeräte verwaltet. Außerdem wird die Möglichkeit der Rekonfiguration ohne Dienstunterbrechung untersucht. Eine kritische Anforderung der Rekonfiguration der Datenebene besteht darin, die Konsistenz sicherzustellen, d. h. die Zustände innerhalb der Datenebene müssen beibehalten werden. Die Wartung sollte effizient sein und nur auf die erforderlichen Statusvariablen abzielen und die Temporären ignorieren. Dementsprechend stellt diese Arbeit eine Reihe von Algorithmen dar, die ein P4-Programm analysieren und Zustände lokalisieren, die während der Rekonfiguration beibehalten werden müssen. Darüber hinaus wird die Leistung der Datenebene während der Rekonfiguration umfassend gemessen, um den potenziellen Overhead zu verstehen.

Contents

Chapter 1

Introduction

Communication networks have become a critical building block of our digitalized society. Billions of devices, ranging from computers to mobile devices, are connected from nearly all parts around the globe and break all types of geographical boundaries. The last ten years have witnessed the emergence of Internet of Things (IoT), 5G, and tactile networks, which further revolutionize the way that human interaction with the physical world [22]–[24]. The various types of networking applications often place distinct requirements on the underlying networks. Some of them introduce stringent requirements in terms of performance (e.g., throughput and latency [25]) and dependability (e.g., failover recovery [26]). Others require sufficient connection coverage and massive end-to-end communication. From the functionality perspective, a heterogeneous set of functions need to be supported, including basic L2 to L4 processing, tunneling, load balancing, congestion control, and intrusion detection, etc [27], [28]. Network operators need to deliver different services under the respective negotiated Quality of Service (QoS) within their physical infrastructure efficiently.

However, it becomes more challenging, and sometimes even impossible, to provision the networking services and support an extensive function set with rigid legacy networking devices. Traditional networking devices are inter-connected in a homogeneous setting and needed to be configured manually [29], which is not suitable for the new requirements. The requirements of different applications can fluctuate on time scales from milliseconds to minutes and also show significant geographical patterns [30], [31]. Moreover, those devices only support limited protocols lacking efficient mechanisms to accommodate the fast-changing requirements in a timely manner [32]. Rolling out new protocols and functionalities is also not possible without purchasing new costly equipment or interrupting the data flow, which hinders the speed of development and deployment of new networking algorithms. As a result, legacy communication networks fail to reveal flexibility in managing physical resources and operating multiple networking applications [1]. Indeed, both network operators and end-users anticipate the emergence of better technology concepts to cope with this issue.

Two novel networking paradigms – namely Software-Defined Networking (SDN) [33] and Programmable Data Plane (PDP) [34] – are proposed to address the new requirements of communication networks. SDN is seen as a key enabler to program the control plane applications with a global knowledge of the undelying network, while PDP pushes the programmability down to the data plane and empowers customization of packet processing behaviors. They target different components of a network and complement each other towards the softwarization of the control and data plane.

SDN decouples the control plane from the data plane, e.g., switches and routers, and encloses the control plane functionalities into a new type of networking entity named controller. The controller is

responsible for handling the networking traffic and generating flow rules for the underlying forwarding devices. The open interface between the two planes, e.g., OpenFlow [180], provides a means to send the flow rules to the data plane and request flow setup from the control plane, which can potentially improve the efficiency of networking resource management. With such programmability, SDN can support the demands coming from applications and services, translate them into high-level policies, and realize them on each forwarding device, which reveals the concept of Network Operating System (NOS), e.g., Open Networking Operating System (ONOS) [35].

PDP, on the other hand, abstracts the basic packet processing constructs, such as parser and matching tables, and enables network designers to leverage these constructs by freely specify their type, sequence, and semantics to build their own packet processing pipelines. PDP enables rapid device functionality upgrade cycles and avoids "big-fat" devices supporting all networking protocols, which incurs inefficiencies and opens sources for failures [34]. A networking device only contains the logic that it needs, which can also be reconfigured on the fly. Another feature of PDP is target-independence: the same packet processing description can be supported to multiple platforms, including software switches [181], Netronome SmartNICs [182], Field-Programmable Gate Arrays (FPGAs) [36], and reconfigurable Application-Specific Integrated Circuits (ASICs) [183].

Combining SDN and PDP offers an excellent opportunity to combine their advantages: dynamic and flexible management on the resources of both the control and data plane to accommodate changing networking conditions and requirements. With both paradigms, it is possible to create a data plane with forwarding devices, each with a distinct packet processing pipeline, controlled by the same control plane that processes the policies specified by network operators. The existing networking infrastructure can also foresee the implementation, evaluation, and roll-out of new functionalities at runtime, comprising updates from both the control and data plane [37], [38].

The management of softwarized networks needs a proper design of algorithms [33], e.g., resource allocation [39], function placement [40], and flow routing [41]. With an understanding of the problem's nature, the algorithm design starts with mathematical modeling. Depending on the categories of the problem, different solution methodologies can be applied, which can be classified into two categories: the ones deriving optimal solutions and the ones deriving sub-optimal solutions. Optimization approaches such as Integer Linear Programming (ILP) can find the optimal solutions with the help of linear optimizer but may suffer from runtime issues. More efficient sub-optimal algorithms either leverage classical algorithm frameworks, e.g., greedy and simulated annealing, or develop a unique heuristic solution from scratch.

Meanwhile, Machine Learning (ML) also becomes a viable alternative to solve decision-making problems of resource management [42]. The recent success of applying ML techniques to other challenging research domains [43]–[45] hints that the idea might not be too far away. Given a feature vector that comprises the problem's input, the algorithm can predict a class that the feature vector belongs to, which is mapped to a particular decision. Indeed, ML techniques are well-suited to accelerate resource management tasks. Decisions made in these tasks by non-ML algorithms are often highly repetitive, hence generating sufficient training data for the ML algorithms. Complex decision-making systems can be modeled as, e.g., decision trees or neural networks, that implicitly builds a relation between input variables and output decisions. Furthermore, by the continuation of learning from new decisions, the ML algorithm can be optimized accordingly towards new problem settings.

Although increased flexibility is often claimed in the context of SDN and PDP, a formal argument for support is missing in the respective literature [1], [2]. Besides, it is often judged from an intuitive

notion of the ability to adapt, which neglects potential trade-offs that might be induced by increasing the capability of networking systems. To formally analyze the impact of flexibility, strengthen the argument for or against it, and enable meaningful trade-off analysis, flexibility needs to be quantified formally in the first place. The quantification allows the explicit design of networks to increase flexibility and a better understanding of the relation between the flexibility metric and the induced cost, time, and system complexity.

Research on managing programmable control and data plane leveraging SDN and PDP techniques to increase flexibility yields many open questions. Accordingly, the **objectives** of this thesis can be summarized as follows.

- Since the research community still misses a uniform understanding of flexibility, which hinders a fair comparison of different research proposals, it is critical to first formally propose the mathematical definition of flexibility, which can be applied to different networking use cases.
- As the locations of controllers and assignment of switches pose a massive impact on QoS metrics in SDN networks [46], [47], the second objective is to optimize the resource of the control plane, i.e., controllers. Comprehensive analysis of realistic network topologies facilitates the study of the impact of control plane parameters on the control plane performance indicators.
- To deepen the understanding of the flexibility that the dynamic control plane reveals, the next goal is to measure the flexibility metric numerically and compare different design choices. Furthermore, the valid measure enables the explicit optimization of the flexibility that the control plane can deliver, which is restricted by the number and locations of static Data Centers (DCs).
- For the data plane, its runtime reconfigurability also dramatically improves flexibility. The last objective is to explore different possibilities to reconfigure a P4 data plane, address the potential state consistency issues between different data plane functions, and evaluate the performance during reconfiguration.

1.1 Research Challenges

Network softwarization is an essential step in the evolution towards the next-generation communication networks that are more flexible. Meanwhile, it is critical to leverage the capability of network softwarization to address the new requirements of the control and data plane. The main goal of this thesis is to model, analyze, and optimize the softwarized control and data plane, which comprises various research challenges. This section summarizes the main challenges which are addressed in the subsequent Chapter 3 to Chapter 6 with detail.

Towards the Flexibility Metric

For the research in communication networks, the flexibility of networking design and solutions is considered as a competitive advantage. However, this advantage is typically only claimed on an argumentative level and not thoroughly investigated. Its interpretation is also not consistent throughout different research works, hindering a fair comparison between different proposals. To address this issue, a rigorous understanding of flexibility should be derived, which unfolds the intrinsic properties, i.e., timely adaptation under small cost, of a flexible networking system. With the interpretation, a formal definition of the flexibility metric using the measure theory can be proposed. As a result, the metric

should provide a whole new dimension to compare different design choices of networks, which should also be supported by realistic use cases.

Modeling and Analyzing Placement Opportunities of the Dynamic Control Plane

Under the flexible distributed control plane architecture, the locations of the controller instances and the assignment of the switches to the controllers can be optimally determined for the sake of the QoS metrics. The optimization analysis is investigated as the Controller Placement Problem (CPP). To reveal the latency overhead introduced by the control plane in the process of reactive flow setup, CPP should focus on the modeling of the underlying latency components, comprising initial and intermediate flow setups when the flow traverses multiple control domains. It should answer questions about how many controllers are needed and how to place controllers and assign switches.

In the face of varying traffic demands, e.g., a varying number of flows between different node pairs over time, the control plane might need to adapt, i.e., change the controller locations and switch assignments. From the flow setup point of view, the adaptation can potentially move controllers to the nodes where most of the flow setup requests emerge, thus reduce the end-to-end flow setup time effectively. However, it may also introduce potential sources of failures and service degradation because of the additional mechanisms. It is critical to consider both aspects as cost components while operating the dynamic control plane over multiple periods towards cost-effectiveness. Because of the optimization problems' NP-hard complexity, heuristics, and even machine learning techniques can be leveraged to design algorithms to improve the solution's efficiency.

Evaluation and Optimization of the Flexibility of the Dynamic Control Plane

The dynamic control plane shows its advantages in adapting to varying traffic demands, which is envisioned as flexible. Meanwhile, comparing different design choices of the control plane, e.g., the number of controllers, has been performed in terms of the standard performance metrics, e.g., control latency and reliability, but not in terms of the flexibility metric. In this regard, a quantification framework needs to be developed and applied accordingly. The framework should be able to accurately calculate the time the system takes for the adaptation process. It should further generate a representative demand set to ensure a fair comparison between different design choices. With the framework, network operators can determine their best choices after analyzing the flexibility metric and the incurred cost.

Evaluating flexibility opens the gate for optimizing flexibility, which has seldom been addressed in the literature. For the SDN controllers, they are software processes that can only execute in the node where a DC exists. Therefore, the planning of DC locations can have a significant impact on the adaptation time of the control plane and, accordingly, the flexibility. The optimization should base upon proper modeling of the adaptation time and cost, and the fulfillment of the time and cost constraints should be revealed in the objective function. Similar to Dynamic Controller Placement Problem (DCPP) problems, heuristic methods might also be needed to yield decision efficiency.

Design of Programmable Data Plane Towards Runtime Reconfigurability

The emergence of programmable data plane technique, e.g., P4, provides a new angle to build network functions and address different networking requirements. P4's platform-independent feature also enables the hybrid deployment of both software and hardware targets. However, a general architecture is needed in order to leverage the flexibility of software targets and the line-rate performance guarantee

of hardware targets without interoperability issues. The architecture should hide the implementation details of different P4 targets, and at the same time, provides abstractions of packet processing resources to ease the programming network functions with different requirements. From the flexibility perspective, the data plane needs to reconfigure its functionality at runtime without service interruption. While field-reconfigurability is another feature of P4, realizing runtime reconfiguration with the limited syntax feature of P4 has been missing in the literature.

Runtime reconfiguration schemes need to consider several aspects. First, additional target-specific features that violate the target-independence of P4 should be avoided to make the scheme more general. Second, states that reside in the P4 targets should be preserved to ensure the consistency of the data plane. For example, migrating a stateful firewall needs to maintain all state variables to avoid allowing malicious traffic and blocking regular traffic. Third, state maintenance should be efficient, and neglect states that are coupled only to a particular target, e.g., a temporary state. This will become more important in the future when more complex P4 programs with millions of state variables emerge. Furthermore, the performance of the data plane in terms of packet processing latency should also be measured and analyzed to understand the cost of reconfiguration and, in turn, flexibility.

1.2 Main Contributions

This section summarizes the contributions of the thesis in the research area of softwarized networks. Specifically, it overviews the research endeavors and introduce their relations. This thesis mainly covers the research in four areas: *(i)* contribution to the interpretation of flexibility based on a literature study and a formal mathematical definition, *(ii)* mathematical models and numerical analysis for the single- and multi-period dynamic controller placement problem in SDN, *(iii)* quantifying and optimizing the flexibility of the SDN dynamic control plane based on the modeling of adaptation time and cost factors, and *(iv)* proposals that leverage PDP technique to enable runtime reconfiguration.

The **first** contribution is based on joint works and relates to the high-level goal of this thesis, i.e., flexible softwarized networks. In order to propose a metric to capture flexibility in a quantitative fashion instead of intuitive arguments, we take the following steps. We first propose a uniform interpretation of a network's flexibility as its timely support of changes in the network requirements with small cost. Afterward, we derive the essential elements of a network, namely topology, flows, node functions, and resources, each of which reveals flexibility in distinct aspects. The flexibility of a network is defined as the relative ratio between the number of supported demand changes and the total number of demand changes we feed into the network. This formal definition is applied for the subsequent research study on specific use cases.

The **second** major contribution targets optimization models for operating the dynamic control plane with the ability itself in the face of dynamic data plane traffic demands. Indeed, the control plane needs to be carefully planned in terms of the number of controllers and their locations, as well as the switches' assignments. A controller can be placed on any node with a DC within a network topology; similarly, a switch can be assigned to any controller. Those features, combined with proper planning, produce significant benefits of the flow setup performance. Correspondingly, we propose single-period DCPP [7] that answers the question of how to structure the control plane based on accurate modeling of end-to-end flow setup time. If a flow traverses multiple control domains, the flow setup procedure will comprise several Round-Trip Time (RTT) between the data plane and the control plane. DCPP belongs to the category of Mixed Integer Programming (MIP) and hence can be optimized by linear

optimizers. A study based on simulations compares different control plane design choices and the benefit of adopting the control plane according to the traffic demands. In the spirit of data-driven networking, we also examine the utility of ML approach that leverages the algorithmic data of previous solutions [9]. The new approach offers fast and accurate placement decisions and can be extended to cover more complex problem instances. In order to further cope with the adaptation aspect of the control plane, we model multi-period DCPP [3], which consists of placement variables of multiple time-slots. To solve the problem efficiently, look-ahead control with simulated annealing splits the original problem into smaller sub-problems and targets them, respectively. Similar to the single-period counterpart, multi-period DCPP is also evaluated numerically with comprehensive simulations.

The **third** contribution targets the flexibility aspect of the dynamic control plane of SDN, which is classified as a general function placement problem. To achieve adaptation, the control plane first needs to update the controller's locations, implemented as controller migration, and the switches' assignment, implemented with reassignment protocol. Both steps need some time to finish, during which the control plane can experience vulnerability and suffer from service degradation, which is considered by the flexibility metric. Hence, we provide a procedure to calculate the overall adaptation time. Following the formal mathematical definition, we also need to define the demand changes as varying data plane traffic represented as flow profiles. The quantification framework with generated demand changes and calculation of adaptation time makes the comparison of different control plane design choices possible and persuasive. Simulation study on different network topologies shows that more controllers do not necessarily result in higher flexibility. Moving one step further, we consider a more realistic assumption: limited DC facilities. It is not possible to have a DC on each node of network topology, especially the large ones, and their locations can have a significant impact on the flexibility of the dynamic control plane, because for example two DCs that are too far away from each other can lead to infeasible adaptation due to the long migration time. Therefore, we should plan ahead and determine the locations of DCs to deliver the highest flexibility, thus optimizing for it. Accordingly, we propose a mathematical model that considers traffic profiles of multiple time-slots and increase the number of profiles that are satisfied under adaptation time and cost constraints. Efficient methods are also introduced to accelerate the decision process without losing much optimality.

The **last** contribution is oriented around PDP with P4. As the most remarkable representative, P4 demonstrates all benefits of PDP, namely target-independence, protocol-independence, and field-reconfigurability. We explore the potentials of P4 in the direction of runtime data plane reconfiguration and decompose them into the following tasks. First, we propose a general architecture called P4NFV that manages Network Functions (NFs) implemented in a heterogeneous P4 data plane, which does not exist in the literature. The architecture hides the technical details of different P4 targets and abstracts each target as a packet processing entity with a particular look-up table and register resources. Second, we present two approaches, i.e., pipeline manipulation and program reload, to enable the reconfiguration without disrupting the original operation of a P4 target. The former leverages the usage of register values to enable and disable a particular section of a P4 program, whereas the latter changes the P4 program entirely, which needs additional support from the target. Proof-of-concept implementation and measurement demonstrate the trade-off between the duration of service degradation and the applicability of the approach, which is a critical factor to consider while realizing a P4 program on a particular target. During data plane reconfiguration, the states, e.g., registers, that reside in the NFs need to be consistently updated. Therefore, in order to facilitate fast reconfiguration, we propose a code analyzer P4STATE to understand the usage of state variables within a P4 program and derive the

ones that need to be migrated. The derivation procedure comprises of multiple steps from binding the state variables to headers, actions, tables, and conditionals, to pruning the Control Flow Graph (CFG), which eliminates stateless operations. A use case study of HULA [184] shows that the analyzer suggests migrating only two types of registers, out of four that are initially specified in the program.

1.3 Thesis Outline

Figure 1.1 illustrates the overview of the thesis and also outlines the structure and maps the main contributions of the thesis to the corresponding chapters.

Chapter 1 introduces the topic of this thesis. It presents the motivation and defines the problem scope and research challenges in managing programmable control and data plane towards flexibility, followed by an overview of the key contributions.

Chapter 2 provides background information on Software-Defined Networking (SDN) and Programmable Data Plane (PDP), as well as their relation to each other. It gives a holistic view of how SDN and PDP contribute to next-generation softwarized networks. Moreover, it describes some features of the data plane programming language Programming Protocol-Independent Packet Processors (P4).

Chapter 3 introduces "flexibility" as a new metric for comparing different network design choices. It presents the necessity of such a metric, as well as its definition. With an example of the controller placement problem, it unfolds how "flexibility" can benefit the decision-making process while designing the control plane of SDN.

Chapter 4 initiates the study of optimization problems that address the dynamic control plane of SDN, i.e., it introduces, models, and analyzes the dynamic controller placement problem. To better model the controller placement problem and reveal the case of reactive flow setup, it presents Mixed Integer Programming (MIP)-based optimization models that consider the end-to-end flow setup latency during the path setup of a new flow. Moreover, it analyses the proposed models for different network topologies and varying traffic inputs.

Chapter 5 is mainly concerned with the detailed application of flexibility measure. To better use the measure, it introduces a general work-flow to compare the flexibility of different control plane designs. It then opens a new research problem of optimization of flexibility. In particular, it proposes a new optimization problem that targets to optimize the flexibility of the dynamic control plane.

Chapter 6 addresses the research gap of how to build an actual flexible data plane described in P4 that can react to external requirements. It starts with an Network Function Virtualization (NFV) architecture proposal to enable runtime reconfiguration of the data plane. With the stateful reconfiguration in mind, it introduces a suite of algorithms that can assist the migration of states in the data plane. In the end, it evaluate the performance of reconfiguration of both software and hardware P4 targets..

Chapter 7 concludes this thesis with the summary and discussion of the critical results. It also provides a broader overview of the impact of this thesis on other researches, as well as the remaining open questions and promising directions for future work.

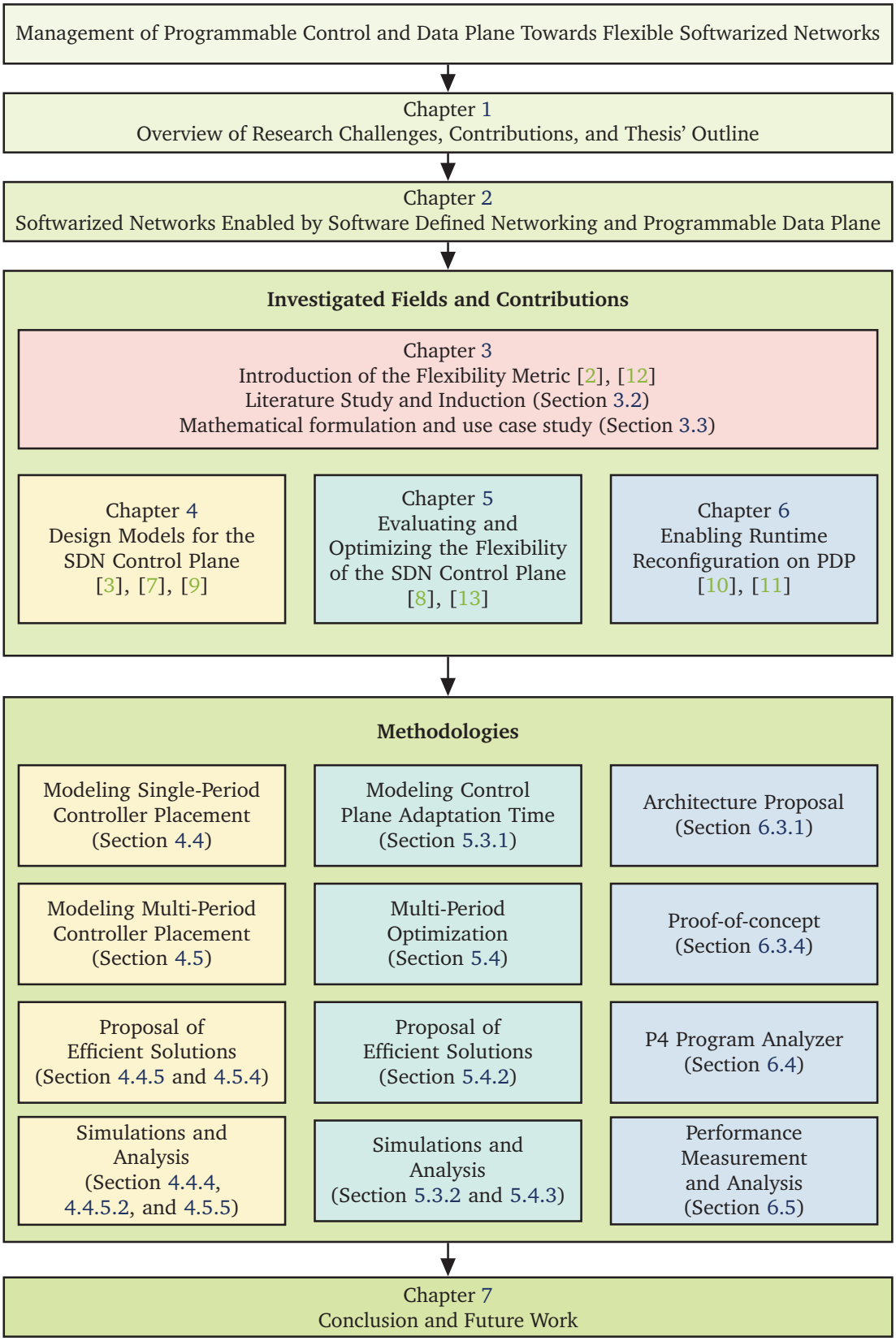


Figure 1.1: Outline of the thesis with the main contributions mapped to the corresponding chapters. Notably, in Section 3, the contribution of literature study and induction is based on the joint work in [2], and the contribution of the mathematical formulation is based on the joint work in [12].

Chapter 2

Background: Network Softwarization with Software Defined Networking and Programmable Data Plane

Software-Defined Networking (SDN) abstracts the decision-making tasks of packet processing and forwarding from distributed networking devices, e.g., switches and routers, and consolidates them within a logically centralized entity, namely SDN controller. This is a significant step forward compared with the legacy networking architecture where networking devices make decisions based on their local knowledge. With SDN, network operators enjoy the ease of management, fine-grained monitoring, and fast convergence time. SDN still relies on a fixed data plane that must process all types of packet headers defined in the specification, e.g., OpenFlow [180], making the hardware implementation potentially bloated but still hard to adopt new features. In this regard, Programmable Data Plane (PDP) is proposed to further abstract the operations networking devices can perform, e.g., parsing and table-matching, and describe them with formally defined syntax so that the same functionality can be executed on multiple heterogeneous devices. PDP enables the specification of networking functionality on-demand. Both endeavors of computer networking contribute towards the concept of *network softwarization*, where it is possible to accurately describe both control and data plane behaviors like writing a piece of software. Meanwhile, how to *flexibly* manage the resources of both planes in the face of dynamic networking condition becomes a critical and challenging research direction. The related research problems will be addressed in the later chapters in this thesis.

This chapter presents the general background related to this thesis, i.e., the concepts of SDN (Section 2.1), PDP (Section 2.2), and their combination towards network softwarization (Section 2.3). The content of this chapter replies partly on the background sections of the publications [3], [7], [10], [11].

2.1 Software Defined Networking

As a new networking paradigm to address the limitations of legacy infrastructures, SDN breaks the virtual integration, decouples the functionality of the control and data plane, and realizes a well-defined programming interface between them [33]. Figure 2.1 illustrates the layered architecture of SDN.

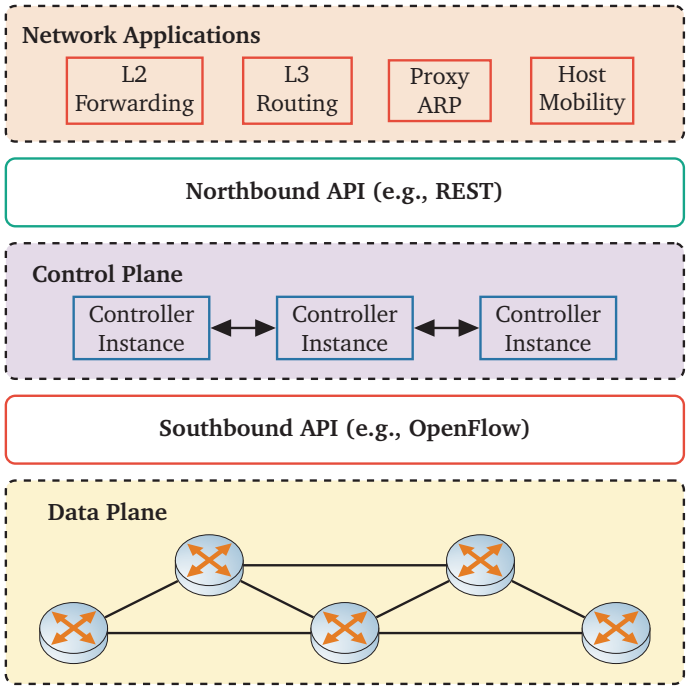
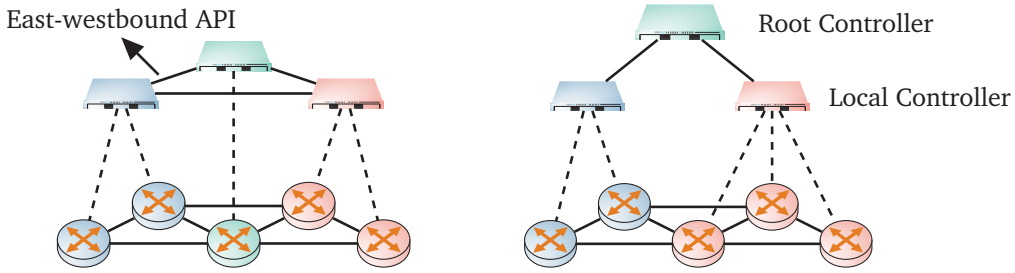


Figure 2.1: Layered architecture of SDN (adapted from [33]): three layers (data plane, control plane and network applications) are connected by south- and northbound Application Programming Interface (API). The data plane layer consists of forwarding devices that do not decide how to forward packets but rely on the indication from the above layer of the control plane that comprises multiple controller instances sharing the same knowledge of the data plane. On the top, the layer of network applications represents various types of functionalities that can be implemented by the data plane.



(a) Flat architecture. (b) Hierarchical architecture.

Figure 2.2: Two types of control plane architectures. The main differences are the connection of the controller instances and the role of each controller instance. In the flat architecture, all controller instances are connected through proprietary east-westbound API and act with the same role. In the hierarchical architecture, only local controller instances are connected to the switches directly.

2.1.1 Control Plane

The SDN control plane has a global view of the data plane, i.e., topology structure, and traffic, i.e., network flows. The control logic is implemented in a logically centralized controller and executed on commodity servers to facilitate the programming of the data plane forwarding and simplify the development of network applications. The forwarding decisions are flow-based: a flow is defined as a set of similar packets that share the same set of packet header fields.

Distributed structure. A logically centralized control plane does not posit a physically centralized architecture. Indeed, the need to guarantee a decent level of performance, scalability, and reliability

would call for such an architecture. As shown in Figure 2.1, controller instances are connected via east-westbound API that is platform-specific. There are two main categories of distributed control plane architectures, which are illustrated in Figure 2.2. For the first category, forwarding devices are partitioned horizontally into multiple areas, with each area controlled by a single controller instance. The physically distributed controllers share the same global network knowledge through synchronization techniques such as the publish/subscribe system and distributed database. This architecture has the advantages of reduced control latency and increased resilience, and is therefore adopted by most controller platforms. The other category employs the hierarchical control structure. The lower layer comprises local controllers and handles local and frequent events that do not depend on the global network state, e.g., processing of Address Resolution Protocol (ARP) requests. The upper layer has a root controller that handles events that require global network knowledge, e.g., load-balancing elephant flows. In this way, the control plane enjoys higher scalability because local controllers do not need to share information among themselves, but at the sacrifice of longer latency of the control messages going to the root controller and lower resilience of the higher layer. The subsequent models in this thesis are based on the *horizontal control plane* because of its wide adoption.

Control Channel. The introduction of *southbound* API removes the dependency of vendor-specific APIs, so that heterogeneous networking devices can be controlled all at once via the same control channel. As one of the first proposals for this API, OpenFlow [180] has become the de facto standard. The control channel can be realized in either *in-band* or *out-band* manner. With in-band control, the control plane shares the same physical infrastructure with the data plane, which saves the deployment of additional infrastructure but hurts from reliability and security issues. Failure of data plane links would impact control plane's connectivity as well, and attackers can tap control packets by leaking from data plane traffic. Out-band control can address the former issues with its dedicated infrastructure but needs a non-trivial control bootstrapping process [48]. Moreover, for Wide Area Network (WAN) that covers multiple countries, additional infrastructure is not at all cost-effective. Because this thesis focuses more on the WAN scenario, the subsequent placement models in this thesis are based on *in-band control*, which is also widely adopted in state-of-the-art research.

Controller Platforms. Since the proposal of SDN, various experimental controller platforms, e.g., Beacon [49] and Floodlight [185], have emerged with basic functionalities to enable proof-of-concept implementations. Meanwhile, to guarantee production-level performance, platforms such as Open Networking Operating System (ONOS) [186] and OpenDaylight (ODL) [187] provide the capability to process high volumes of flows reliably and securely, aided with a multitude of control applications. Both platforms adopt the horizontal architecture. ONOS provides seamless scalability by allowing network operators to incrementally add resources to satisfy the control plane's ever-increasing capacity requirement. Besides, it exposes a global state to network applications so that they do not need to worry about consistency issues. ODL allows the split of the data plane into multiple slices (the idea of virtualization), each slice with its logical controller applications, thus enabling different views of controller given different slices. The subsequent models in this thesis can be applied to both controller platforms.

2.1.2 Data Plane

Data plane in the context of SDN comprises of networking devices with programmable forwarding tables. Similar to the structure of Forwarding Information Base (FIB) in a layer-2 switch, forwarding table

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Figure 2.3: Components of a flow entry in an OpenFlow flow table [180]. Match fields define the packet header components that need to be matched upon. Priority indicates the order of flow entries in which a packet is applied. Counters record the statistics (e.g., the number of packets and the number of bytes) of each entry. Instructions define the set of actions to take upon a successful matching. Timeouts define the allowed valid amount of time during which a flow entry is valid. Cookie is the information that controllers can use to keep track of the entry. Flags indicate the way a flow entry is managed.

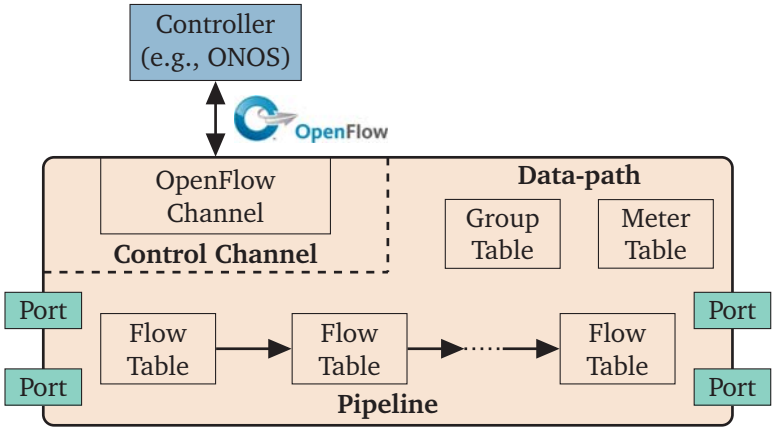


Figure 2.4: Components of an OpenFlow switch. The physical ports (filled with green) are inter-connected by the back plane fabric with the forwarding logic defined by the packet processing pipeline, i.e., the data-path. The pipeline consists of multiple stages of flow tables. The group table allows a common output actions across multiple flow entries. The meter table enables rate-monitoring of traffic prior to egress port. The control channel establishes at least one connection to a remote controller via OpenFlow, which is used to update the table entries at runtime.

defines *match fields*, *instructions*[1], *priority*, *counter* and other ancillary fields for each entry, as illustrated in Figure 2.3. According to the latest specification of OpenFlow [180], match fields are defined to match against packets (also known as a *flow*), consisting of the ingress port and packet headers. Upon matching the fields of an entry, a set of instructions are applied to the packet for processing. Wildcard matching and match fields prioritization facilitate the implementation of more complex networking functions, compared against MAC-based switching and IP-based routing. Counter helps to maintain the statistics of all flows in the data plane and provides insights into the control plane for advanced flow monitoring tasks.

Figure 2.4 demonstrates the main components of an OpenFlow switch. It comprises one or more flow tables and a group table that perform lookups and forwarding and an OpenFlow channel to an external controller [180]. OpenFlow protocol defines the communication pattern between the switch and the controller. With the protocol, the controller can add, update, and delete flow entries in flow tables in reactive (i.e., triggered by *PacketIn* packets) or proactive manner. The former allows the flow tables to be updated in time, whereas the later depends on a preliminary estimation of the traffic to calculate flow entries and does not reflect the timely changes of networking conditions [50]. As this thesis focuses the runtime adaptability of networks (i.e., *flexibility*), it only considers reactive flow entry setup in subsequent models. Matching always starts from the first table and might proceed to the following tables in the pipeline. Group table specifies additional processing that is more complex,

[1]Match fields and instructions are also know as rules and actions in the literature.

e.g., fast reroute and multi-path forwarding, and enables multiple flow entries to forward to a single identifier [180].

2.1.3 Networking Applications

SDN supports a multitude of networking applications with its programming model. These applications run on top of the control plane and connect to it via the northbound API, which offers a common programming abstraction to the upper applications. Hence, SDN offers an increased level of flexibility in terms of network management and control. For example, Li et al. propose a load balancing framework that leverages the controller's global view of the network and optimizes the routing paths to split the workload and increase the reliability of the network [51]. The optimized paths are translated into OpenFlow rules and distributed among the switches. For network security, Fawcett et. al apply a modular and distributed design for efficient monitoring and remediation at scale [52]. Multiple controllers cooperatively offer the multi-level capability for monitoring: light-weight surveillance is implemented by fetching flow statistics from switches, whereas Deep Packet Inspection (DPI) is performed on packets that are sent to the controllers. In cloud scenarios, such as OpenStack [188], SDN can also improve the efficiency and scalability of provisioning virtual networks within a shared infrastructure. The performance of networking applications running in controllers relies heavily on the placement of the controllers and the assignment of switches, which motivates the optimization of both decision variables in Chapter 4.

2.2 Programmable Data Plane

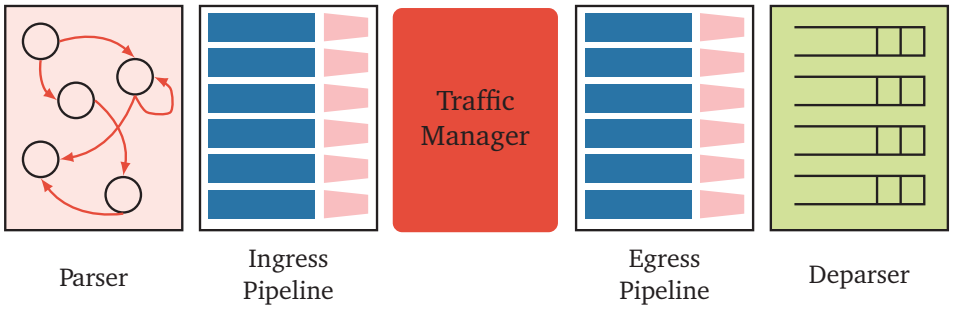
2.2.1 From SDN to PDP

With the help of proposals like OpenFlow, SDN enjoys the benefits of the control plane programmability; however, some limitations and new requirements appear gradually given new networking scenarios. OpenFlow specification becomes bloated from the original version 1.0.0 [189] that supports 12 fields to the latest version 1.5.1 [180] that supports 45 fields but still does not cover all potential fields. The proliferation of OpenFlow also makes it prone to bugs and renders the difficulty of functionality debugging. Furthermore, it is impossible to support custom protocols, as OpenFlow is the protocol that needs to be abode. More advanced data plane statistics other than packet and byte count are also not supported. In these regards, data plane programmability is the only way to go towards full-dimension network softwarization.

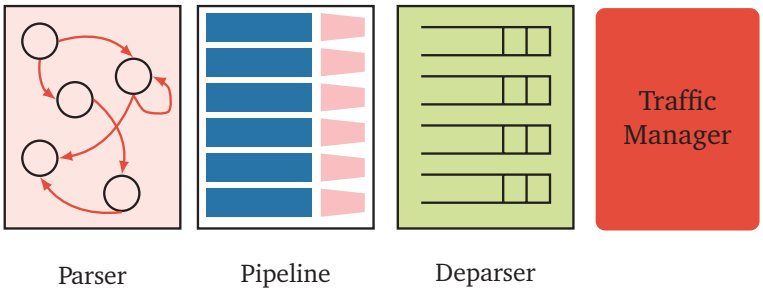
The goals of data plane programmability are as follows. First, the forwarding table supports arbitrary matching and is not limited to the fields that are specified by OpenFlow. Second, the data plane does not rely on any explicit protocols, i.e., protocol-oblivious; therefore, the same forwarding devices can support any existing and future new protocol without any hardware upgrade. Third, the same program[2] is not coupled to any specific hardware target. Therefore it ensures target-independence. Last but not least, line-rate processing should not be sacrificed.

Another technique Network Function Virtualization (NFV) benefits from implementing Network Function (NF), i.e., data plane functions, completely in software and execution in commodity servers

[2]It is unrealistic to assume that all types of hardware targets can execute one data plane program because of the non-trivial difference of their architectures. The point here is the same data plane logic can be applied to multiple hardware targets without much modification, i.e., ease of inter-target transplantation.



(a) Architecture of V1Model (default of BMv2, T4P4S, and Netronome SmartNIC targets).



(b) Architecture of SimpleSUMESwitch (default of NetFPGA-SUME target).

Figure 2.5: Definition of two typical P4 architectures, which specifies a sequence of programmable P4 stages including parser, control flow (represented as match-action units), deparser, and their interfaces. In the parser block, each circle represents a states that the parsing can be. In the pipeline block, the blue rectangles denote the memory spaces reserved for different tables, and the pink trapezoid denote the Arithmetic Logic Units (ALUs) that compose the actions associated with each table.

with Data Plane Development Kit (DPDK) [53]. Whereas this paradigm enjoys the same degree of freedom, it is hard to achieve line-rate for packets smaller than 128 Bytes on X86 Central Processing Units (CPUs) [54]. In order to be protocol-independent, the programming model provides an instruction set suitable to build any arbitrary protocols, and to guarantee performance. Each hardware target has its detail in terms of implementation for each instruction.

While there are other proposals such as Protocol-oblivious Forwarding (PoF) [55], this section mainly elaborates on Programming Protocol-Independent Packet Processors (P4), due to its popularity and promises within both academic and industrial community [34] [3].

2.2.2 P4 Language

The emerging reconfigurable match-action architecture [56] enables the customization of networking devices that were fixed in functionality. Based on this concept, P4 is introduced as a domain-specific language for data plane packet processors and makes it easy to leverage the architecture [57]. Following the common behaviors in packet processing, P4 specifies which header information to parse, how to match on the parsed header, and what actions to take when a matching hit takes place. It adopts an abstraction model for the packet processing pipeline that defines a basic set of constructs [190] as follows.

[3]The main features of P4 and PoF are coincident, but P4 is proposed from the academia following the step of OpenFlow and thus receives more attention

Header type defines the format of each header in a packet as a set of fields and their respective sizes. When a packet arrives, *parser* behaves as an Finite State Machine (FSM), which traverses a predefined sequence, extracts header fields from the packet, stores them into runtime metadata, and forwards the metadata to the processing pipeline. The header type always indicates the first header (e.g., Ethernet header) to extract. Afterward, some of the first header's fields (e.g., Ethernet Type) are examined, which leads to the extraction of the following headers. The FSM ensures that different packets can have different headers extracted and stored into the parsed-headers-stack. Meanwhile, the packet payload, including un-parsed inner-layer headers, is passed directly to *deparser*, waiting to be reassembled with processed packet headers.

Control flow specifies an invoke sequence of match-action units which operates on the extracted headers fields stored in the parsed-headers-stack. Each **match-action unit** does the following three steps: *(i)* builds lookup keys from packet header fields and runtime metadata, *(ii)* performs lookup with the key in the table and choose the associated action and input data, and *(iii)* executes the chosen action. A **table** is a generalized flow table and associates user-defined keys (including header fields and runtime metadata) with the list of possible actions. An *action* is made up of a set of operations that can modify fields of a header, or manipulate the parsed-headers-stack by copying, adding, and removing headers. The metadata in a P4 program includes *user-defined metadata*, which stores any temporary data for packet processing, and *intrinsic metadata*, which provides runtime information about each packet, e.g., ingress queue length and en-queue time.

After traversing the control blocks, the packet reaches the *deparser*, which reassembles the packet through attaching the processed headers back to the payload and pushing it to an egress queue, if not dropping it. *Extern objects* are architecture-specific constructs (e.g., header checksum and encryption units) that are hard-coded and can only be invoked via predefined APIs. The introduction of extern objects extends the applicability of P4 programs by enabling functionalities that cannot be supported natively by P4, such as encryption and hashing. Furthermore, registers, meters, and counters can be declared to maintain states across multiple packets and therefore open the gate to stateful packet processing.

2.2.3 P4 Compilers

As a high-level language, P4 cannot execute on targets without compilation. To enable target-independence and ease of future upgrades, P4 compilers are divided into a common *frontend* compiler and a target-specific *backend* compiler as illustrated in Figure 2.6.

P4 community provides a standard open-source front-end compiler (i.e., P4C [191]) that transforms a P4 program into an Intermediate Representation (IR) in the form of JavaScript Object Notation (JSON) consisting the definitions of parser, tables, and actions. The pipeline is translated into a logical Table Dependency Graph (TDG). P4C provides some basic implementations of optimizations (e.g., dead state elimination and constant propagation) [58].

The back-end compiler, on the other hand, is provided by the vendor who designs the target. It generates a configuration file to program the target's behavior, as well as the control plane APIs to update its state (e.g., table entries and register values) at runtime. Based on the properties of the target, the back-end compiler implements all atomic operations of P4 in a compatible way. Additionally, the back-end compiler may perform optimizations to enhance the packet processing performance of

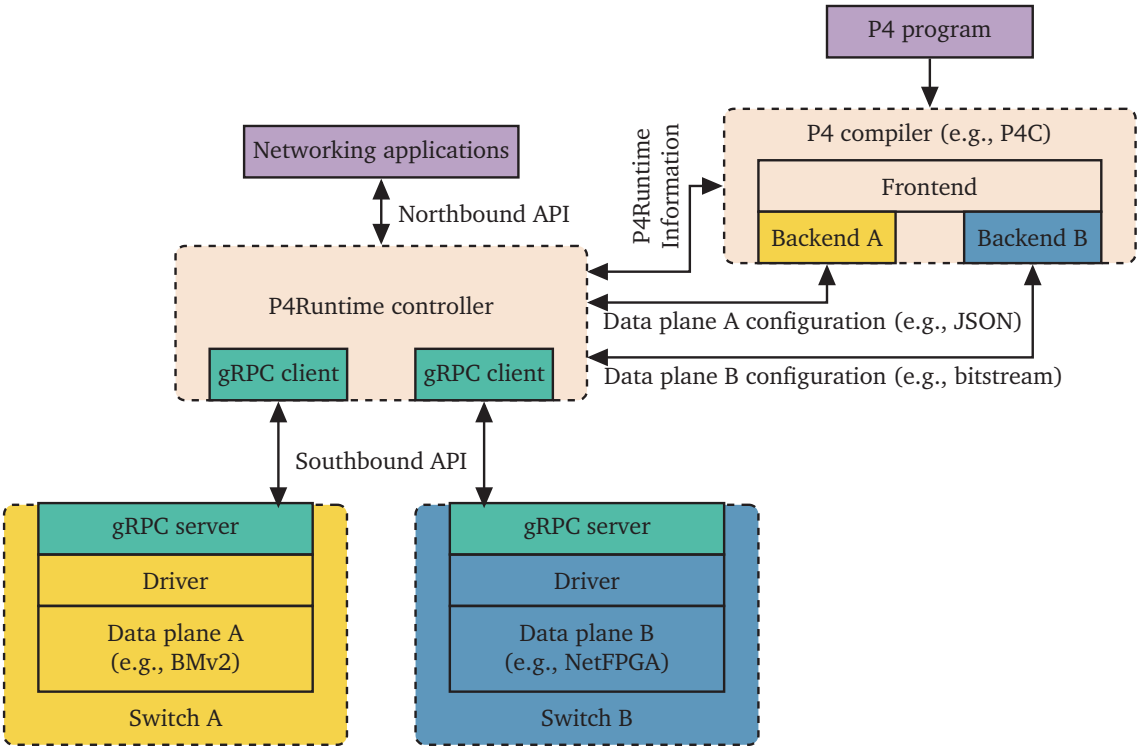


Figure 2.6: Deployment of a hybrid data plane consisting of different switch targets (software switch A and hardware switch B) with P4Runtime as the control plane. The compiler suite has a common front-end and distinct back-end for each target. The P4C compiler is equipped with a general frontend and various target-specific backends. gRPC framework handles the southbound API of the control plane.

the target. In our analysis, we measure the packet forwarding latency of a P4 target considering all the optimizations of the back-end compiler as a part of the tested P4 target, i.e., the P4 target and its back-end compiler define the device under test.

2.2.4 P4 Targets

A P4 *target* is an entity that can execute P4 programs to process data packets. The programming of each target builds on top of a specific architecture. Architecture defines a sequence of programmable P4 stages, including parser, control flow (represented as match-action units), deparser, and the data plane interfaces between them [190]. It serves as a template of a pipeline structure for a P4 target. The interfaces are a set of control registers and signals, represented as *intrinsic metadata* and *user-defined metadata*. Figure 2.5 illustrates the two most common architectures: *V1Model* and *SimpleSumeSwitch*. Whereas V1Model has two control flows, i.e., ingress- and egress pipeline, before and after *traffic manager*, SimpleSumeSwitch only instantiates one control flow and moves traffic manager after deparser. The differences between architectures come from respective intrinsic hardware implementations and need to be considered while operating and transplanting P4 programs across multiple targets.

P4's target-independence feature enables different types of software and hardware networking devices to be programmed. In the following, we provide details about these investigated targets:

BMv2. Behavior Model version 2 (BMv2) [181] is the first software switch prototype and is mainly used for proof-of-concept implementations. Packets are processed sequentially and therefore induces

low resource utilization and limited performance, i.e., around 1 Gbps throughput. Nevertheless, it is assumed to provide the highest flexibility in building up the pipeline and creating extern functions [59].

PISCES. Another software proposal is PISCES [60], which automatically generates customized Open vSwitch (OvS) [61] source code from P4 while optimizing the performance overhead. The generated OvS programs are on average 40 times shorter than equivalent changes to OvS source code and incur a forwarding performance (i.e., throughput) overhead of only about 2% [60].

T4P4S. With the acceleration capability offered by DPDK [192], T4P4S [62] is a framework that generates target-independent C program that is able to execute across multiple platforms. Incoming packets of all flows are distributed to all reserved CPU cores and processed in batches to mitigate CPU cache misses. Data-path connects the interfaces directly to the processing threads, thus avoids unnecessary memory copy.

Netronome SmartNIC. Netronome SmartNIC is a Network Flow Processor (NFP) with tens of multi-threaded purpose-built cores that enable high parallelism. Hierarchical transactional memory and built-in accelerators in the device also boost the processing capability. The back-end compiler generates a C implementation of the data-path and creates the firmware for the SmartNIC. While processing, each packet is distributed to the next available thread for optimum throughput, while the packet order is still maintained [193]. The matching is accelerated with various hardware algorithms [193].

NetFPGA-SUME. Field-Programmable Gate Array (FPGA) is a promising candidate to satisfy the requirements of low-latency, high-throughput concurrent packet processing. NetFPGA-SUME [36] enables easier programmability of packet processors on FPGAs by leveraging the P4→NetFPGA workflow [63]. The back-end compiler generates a Register-Transfer Level (RTL) implementation of the data-path, which is then synthesized to a bitstream to program the FPGA chip. The P4 architecture of NetFPGA is SimpleSumeSwitch. To ensure the minimum latency, NetFPGA implements each type of matches with the most suitable memory resources, e.g., Ternary Content-Addressable Memory (TCAM) for ternary matches, and static Random-Access Memory (RAM) for hash-based exact matches.

Tofino. Tofino [183] is a programmable Application-Specific Integrated Circuit (ASIC) that is designed for networking devices. The chip incorporates a pipelined architecture of reconfigurable match-action tables with arbitrary depth and width [56]. A separate processing unit is provided for each packet header field, so that all may be modified concurrently. Tofino chooses its instruction set carefully to ensure fast execution in heavily pipelined hardware.

The thesis investigates the software target BMv2 and hardware target NetFPGA-SUME, two candidates that are commonly used in state-of-the-art research for prototyping [37], [64]–[66] and performance evaluation [15], [67].

2.2.5 P4 Applications

A simple design and the demand for a high-level human-readable interface have been the main drivers for P4. To represent simplicity, using P4 can reduce the Lines of Code (LoC) of a network function implementation when compared to non-P4 software implementation, thus enabling the fast prototyping in networking research. For instance, the functionality of parsing all Transmission Control Protocol (TCP) flags results in 4 LoC in P4, compared with 370 LoC in the OvS implementation [60].

Table 2.1: Comparison of different proposals of control APIs.

API	Target-independent	Protocol-independent
P4 compiler generated	✓	✗
BMv2 CLI	✗	✓
OpenFlow	✓	✗
P4Runtime	✓	✓

P4 has been used in the design of several network protocols and functions that show great flexibility and performance, such as heavy-hitter detection [68], load balancing [69], [70], and packet scheduler [71].

2.2.6 P4 Control Plane

The match tables, associated with the user-defined actions, define the pipeline of packet processing, and the tables are populated via a control plane interface. Table 2.1 compares four candidates for control plane API. Initially, the P4 compiler generates a set of runtime APIs for the control plane. The disadvantage of this approach, however, is that the APIs' program-dependency hinders provisioning new P4 programs without restarting the control plane. BMv2 CLI provides abundant features of control plane (including table manipulation and configuration reload), yet exclusively for the BMv2 target and not portable. Inherited from SDN, OpenFlow only supports the headers and actions declared in the specification. The limitations of the previous candidate motivate the proposal of *P4Runtime*, a framework for runtime control pf P4 targets that are target- and protocol-independent and enable reconfiguration in the field. Control plane instructions are serialized in a structured format of Protobuf [194], thus breaks the dependency on the program and protocol and gRPC [195] transports them over HTTP/2.0 and TLS, thus breaks the dependency on the target while ensures reliability and security [196]. As shown on the top right side of Figure 2.6, the P4 compiler generates data plane configuration files, as well as a standard P4Runtime Information file that captures attributes of a P4 program including IDs and structures for tables, actions, and their respective runtime parameters. The P4 control plane is needed in the P4NFV management architecture presented in Chapter 6 to manage the multiple P4 devices and backup runtime reconfiguration of functionality.

2.3 Towards Network Softwarization

2.3.1 Difference between OpenFlow and P4

OpenFlow is a representative of SDN, whereas P4 is a representative of PDP. Both focus on opening up the networking devices, but with different approaches. When introducing OpenFlow, the requirement was to have a common API to remotely control several types of forwarding devices. In most networking scenarios, the switches' tasks are similar: exact match of Ethernet Media Access Control (MAC) address, longest prefix matching of IP address, and wildcard match of Admission Control List (ACL). Those tasks can be abstracted in a series of forwarding tables; therefore, the requirement is transformed

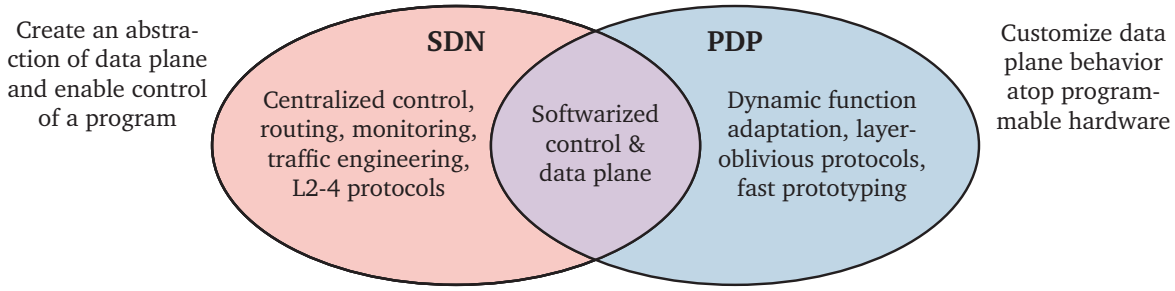


Figure 2.7: Relation between SDN and PDP and their combination towards network softwarization.

into designing a standard and open interface to populate these tables. The interface makes it easier for network operators to write better control planes. OpenFlow assumes the switches have a fixed behavior described in their specifications. While building the control channel, switches can inform controllers about the set of headers they support. OpenFlow does not change switch's behavior per se; it instead provides a way to populate the flow tables with entries. The capability of switches restricts the development of networking applications, i.e., follows the bottom-up paradigm.

P4 turns the paradigm upside down: instead of relying on what a switch can support, P4 tells the switch directly what it needs to do and how it should process packets. With P4, we can define *(i)* what header fields are needed for the subsequent processing, *(ii)* how to match on each header field, *(iii)* what actions to perform on a successful match, and *(iv)* what is the API to populate the switch. In this way, network protocols are merely programs expressed in P4 so that the design of new protocols puts the network programmer (not the switching chip vendor) in charge. With the emerging reconfigurable switching chips that are as fast as the fixed-function chips, the flexibility to support various networking use cases does not come at the sacrifice of performance. While OpenFlow is still useful for networks of programmable switches, P4 makes it possible to orchestrate networks that are composed of heterogeneous data plane devices with customized protocols[4].

2.3.2 Combining SDN and PDP

The above introduction and analysis show several advantages of migrating from the legacy networking composed of devices with fixed functions and distributed protocols towards the emerging SDN and PDP concepts. Figure 2.7 sketches the relation between them: whereas SDN creates an abstraction of the data plane and enables a uniform control over it, PDP programs the behavior of the data plane and achieves reprogrammability. Combining both facilitates the softwarization of both control and data planes, leading to networks that have the ability to adapt in the face of dynamic conditions. Because the combination is promising, the research in this regard starts to flourish. For example, Zhang et al. propose HyperVDP [37] to manage virtualized data plane programs on different P4 targets. The controller of HyperVDP allocates resources and populates tables on the virtual PDP dynamically. Yassen et al. [38] design a fine-grained and precise measurement framework that operates on the scale of an entire network: a set of measurements from each programmable device provide a coherent image of the entire data plane, and the control plane coordinate the measurements and stitch the results.

[4]In fact, P4 can describe OpenFlow. See https://github.com/p4lang/switch/blob/master/p4src/openflow.p4 for an example.

In the meantime, challenges also arise due to the new degrees of freedom offered by these concepts, especially in how to *manage both planes* to accommodate dynamic networking conditions and requirements with reconfiguration. SDN introduces distributed controller instances, whose placement can become a bottleneck to deliver end-to-end flow setup promptly. The *dimensioning and reconfiguration* of the control plane includes resource orchestration across multiple Data Centers (DCs), during which the data plane should be intact, i.e., without a perception of the transaction. PDP directs some of the challenges to hardware vendors, forcing them to rethink the design of *reconfigurable and parallelized* packet processing structures. It is also non-trivial to *reconfigure* a stateful data plane without violation of consistency properties. Fortunately, these challenges are overwhelmed by rewarding opportunities and can be addressed differently according to the design requirements and objectives. Operators can decide on various realistic factors, which can be different between one another. For instance, one could focus on better operating a hybrid data plane with non-programmable hardware switches. Another operator could be interested in provisioning high-performance NFs that are reconfigurable at runtime. It becomes compelling to observe the consequences and dependencies of SDN and PDP together on our networking infrastructure and how they can be devoted to the next-generation network softwarization.

2.4 Summary

In this chapter, we first elaborate on SDN, including the details of the control plane, the OpenFlow protocol, and network applications. Afterwards, we introduce PDP following the road-map from SDN to PDP. In particular, the P4 language and its application details are explained further. At the end, we shed light on the potential of leveraging both SDN and PDP towards network softwarization, which leads to higher flexibility, and it is important to consider the management of both control and data plane.

Accordingly, this thesis fills some of the potential research gaps from both *algorithmic* and *practical* perspectives. Regarding the algorithmic perspective, we propose multiple optimization models to manage the dynamic control plane and evaluate the flexibility, together with other performance metrics. Regarding the practical perspective, we design a data plane management architecture and present a program analyzer to enable efficient data plane reconfiguration, whose performances are evaluated with prototype implementations.

Chapter 3

Flexibility of Softwarized Networks: A New Perspective

3.1 Introduction and Motivation

The techniques of softwarized networks, i.e., SDN, NFV, and Network Virtualization (NV) provide a new level of indirection as well as new interfaces for programming the control plane and setting up physical or virtual network functions and networks on demand. Figure 3.1 is contrived for this chapter with the goal of illustrating the new possibilities enabled by network softwarization [2]. In the figure, the backbone network of a city is deployed with SDN, connecting different facilities such as office and residential buildings (lower left), factories (upper left), and DCs (right). A significant event covering many office buildings demands for high-quality connectivity to a service, which is implemented as Virtual Network Function (VNF) in the upper right DC. To enable the connectivity, an SDN controller configures the flows at runtime and steers traffic towards the service (green line). With NV, the event's traffic can acquire its own Virtual Network (VN) with exclusive virtual resources of the nodes and link, and is isolated from other VNs, such as the one hosting the traffic from factories to the lower right DC. When the upper right DC becomes overloaded, one VNF is migrated to the other DC to keep its Quality of Service (QoS). VNF migration requires both reconfiguration of NFV in the functionality and reconfiguration of SDN in the network. Likewise, link failure in the networking infrastructure triggers the migration of two virtual links (dashed blue line) and maintains the connectivity of factories. In a word, the technologies of softwarized networks support the accommodation of more dynamic changes, therefore rendering higher flexibility.

Flexibility has become a buzzword in networking research and attracted much attention of researchers. To illustrate this trend, a literature study is performed to search publications containing keywords "flexible" or "flexibility" in their titles and abstracts in three major journals of IEEE[1] covering the research field of computer networking from the year 2001 to 2019[2]. As a comparison, we also search publications containing keywords "delay" and "throughput", which are well-known performance indicators of networks. Figure 3.2 illustrates the trend of evolution of the keywords over time, compared to the base number in the year 2001. The usage of "flexibility" escalate sharply from the year 2015.

[1]The selected journals are *IEEE/ACM Transactions on Networking*, *IEEE Journal on Selected Areas in Communications* and *IEEE Transactions on Network and Service Management*.

[2]The original version of this figure is presented in our work [2] summarizing keywords of "flexibility", "bandwidth", and "capacity" in in four major IEEE journals and magazines on wireless communication.

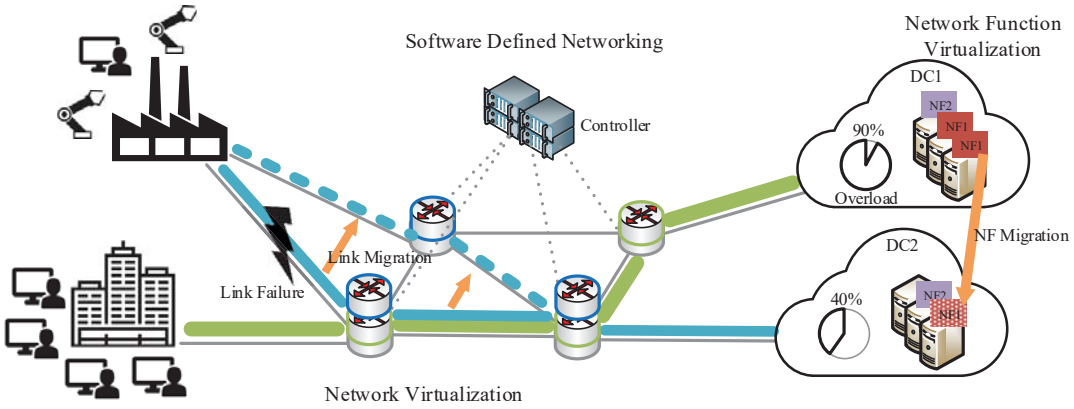


Figure 3.1: An illustration of softwarized network scenario that combines technologies of SDN, NFV and NV (taken from [2]). A central SDN controller directs traffic to particular network services upon demand and sets up the new path as physical link failure occurs. Networks serving various tenants are isolated and controlled dynamically in the same physical infrastructure. NFs are virtualized in different DCs. When one of the DCs is overloaded, some NFs will be be migrated to the other DC to ensure load balancing.

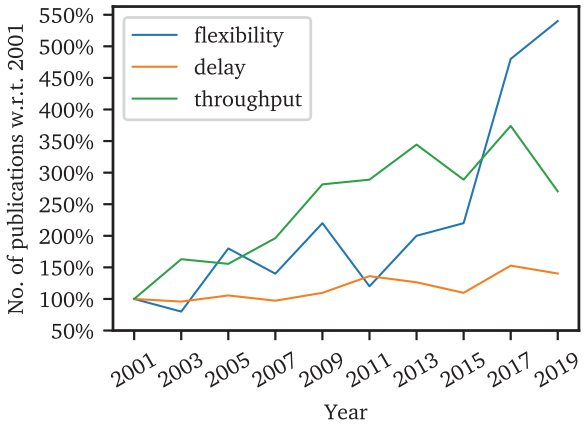


Figure 3.2: Evolution of the number of publications containing the words ”flexible” or ”flexibility” in contrast with those containing ”delay” and ”throughput” in three major IEEE journals on networking, with respect to the number of publications in 2001.

Compared with the year 2001, “flexibility” has increased over 500%, whereas “delay” and “throughput” not more than 150% and 300% respectively.

However, the literature study also observes the lack of a rigorous definition of flexibility and, therefore, different interpretations within various research problems. In DC traffic engineering, flexibility can be considered as the additional throughput performance for increasing traffic, in comparison to a proportional behavior [72]. In NF acceleration, flexibility also means the ability to adopt new policies, algorithms and even functionalities [73], [74]. The intention of this chapter is to propose a formal definition of flexibility based on the most common understanding throughout the literature. The definition serves as the basis of the subsequent chapters on improving the management of programmable control and data planes.

The **contributions** of this chapter include *(i)* the literature survey of research proposals that consider flexibility as one design target [2], which is accomplished together with Alberto Martínez Alba with

equal contribution, *(ii)* the mathematical formulation of flexibility [12], which is a joint work together with Markus Klügel and Péter Babarczi, and *(iii)* the SDN dynamic controller placement use case example [1].

This chapter is structured as follows. We first elaborate on network flexibility from a uniform interpretation in Section 3.2. The formal definition of the flexibility measure is proposed in Section 3.3. Diving into the definition details, Section 3.4 and Section 3.5 introduce the demands and constraints, respectively. Section 3.6 compares the design and operation phases during the life-cycle of a flexible network. Finally, Section 3.7 concludes the chapter with an example of dynamic controller placement explaining all the terms related to network flexibility.

3.2 What is Flexibility?

The section introduces a uniform interpretation of flexibility and the categories and aspects of a network that exhibits flexibility. The content (same for Section 3.4 and 3.5) is based on the joint work [1], [2], [12].

3.2.1 Uniform Interpretation

Flexibility evaluation is specific to a particular context, which in our case is the communication networks. We can decompose a computer network into four components, namely *topology*, *flows*, node *functions* and *resources*, and each component has its own state. *Topology* is represented as a set of nodes and edges. *Flows* are defined as respective source and destination node pairs with data rates. Node *functions* describe specific actions that are applied to network flows, such as routing and firewalling. *Resources* abstract all needed to ensure a network's operation, including CPU, memory, and bandwidth. Table 3.1 shows the different subsets of components and respective supported flexibility of SDN, NFV and NV, due to their conceptual distinctions. SDN controls flows' forwarding paths and allocates bandwidth on each link. NFV features placing and chaining various NFs and provisioning resources to them. NV enables multiple virtual networks with distinct topologies in the same physical substrate.

We abstract as many notions from the literature as possible while proposing network flexibility's true meaning. We interpret it as the *timely support of changes in the network requirements with small cost*. The *changes in the requirements* can also be referred as *demand changes* in short[3]. We expect a network to support the demand changes in two ways. It can either accommodate the changes directly without any internal adaptation or adapt the state of its four components to appeal to the demand changes. Both *time* and *cost* is involved in our interpretation. When adaptation takes place, it should finish before a deadline (i.e., time constraint). Violation of time constraints leads to adaptation failure. Besides, the cost involved in adapting to the state should be small enough.

3.2.2 Flexibility Categories and Aspects

The concept of *demand changes* should also be clarified while analyzing flexibility. We consider a *demand change* as a *change in the network requirements that may imply its state's modification*, i.e., updating the states of the network's components. Because of various types of adaptation, we classify them based on the general operations that a network can apply. *(i)* A network can *adapt its configuration*, which

[3]Note that in [2], the definition of the "requests" is the same as that of the "demand changes" here.

Table 3.1: Comparison of SDN, NFV and NV technical focus areas w.r.t. flexibility support [8].

	Flows	Functions	Resources	Topology
SDN	Routing, traffic engineering, flow monitoring	–	Bandwidth, control plane adaptation	–
NFV	–	Provision & placement, service function chaining, monitoring	Adaptation of computation, memory, I/O	–
NV	–	Virtual network implementation	Virtual node capacity, link bandwidth embedding	Virtual network topology adaptation, multi-tenancy, isolation

Table 3.2: Technical Concepts and their support of flexibility in networks [8]. (✓: main target)

Category	Aspect	SDN	NFV	NV
Adapt configuration	Flow Configuration: flow steering	✓	-	-
	Function Configuration: function programming	-	✓	-
	Parameter Configuration: change function parameters	-	✓	✓
Locate functions	Function Placement: distribution, placement, chaining	-	✓	✓
Scale	Resource and Function Scaling: processing and storage capacity, number of functions	✓	✓	✓
	Topology Adaptation: (virtual) network adaptation	-	-	✓

consists of steering flow(s), setting parameter(s) and function(s). *(ii) Locating functions* of a network can meet new latency and service requirements. *(iii)* Scaling a network's components, such as adding or removing functions, links, or resources, can ensure its performance without too much over-provisioning. We refer to each group as a category and use Table 3.2 to show the categories.

While examining the details of each category, we notice that particular action is always associated with the specific technology of softwarized networks. For instance, *flow configuration* is coupled with SDN, whereas *topology adaptation* is enabled by NV. We refer to the action as *aspect*, which is vital to dissect the flexibility of softwarized network designs. All available aspects are summarized in Table 3.2. For details of these aspects, readers are encouraged to read our survey of flexibility [2].

3.3 The Measure of Flexibility

The section presents the measure of flexibility with a formal notation, and the content is based on the joint publication [12].

3.3.1 System Model

As mentioned before, the main features of interest for network flexibility are threefold: a variety of adaptation possibilities, speed of adaptation, and overhead of adaptation. Therefore, without claiming its completeness, we have the following formal definition [12]:

Definition 1 *Network flexibility is the ratio of the number of demand changes that can be satisfied (adapted) over the total number of demand changes, under predefined time and cost constraint.*

We consider a network that can be described by a *system state* $\mathbf{S} \in \mathcal{S}$, where $\mathcal{S}$ contains all possible states of the components (introduced in Section 3.2.1) that the network can realize. For example, $\mathcal{S}$ of the SDN dynamic control plane reflects the number and locations of all active controllers, as well as the assignment of all switches from the data plane. Furthermore, we define a *demand set* Ω that captures demands posed to the network. Demands represent requirements that trigger the adaptation of network state $\mathcal{S}$. In particular, each demand $d_i \in \Omega$ is associated with a set of valid states $\mathcal{V}(d_i) \subseteq \mathcal{S}$ that can satisfy that demand. The network will adapt its state to satisfy each demand, which changes over time. A *demand change* is an event denoted by the tuple of initial demand d_i and new demand d_j, i.e., $d_{i,j} = (d_i, d_j)$ for $d_i, d_j \in \Omega$. Each change $d_{i,j}$ requires the system to adapt from $\mathbf{S}_i \mapsto \mathbf{S}_j$, where $\mathbf{S}_i \in \mathcal{V}(d_i)$, $\mathbf{S}_j \in \mathcal{V}(d_j)$, respectively.

We define *system implementation* $X \in \mathcal{X}$, where $\mathcal{X}$ is the set of possible implementations, which is associated with specific algorithms, protocols, hardware, and software modules. All system states that can be realized by an implementation X is denoted as $\mathcal{S}_X \subseteq \mathcal{S}$. Consequently, the set of valid states that can be achieved by X given a new demand $d_i \in \Omega$ is represented by $\mathcal{V}_X(d_i) = \mathcal{S}_X \cap \mathcal{V}(d_i)$.

3.3.2 Formal Definition

With the previous system model, we define the set of achievable demand changes by the considered system implementation X under given adaptation time constraint T and cost constraint C as:

$$\begin{aligned} \mathcal{A}_X(T, C) \quad = \quad & \{d_{i,j} \in \Omega \times \Omega : i \neq j; \mathcal{V}_X(d_i), \mathcal{V}_X(d_j) \neq \emptyset; \\ & \tau_X(d_{i,j}) \leq T; c_X(d_{i,j}) \leq C\}\,. \end{aligned} \tag{3.1}$$

The first part inside the set definition ensures that $d_{i,j}$ is valid and can be realized by implementation X. Afterward, we evaluate the flexibility of a network system implementation as the size of the set $\mathcal{A}_X(T, C)$.

Definition 2 *Given a system implementation X with the set of achievable demand changes $\mathcal{A}_X(T, C)$ concerning time and cost constraints T and C. The flexibility of X is defined as $\mu(\mathcal{A}_X(T, C))$, where μ is an appropriate measure on $\Omega \times \Omega$.*

The operator $\mu(\cdot)$ can be any measure that satisfies the following properties of mathematical measures [75].

Property 1 *An implementation X that cannot react to any demand change $d_{i,j}$ has zero flexibility (i.e., is inflexible), as $\mathcal{A}_X(T, C) = \emptyset$ and hence $\mu(\mathcal{A}_X(T, C)) = 0$.*

Following Property 1, we know that the definition of "inflexibility" is strict in a sense that the system only supports one demand.

Property 2 *An implementation X is more flexible than its alternative Y if it can react to more demand changes under time and cost constraints T, C, indicated by $\mu(\mathcal{A}_Y(T,C)) \leq \mu(\mathcal{A}_X(T,C))$.*

Property 2 enables the comparison of flexibility and is consistent with what is commonly used in literature of softwarized networks. That is, the ability to configure a network's flows, functions or topology to support more demands makes itself more flexible.

Property 3 *If implementation X and Y can realize different demand changes, an implementation $Z = X \cup Y$ can be constructed, that selects among X and Y the one that can realize a given demand change within the constraints, with ties broken arbitrarily. It holds $\forall T, C$ that $\mu(\mathcal{A}_Z(T,C)) \geq \max\{\mu(\mathcal{A}_X(T,C)), \mu(\mathcal{A}_Y(T,C))\}$.*

With the last property, we can design a more flexible network by combining different techniques of network softwarization. Note that if an adaptation incurs delay and cost, there are time or cost thresholds under which the adaptation can fail, and in the end, the network is not as flexible as we originally expect. In the literature, we observe that typically such delay and cost are not negligible, and the corresponding thresholds, in reality, are quite low.

Different measure types result in different flexibility values and even might lead to different relative orderings of network systems. For the sake of simplicity and without loss of generality, we define flexibility $\varphi(\mathcal{A}(T,C))$ with the help of a set of predefined demand set Λ:

$$\varphi(\mathcal{A}(T,C)) = \frac{\mu(\mathcal{A}(T,C))}{\mu(\Lambda)} = \frac{\text{\# achievable demand changes by } X}{\text{\# defined demand changes}}, \tag{3.2}$$

where $\mathcal{V}_X(d_i)$ and $\mathcal{V}_X(d_j)$ are defined by Λ. Apparently, $\varphi(\mathcal{A}) \in [0,1]$. In this case, the flexibility of 1 corresponds to a network that can satisfy all demand changes. A detailed discussion about other candidate measures can be found in [12].

In the following chapters (Chapter 4-6), we will clarify the representations of all notations in each discussed use case.

3.4 Demand Changes

The definition of demand changes is coupled with the use case; its definition depends on the functionality of the system and what factors can trigger the system's adaptation. In softwarized networks, we observe the following types of demand changes.

- Traffic variation: It requires a network to reconfigure flows, scale resources, or update functions.
- Network lease: Network needs to address varying requirements coming from tenants, e.g., VM size, interconnection bandwidth, or virtual topology.
- Network upgrade: Operators of a network can be interested in enhancing network performance or upgrading new protocols.
- Failures mitigation: The ubiquity of networking asks for fast connectivity and service recovery upon various types of failures.

One type of demand change can be accommodated by exploiting various aspects. For instance, to reduce control latency experienced by switches, a network can either move controller (as a function) to a location closer to the switches or reconfigure the paths of control traffic. Thus both function placement and flow configuration are valid solutions.

3.5 Time and Cost Constraints

The adaptation time constraint reflects how fast the system can adapt. We argue that a system which takes too long to react to new demand changes would not be deemed as flexible. From the literature, we observe that the choice of adaptation constraints depends on technology, network domain, and use case, which falls into the range from milliseconds to minutes. Re-placing network functions of the mobile core network can take up to one hour, because of the underlying optimization problem [76]. The adaptation time of redirecting flows' paths is in the magnitude of seconds, considering the flow rule installation time [77] and the forwarding latency between switches and controllers [78].

The cost of flexibility reflects the induced cost of adaptation, which relates to the other mechanisms to enable the adaption, the overhead to implement the adaptation, and even a function of the system's performance after or before the adaptation [12]. For example, more DCs need to be built to support more possibilities in terms of function migration. Moreover, the cost associated with adaptation itself also plays a role and is not negligible. Despite its importance, we observe that the literature has overlooked it. For instance, SDN enables reconfiguration of flow paths at runtime, and load balancing proposals adapt flows' paths according to the traffic demand change. If the reconfiguration takes place without proper coordination, the consistency properties, e.g., loop-free and blackhole-free [79], can be easily violated and as a result, it induces high cost because of high delay or packet drops. Another example is leveraging FPGA and P4 for hardware NF acceleration, which benefits from its programmability [74]. However, it is not trivial to update the running algorithm or functionality at runtime without stopping the operation of NF. Techniques such as hardware partial reconfiguration [80] and temporary backup [81] are proposed to address the issue. In short, the cost factor indeed opens new research directions.

As an example, the study in Chapter 5 considers the use case of the dynamic control plane with the following time and cost constraints. The adaptation of the dynamic control plane is meant to deliver optimal flow setup performance (measured as the average flow setup time) and includes the procedure of controller migration and switch re-assignment [3], whose latency should be considered. To avoid delay of data plane packets, the adaptation should finish within tens of milliseconds [82]. The definition of cost constraint reflects the flow setup performance before the adaptation. Facing fast-changing traffic, the flow setup would potentially take longer without adaptation and thus incur high cost.

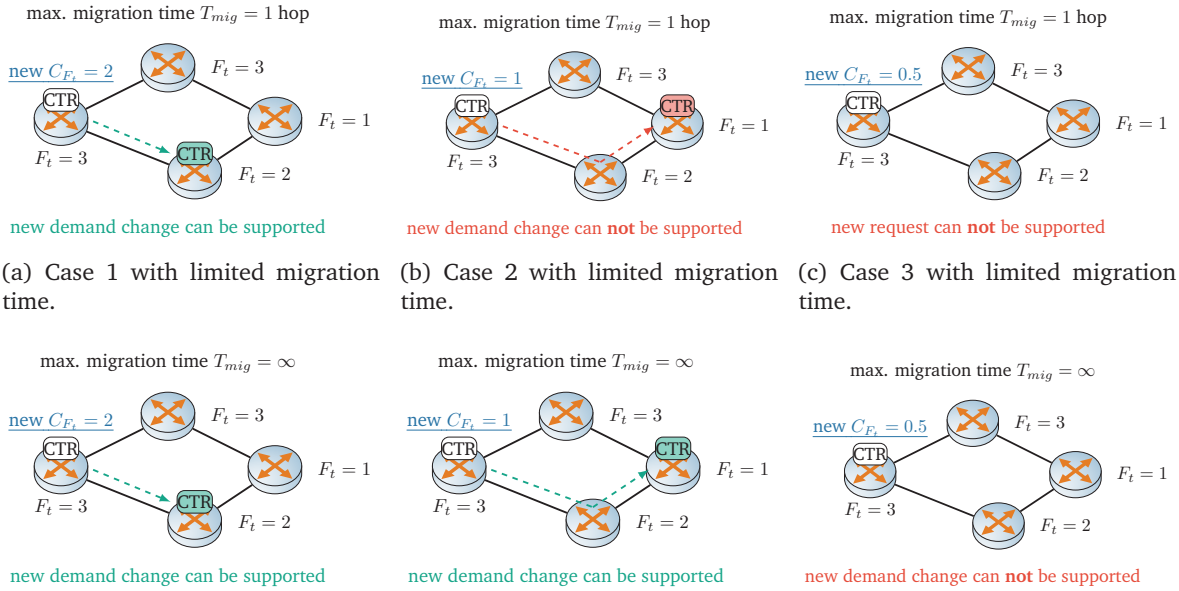
3.6 Design and Operation Phase

Two phases are involved in managing a flexible networking system during its life-cycle: design and operation phase. In the design phase, we decide the parameters of the system, which are hard to change once the system starts to operate. In the operation phase, we optimize the system-specific performance metric at runtime, but under the constraints of system design and operation. In our use case, the design parameters include the number of DCs and their locations. The constraints of the operation phase cover the parameters of DCs, the relation between controllers and DCs, the properties of the control plane,

and the time of the control plane adaptation. The possible performance metrics are flow setup time, control plane reliability, and overhead.

For the use case of the dynamic control plane, we address the problems of performance optimization mainly in Chapter 4, and answer the question of "how to optimize its flexibility" mainly in Chapter 5.

3.7 An Example of Dynamic Controller Placement



(a) Case 1 with limited migration time.

(b) Case 2 with limited migration time.

(c) Case 3 with limited migration time.

(d) Case 1 with **unlimited** migration time.

(e) Case 2 with **unlimited** migration time.

(f) Case 3 with **unlimited** migration time.

Figure 3.3: Example of the flexibility measure where the network creates demand changes in terms of flow setup requests that need to be satisfied within a given time threshold T_{mig}. CTR indicates an SDN controller. F_t indicates the incurred cost in terms of average flow setup time if controller is to be placed on the node. C_{F_t} indicates the required flow setup time. This figure is adapted from the version published in [1].

To better demystify the components of flexibility, Figure 3.3 illustrates an example of an SDN network with one controller (denoted as "CTR" in the figure). The introduction of the controller incurs additional control latency while setting up new flows. For time-sensitive flows, their average flow setup time F_t should be suppressed under a certain threshold C_{F_t}. Because the dynamicity of flows induces shifts of control traffic, the control plane needs to adapt in order to satisfy the threshold accordingly. In this case, the *demand* belongs to category of *traffic variation*. The control plane adaptation is realized as the migration of the controller across multiple nodes. Since SDN controller is an NF running in commodity servers, the flexibility *aspect* related to this example is *function placement*.

Initially, most flows originate at the switch on the left side of the topology, making that switch the optimal location for the controller. A new *demand change* stems from traffic variation requires the controller to migrate to a location with F_t smaller or equal to 2; otherwise, the *cost constraint* C_{F_t} would be violated. Suppose migrating a controller over 1 hop link takes 1 time unit. When a maximum migration time threshold (i.e., *time constraint*) T_{mig} of 1 unit is allowed, the controller would be able to migrate, and therefore, the control plane is flexible enough to support the demand change (shown in Figure 3.3a).

In case the new demand change asks for a location that can reduce the average flow setup time to 1 and the maximum migration time threshold T_{mig} is 1, the control plane would fail to support this demand change, which means the cost and time constraints cannot be fulfilled at the same time. This is illustrated in Figure 3.3b. Even if the network has a location for the controller to ensure low average flow setup, i.e., $F_t = 1$, the system is not flexible enough to support the controller migration within 1 time unit. In case we tolerate unlimited migration time, i.e., $T_{\text{mig}} = \infty$ (shown in Figure 3.3e), this network design would be able to support the new controller migration request. Note that if the request asks for an average flow setup time limit of 0.5 (shown in Figure 3.3c and Figure 3.3f), the control plane would not be able to support it regardless of the migration time threshold.

Following the previous mathematical notation, the demands d_i of this use case are the controller placement requests under cost constraint of an average flow setup time target and time constraint of migration latency. The valid system state $\mathcal{V}(d_i)$ for demand d_i are the possible locations of the single controller. We refer to each switch's location from the top clockwise as v_1, v_2, v_3, and v_4 respectively. In Figure 3.3a the given implementation X realizes the controller location on v_3.

With the given set of new demand changes $\mu(\Lambda)$ defined as r_1, r_2, r_3, satisfying r_1 means $\mu(\mathcal{A}(T, C)) = 1$ under migration time threshold T_{mig} of 1 unit. The flexibility of the control plane is $\varphi(\mathcal{A}(T, C)) = 1/3 = 0.33$, which are shown in Figure 3.3a, 3.3b, and 3.3c. Without a deadline of migration time, the flexibility would be $\varphi(\mathcal{A}(T, C)) = 2/3 = 0.66$, which are shown in Figure 3.3d, 3.3e, and 3.3f. Note that for the sake of clarity, we only consider a small number of demand changes in this example. In reality, we might need a broader set of demand changes to make a much more meaningful comparison.

Section 5.3 will extend this use case by considering a more realistic way to measure the latency of controller migration, as well as switch reassignment.

3.8 Summary

In this chapter, we analyze the benefits of various state-of-the-art techniques of softwarized networks, which greatly enhances network's flexibility. Meanwhile, we realize the lack of a common understanding of flexibility, which hinders a fair comparison among various research proposals and therefore concerted efforts towards a network with higher flexibility. We also notice that adaptation time and cost, despite their importance on flexibility, are not always considered. Accordingly, this chapter offers a uniform interpretation of flexibility. The interpretation brings in a structural comparison of SDN,NFV, and NV with respect to flexibility.

Further, we derive basic elements of evaluating flexibility of a network and provide a mathematical basis to quantify the degree of flexibility achieved by a network. We propose to define flexibility as the number of achievable demand changes by a network design over the total number of demand changes. With a use case of the dynamic control plane, we show the application of the measure and its merit. This use case initiates the discussion of the dynamic control plane in SDN; the next chapter will focus on how to design a flexible dynamic control plane considering the average end-to-end flow setup time.

Chapter 4

Design Models for Dynamic SDN Control Plane

4.1 Introduction

4.1.1 Motivation, Problem Scope and Research Challenges

SDN changes the paradigm of network configuration and management by splitting control and data plane. With a global view of the data plane, the logically centralized control plane makes forwarding decisions and manage link and node resources more efficiently. As the application of SDN in WAN, Software Defined Wide Area Network (SD-WAN) faces the challenge to cover very large geographical area with high forwarding delays[1], and the underlying traffic requires both high processing capacity [84] and reliability [83]. To address these, SD-WAN deploys a control plane with multiple physically distributed controllers, each controller managing a subset of forwarding devices (i.e., switches or routers) [85]. The controllers synchronize periodically to maintain a consistent global view of the network. Compared to the legacy control plane running Multiprotocol Label Switching (MPLS) or Open Shortest Path First (OSPF), the new control plane design is more flexible in terms of resource provisioning. First, more controller instances or resources can be allocated to alleviate potential increase of controllers' response time, in the face of temporal increase of the control plane traffic. Second, forwarding devices with high data plane traffic can be reassigned to different controllers to ensure load balancing. In all, it introduces a new degree of freedom while operating the flexible control plane.

The management and operation of the control plane are particularly critical, given that a dynamic placement of controllers and assignment of switches[2] according to the real-time data plane traffic is supposed to provide lower cost in terms of flow setup time and controller synchronization. Nevertheless, the state-of-the-art has overlooked a comprehensive study on the optimization of the flexible control plane. The goal of this chapter is to close the research gap by systematically modeling the incurred cost and providing realistic optimization models that outputs optimal control plane deployment decisions. Further, considering the computation efforts of the proposed models, efficient solutions are proposed to save the algorithms' execution time without losing much optimality. The efficient solutions cover from legacy meta-heuristics to new machine learning algorithms, allowing us to take advantage of both the optimization theory and the spirit of data-driven networking.

[1] As an example, Google's B4 [83] worldwide SD-WAN deployment in the year 2018 consisting of 33 sites connects east Asia, north America, and north Europe, which can incur more than 100 milliseconds end-to-end forwarding latency.

[2] For the sake of simplicity, we will use "switch" to represent "forwarding devices" in the remainder of this thesis.

4.1.2 Key Contributions

The contributions presented in this chapter are based on the work in [7], [9], and [3], and can be summarized as the following:

1. Exact models of the dynamic control plane are provided [3], [7]. First, end-to-end flow setup is modeled considering the latency of flow setup requests and controller processing. Second, control plane reconfiguration is modeled considering controller migration and switch reassignment.
2. In-depth mathematical formulation of Dynamic Controller Placement Problem (DCPP) considering both single- and multi-period is presented with respect to operational and reconfiguration cost [3], [7]. DCPP decides the best locations of controllers and switch assignments according to each new data plane traffic distribution and demonstrates the advantage of the dynamic control plane.
3. Efficient algorithms are proposed to address the optimization problems [3], [9]. Various approaches are applied. In the spirit of data-driven networking, the Machine Learning (ML) approach offers fast and accurate solution by predicting the locations of controllers, given data of previous solutions from similar problem instances. Look-ahead control approach with meta-heuristic splits the intractable multi-period optimization into smaller problem instances and target them accordingly, which can significantly save the optimization time.

The rest of the chapter is organized as following. In Section 4.2, the related work of Controller Placement Problem (CPP) and DCPP is clarified. Section 4.3 introduces the background of this chapter. Section 4.4 presents the single-period DCPP, and Section 4.5 extends the problem into multiple periods. Finally, Section 4.6 summarizes this chapter.

4.2 Related Work

CPP is initiated by Heller et al. [46] and formulated as a general facility location problem [86], i.e., deciding the optimal number and locations of controllers, as well as the assignment of switches to the controllers, towards a certain objective. By taking dynamic traffic into account, CPP is extended to DCPP, i.e., the control plane adapts to current new data and/or control plane traffic distributions. This section summarizes the most representative works of CPP and DCPP and classifies them according to different criteria, namely *modeling formulations*, *objectives*, and *methodologies*. A comprehensive survey of the state-of-the-art research in this regard can be found in [87], [88].

4.2.1 Modeling Formulations

While the basic system model defined across the literature is similar, design choices in finer modeling formulations are diverse and depend on the particular context under consideration. Those design choices are also motivated by the objectives introduced later.

- **Latency in network**. As the most common design choice, latency is considered in all literature, e.g., [46], [89]–[93]. Various types of latency components are studied: transmission, forwarding, and controller processing.
- **Controller synchronization**. The distributed control plane needs to ensure a consistent view of the data plane with inter-controller synchronization. Inter-domain flow routing might also need communication between switches [90], [91], [93], [94].

- **Controller capacity**. To ensure control plane's QoS, each controller can only handle data plane requests arriving under a certain rate. This capacity is modeled as the number of switches a controller can handle [89], [91], [93], [95], [96], or in a more realistic sense, the processing latency of a controller [94] based on queuing theory [97].
- **Network failures and post-failure scenarios**. While considering the resilience and reliability perspective of CPP [90], [93], [95], [98], the literature models the failures of controllers or links. After a failure, the affected switches are taken over by the backup controller, or the control paths are rerouted.
- **Dynamic traffic**. For the extension of DCPP, traffic is considered either in the form of the request rate of switches [94], [99], or the amount of consolidated flow setup requests at a controller from data plane flows [91].

4.2.2 Objectives

- **Control latency.** In different contexts, control latency can refer to either the latency between switch and controller [46], [90], [96], or the latency between the moment that switch sends a control plane packet (e.g., Packet-In) and the moment that it receives answer from controller (e.g., Flow-Mod) [92], [100]. In this thesis, we refer to the former definition. Besides, forwarding latency can be ignored in DC networks [94].
- **Controller load balancing**. It refers to the degree that reflects how the requests from data plane are evenly distributed among all controllers [46], [89], [90].
- **Resilience and reliability**. It can be formulated as the number of switches that are prone to node and link failures [98], or the reliability of the backup path [93].
- **Cost-efficiency and multi-objective**. The single cost objective can be represented as the monetary cost in terms of the control plane deployment and interconnection [101]. Compound cost factors considering a trade-off among various objectives also lead to multi-objective optimization [94], [100].

4.2.3 Methodologies

As a general facility location problem, CPP belongs to combinatorial optimization and is NP-hard [46], [102]. Accordingly, the methodologies applied in this regard are classified into two categories based on whether they can derive optimal solutions. **Brute force** enumerates all possible placement variables for the best objective and thus suffers from intensive computations [46], [101]. **Mathematical programming** also targets optimality by efficiently steering in solution space, e.g., Integer Linear Programming (ILP) [95], Mixed Integer Programming (MIP) [93], and Quadratic Programming (QP) [91]. The sub-optimum algorithms are diverse. The most representative type is **greedy** [91]. Algorithms based on **graph theory** also flourish: k-means [92], k-center [96], and spectral clustering [103]. Another distinct branch is **meta-heuristics**, including Simulated Annealing (SA) [90], [91], evolutionary algorithms like particle swarm optimization and genetic algorithm.

4.3 Background

This section introduces background on algorithms that are applied later to DCPP optimization problems.

4.3.1 Heuristics

Greedy Algorithm. Because of its straightforwardness and efficiency, greedy algorithm[3] is often used to solve optimization problems to minimize (or maximize) an objective function value subject to a set of constraints. Greedy algorithm makes a sequence of optimization decisions towards the construction of a valid solution, each decision being the best at that time it was made [104]. When making the sequence of decisions, greedy algorithm never goes back to earlier old decisions. However, there is no guarantee that a greedy algorithm can produce the optimal result for a problem.

Local Search. Local Search (LS) searches in the space of candidate solutions. Starting from a candidate solution that is normally generated in a random fashion, LS moves to a neighbor candidate solution that is better than the current candidate solution in a sense that the objective function value is superior. The search continues until all neighbor candidates are inferior to the current solution [105]. LS is easy to implement and can find good solutions shortly. Nevertheless, local optima that better than its neighbors but still not the global optimum can trap the search process. Various techniques are introduced to help get rid of local optima, such as Guided LS, Tabu Search, and SA.

Simulated Annealing. Analogous to the process of physical annealing with solids, SA is capable of escaping local optima by performing hill-climbing moves, which means the next candidate solution can have a worse objective function value [105]. In detail, SA generates and evaluates a new neighbor solution in each iteration. When better than the current solution, the neighbor solution becomes the current solution; when worse than it, there still exists a possibility to accept the neighbor solution. The possibility is proportional to the gap of objective function value between two solutions, and is non-increasing with each iteration [105]. From the application point of view, both the choice of cooling scheme (i.e., initial temperature and temperature multiplication factor) and the choice of neighborhood play a vital role in the performance of SA.

4.3.2 Look-ahead Control

Classical *offline optimization* problem assumes complete information as input, i.e, there is no uncertainty about any problem instance. In contrast, *online optimization* problem only has access to partial information when making a decision, and as the problem progresses, new information as input becomes available for further decision making. Look-ahead control is a type of online optimization algorithms that makes sequential decisions under incomplete information input where each decision is made based on limited but still a preview (i.e., look-ahead window) of future input data [106]. Formally, we use $\Pi_P(S, \omega)$ to denote an online optimization problem P with a series of input state S, where each problem instance p_t takes place at each time-slot t with current state s_t and look-ahead window size ω. In other words, the look-ahead window indicates that the future input states $s_{t+1}, s_{t+2}, ..., s_{t+\omega}$ are available of each p_t.

Receding Horizon Control (RHC) [107] solves $\Pi_P(S, \omega)$ at each time-slot t over the look-ahead window $(t, t+\omega)$; each optimization takes place given the system state of the last time-slot $t-1$. Fixed Horizon Control (FHC) [94], on the other hand, keeps all the decision variables of time-slot $t, .., t+\omega$ and directly jumps over the current look-ahead window ω. Thereafter, only the decision variables of the first time-slot t are applied and the remaining decision variables, i.e., of $t+1, ..., t+\omega$, are discarded.

[3] We refer to "greedy algorithm" as a set of algorithms that share a common algorithmic pattern, irrespective of the problems that they target.

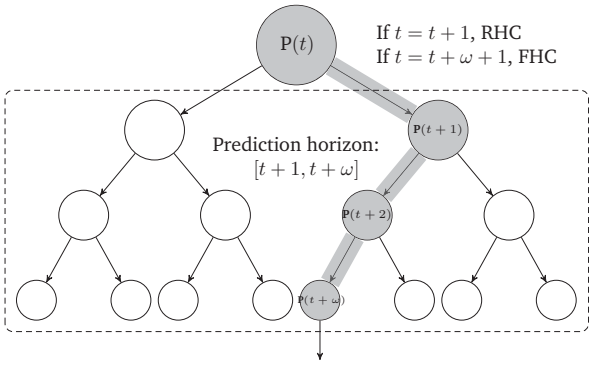


Figure 4.1: Illustration of different look-ahead control schemes to solve online optimization problems (adapted from [108]). The trajectory is explored by a limited look-ahead window size ω. The shaded area represents the solution of an online algorithm, based on the predicted future problem input, i.e., traffic profiles.

Figure 4.1 illustrates the differences between the two alternatives. FHC guarantees shorter running time by solving fewer optimization problems ($\lceil |T|/(\omega + 1) \rceil$) compared to RHC ($|T|$).

4.3.3 Machine Learning

As an application of Artificial Intelligence (AI), ML assists systems with the ability to automatically learn implicit patterns from data set without explicit and sophisticated modeling of the system [109]. ML has been applied in different research fields to make better decisions and predictions, including network optimization that involves many challenging mathematical problems.

We mainly discuss the application of *supervised learning* in this thesis. The other two major categories of ML are unsupervised learning and reinforcement learning. For supervised learning, an algorithm first experiences *training* phase, and finds a mapping between the input data and its ground-truth values. Afterwards, during the *prediction* phase, the same algorithm with the trained mapping takes new input data and produces the predicted values. A typical ML algorithm can address either *regression* or *classification* tasks.

Regression. If the prediction is about continuous quantity given a feature vector (i.e., input variables), we call it regression. An algorithm approximates a mapping function from the input variables to output variable(s).

Classification. The task of classification is to predict a label. Different to regression, it maps a feature vector to *discrete* output variables, which are also called *labels* or *categories*. The number of output labels decides which type of classification it is.

If an algorithm only outputs the prediction of one label at each time, it does *single-label classification*. For instance, when recognizing numbers from handwriting text, the label represents a single digit from 0 to 9. Another example is to classify the weather status, where the single label represents, e.g., sunny, cloudy, or rainy. *Multi-label classification* [110], however, outputs the prediction of multiple labels. To solve the DCPP problem, we categorize the selection of each controller as a separate label; therefore, the problem belongs to multi-label classification. Table 4.1 states the interpretation of the symbols used later. Next, the ML algorithms that can be applied for multi-label classification will be introduced.

Table 4.1: Mathematical notations for ML algorithms.

Notation	Description
$\mathbf{x}^{(i)}$	$\mathbf{x}^{(i)} \in \mathbb{R}^n$, i^{th} feature vector, in our work $n = \mid \mathcal{N} \mid$
$\mathcal{I}$	$\mathcal{I} := \{l_1, l_2, \ldots, l_{\mid\mathcal{N}\mid}\}$, set of possible labels
$\widetilde{\mathcal{I}}$	$\widetilde{\mathcal{I}} \subseteq \mathcal{I}$, one specific labeling
$\widetilde{\mathcal{I}}^{(i)}$	$\widetilde{\mathcal{I}}^{(i)} \subseteq \mathcal{I}$, set of labels associated with vector $\mathbf{x}^{(i)}$
$\mathbf{z}^{(i)}$	$\mathbf{z}^{(i)} \in \{0,1\}^{\mid\mathcal{I}\mid}$, binary indicator vector associated with $\mathbf{x}^{(i)}$. Component $\mathbf{z}_j^{(i)}$ is one, if and only if label $l_j \in \widetilde{\mathcal{I}}^{(i)}$. Vector $\mathbf{z}^{(i)}$ represents the "true" placement.
$h(\cdot)$	$h : \mathbb{R}^n \to \{0,1\}^{\mid\mathcal{I}\mid}$, hypothesis of a classifier.
$\mathbf{y}^{(i)}$	$\mathbf{y}^{(i)} \in \{0,1\}^{\mid\mathcal{I}\mid}$, prediction of a classifier, i.e., $\mathbf{y}^{(i)} = h\left(\mathbf{x}^{(i)}\right)$.

4.3.3.1 Machine Learning Algorithms

Decision Tree. As its name indicates, a Decision Tree (DT) has a structure of a tree consisting of nodes, branches and leafs. Each node represents a test on one entry within the feature vector, each branch represents the result of a single test, and each leaf represents the predicted label. A prediction is made after traversing all entries. To avoid the bias of training only one DT, an ensemble learning approach named *Random Forest* takes the predictions of a number of DTs that are trained separately and summarizes a single output [109].

Logistic Regression. Logistic Regression (LR) is used to map the probability of a certain label given an input vector [109]. Entries of input vector are combined linearly with respective weights to produce a value, which is then transformed by a logistic function to produce the probability value between zero and one.

Neural Network. The basic of Neural Network (NN) is perceptron (i.e., neuron), which takes several binary inputs and produces a single binary output [111]. A NN connects a number of layers of neurons in a certain way, with the first layer known as the *input layer* that takes the normalized feature vector, and the last layer known as the *output layer* that produces output. Each neuron carries its own *weight* that facilitates the prediction; the *backpropagation* algorithm trains all the weight in the network [111].

4.3.3.2 Machine Learning Metrics

For the evaluation of regression algorithms, we can apply *Meas Square Error* (MSE), and of classification algorithms, we can apply *Accuracy* and *Hamming Loss* (HL).

Mean Square Error. Mean Square Error (MSE) is one of the most preferred metrics for regression, which averages the squared difference between the ground truth value and the predicted value. Formally, it is defined as follows:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{N}(y - y(i))^2. \tag{4.1}$$

Accuracy. If a binary classification algorithm makes a prediction, each outcome can be classified into one of four cases: True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (TN).

Accuracy calculates the ratio of how many predictions are correct:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}. \tag{4.2}$$

Note that for imbalanced data sets, i.e., unequal distribution of classes, accuracy can be a misleading metric. Other similar metrics include *precision* and *recall*.

Hamming Loss. As a commonly used metric for multi-label classification, Hamming Loss evaluates the fraction of the wrong labels to the total number of labels [110]. Following the notation in Table 4.1, we define it with the following:

$$\text{HL}(\mathbf{z}, \mathbf{y}) := \frac{1}{N} \sum_{j=1}^{|\mathcal{I}|} \text{xor}\left(\mathbf{z}_j, \mathbf{y}_i\right), \tag{4.3}$$

where N is the number of samples.

4.4 Single-Period Dynamic Controller Placement Problem

This section first introduces problem settings and models end-to-end flow setup time. Then, it provides the mathematical definition of single-period DCPP and discusses evaluation settings and results analysis in detail. A data-driven approach with ML is also proposed to leverage algorithmic data and benefit from runtime saving.

4.4.1 Network Model: SDN with distributed control plane

We consider an SDN network with topology following an un-directed and connected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each node[4] $\mathcal{V}$ in the graph hosts an SDN switch which is assigned to one master controller. The set of all switches controlled by one controller is called a *control domain*. Φ represents the set of controllers that compose the distributed control plane. The controllers need to be placed and the potential nodes that could host a controller are represented by $\mathcal{C} \subseteq \mathcal{V}$. Each controller only installs flow rules to its control domain switches reactively after it receives a flow setup request. We assume in-band control, i.e., the control plane share the same networking infrastructure with the data plane. Shortest-path algorithm calculates a set of paths $\mathcal{P}$ between all node-pairs to determine control/data plane routing. Each path between node v and u is represented by (v, u) with forwarding latency $\ell(v, u)$, and is composed of an ordered set of node pairs $\Omega(v, u)$.

For single-period DCPP, we assume the scenario in which no controller is overloaded and the processing time is negligible. This assumption is deemed for the purpose of simplification but still reasonable. Indeed, measurement [112] confirmed that the processing time for each flow setup request varies between 100 and 150 microseconds for a light-loaded controller, whereas the forwarding latency on each link in SD-WAN can be in a magnitude of millisecond. Nevertheless, this factor will be considered in a more comprehensive model later in multi-period DCPP.

4.4.2 Modeling End-to-End Flow Setup time

We formally define end-to-end flow setup time L_{avg} as follows. Consider an inter-domain flow in Figure 4.2, which originates at H_1 and ends at H_2. When the first data packet of this flow reaches switch S_{11}, the switch has no stored table rule with respect to this new flow and thus needs to send an

[4]We use "node" and "switch" interchangeably in this chapter.

Table 4.2: Notation of sets and parameters of DCPP.

Notation	Description
Network	
$\mathcal{G}(\mathcal{V}, \mathcal{E})$	Graph of SDN network
$\mathcal{V}$	Set of SDN switches (network nodes), i.e., node locations
v	Physical SDN switch (network node) $v \in \mathcal{V}$
$\mathcal{E}$	Set of physical network links
$\mathcal{C}$	Set of potential controller locations where $\mathcal{C} \subseteq \mathcal{V}$
$\mathcal{P}$	Set of pre-calculated shortest-paths between all switch pairs
(v, u)	Shortest path between two switches v and u, with $(v, u) \in \mathcal{P}$
$\ell(v, u)$	Latency of the shortest-path $(v, u) \in \mathcal{P}$ with $\ell(v, u) \rightarrow [0, +\infty)$
$\Omega(v, u)$	Ordered set of switch pairs along the shortest path (v, u) from v to u
Controllers	
Φ	Set of controllers' IDs
θ_ϕ	Processing capacity of the controller with ID $\phi \in \Phi$
κ_ϕ	Reserve factor of the controller with ID $\phi \in \Phi$
K	Number of active controllers with $K = \lvert\Phi\rvert$
Traffic	
$\mathcal{F}$	Set of flows (flow profile) at time-slot t
f	Single flow with $f \in \mathcal{F}$
s^f, d^f	source and destination of flow
r^f	rate to trigger new flow setup of flow f
T	Set of time-slots

initial flow setup request [91] to its master controller C_1. After flow rules are properly calculated (by an algorithm, e.g., shortest-path), the controller sends them to every involved switch inside its own control domain, in this case S_{11} and S_{12}. The data packet is then forwarded inside the first domain until it enters the second domain. Similarly, the switch S_{21} initiates an *intermediate flow setup request* [91] to its master controller C_2, waits for the flow rule setup and forwards it further. T_f is the difference between the time S_{11} receives the first data packet and S_{22} successfully forwards it to the destination host H_2, i.e., the time needed to completely setup the forwarding path of a flow. Next we will provide a formulation of L_{avg}.

For simplicity, we consider L'_{avg} which is the time for the packet to go through one control domain. Figure. 4.3 illustrates two possible scenarios with S_{11} being the ingress switch and S_{12} being the egress switch. T_1 represents the forwarding latency between S_{11} and S_{12}. T_2 and T_3 represent the control latency of S_{11} and S_{12} respectively, regardless of how many switches reside in the control path. While comparing the magnitude of $T_1 + T_2$ and T_3, theoretically we would encounter the following three scenarios.

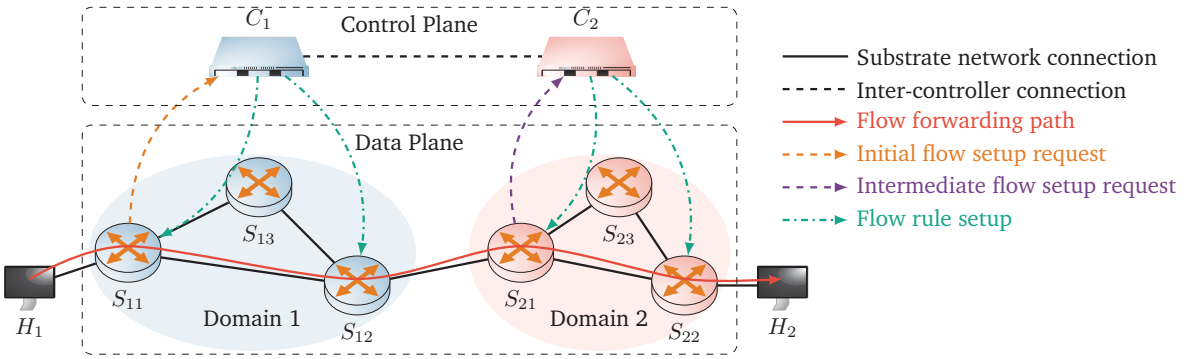
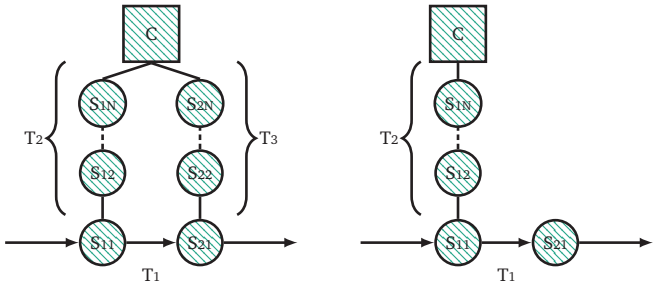


Figure 4.2: Illustration of the flow setup procedure of an inter-domain flow. Two controllers compose the distributed control plane, and each one controls three switches. A flow initiates at H_1 with the destination of H_2, which traverses two control domains.



(a) $T_1 + T_2 > T_3$. (b) $T_1 + T_2 = T_3$.

Figure 4.3: 4.3a and 4.3b cover two possible topologies for the depiction of the flow setup time. A flow enters the control domain of C_1 at S_{11} and leaves from S_{12}. T_1 is the latency between S_{11} and S_{12}. T_2 and T_3 represent control latencies of S_{11} and S_{12}. There are two disjoint control paths in (a) just for illustration and switches in two control paths could actually overlap. The number of switches residing in each control path is greater than or equal to zero.

1. $T_1 + T_2 > T_3$: Shown in Figure 4.3a, $T'_f = T_2 + max(T_2 + T_1, T_3) = T_1 + 2T_2$. The data packet will be directly forwarded when it arrives at S_{12}, since the flow rule setup packet reaches S_{12} earlier than the data packet.

2. $T_1 + T_2 = T_3$: Shown in Figure 4.3b, the control path of S_{12} has to go through S_{11}, resulting in both flow rule setup packet and data packet simultaneously reach S_{12} and therefore $T'_f = T_2 + T_2 + T_1 = T_1 + 2T_2$.

3. $T_1 + T_2 < T_3$: This scenario would not happen in reality, because the control path of S_{12} is not the shortest to the controller C_1, which violates the assumption of our network model.

Therefore, we could conclude that L'_{avg} is a summation of twice the control latency of the ingress switch S_{11} and the forwarding latency of the flow inside this domain. As we consider all control domains that a flow goes through, L_{avg} thus consists of every involved flow setup request and reply, as well as a total forwarding delay from source to destination.

4.4.3 Problem Formulation

This section formulates a model that evaluates different design choices of the control plane, where the aforementioned end-to-end flow setup time modeling is taken into account. We optimize the “average

Table 4.3: Variables of single-period DCPP.

Notation	Description
$p_{\Phi,\mathcal{C}}(\phi, c)$	binary variable representing if the controller with ID $\phi \in \Phi$ is placed at node $c \in \mathcal{C}$
$a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c)$	binary variable representing if the switch $v \in \mathcal{V}$ is assigned to controller with ID $\phi \in \Phi$ placed at node $c \in \mathcal{C}$
$l_{\mathcal{V}}(v)$	non-negative variable representing the control path latency of a switch $v \in \mathcal{V}$
$d_{\mathcal{V},\Phi}(v, u, \phi)$	binary variable representing if both switches $v \in \mathcal{V}$ and $u \in \mathcal{V}$ are assigned to the same controller with ID $\phi \in \Phi$
$\bar{d}_{\mathcal{V}}(v, u)$	binary variable representing if both switches $v \in \mathcal{V}$ and $u \in \mathcal{V}$ are assigned to different controllers
$\tau_{\mathcal{V}}(v, u)$	non-negative variable representing the necessary control forwarding latency if the flow goes from $v \in \mathcal{V}$ to $u \in \mathcal{V}$

flow setup time". In reactive mode, SDN introduces additional flow setup time for every new flow. Slow flow setup can potentially lead to Service Level Agreement (SLA) violations, as network service can run only after the successful setup of the relevant flows.

4.4.3.1 Problem Input and Variables

Table 4.2 summarizes the input sets and parameters of network, controller and traffic. On top of the ones that are introduced in Section 4.4.1, flow profile $\mathcal{F}$ represents the set of flows currently in the network, indicating one flow distribution. Each element $f \in F$ is defined as a source-destination node pair and its forwarding path is represented as an ordered set of node pair p_f from source to destination. Table 4.3 summarizes all the variables of DCPP. Two binary decision variables are decided for each $\mathcal{F}$: the locations of the controllers $p_{\Phi,\mathcal{C}}(\phi, c)$ and the assignment of switches to controllers $a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c)$. The others serve as helping variables to build the objective function.

4.4.3.2 Objective Function

We target the average of all incurred end-to-end flow setup time for current new flows, i.e., flow profile, in the data plane. For each new flow, its setup time sums up the latency of the initial and intermediate flow setup requests and the corresponding flow rule setups, as well as the forwarding latency along the path. The problem's objective is to determine the locations of controllers and the assignment of switches to controllers, which minimize the average flow setup time:

$$\text{minimize} \quad \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \Big[2 \cdot l_{\mathcal{V}}(s^f) + 2 \cdot \sum_{(v,u) \in \Omega(s^f, d^f)} \delta_{\mathcal{V}}(v, u) + \ell(s^f, d^f) \Big], \tag{4.4}$$

where the set $\mathcal{F}$ as a flow profile includes all new flows that each needs the control plane to setup its path. In the square bracket, the three entries denote the latency of initial flow setup, intermediate flow setup, and flow forwarding. Note that the first two entries have multiplication factor of 2, meaning the involved setup request and flow installation.

4.4.3.3 Constraints

This section introduces the constraints for single-period DCPP.

Number of controllers. We ensure that the number of placed controllers (i.e., the number of selected controller nodes) is equal to K:

$$\sum_{\phi \in \Phi} \sum_{c \in \mathcal{C}} p_{\Phi,\mathcal{C}}(\phi, c) = K. \tag{4.5}$$

Assignment of switches. Each switch $v \in \mathcal{V}$ must be assigned to its master controller $\phi \in \Phi$, which means that there is only one control path per switch[5]. Besides, each controller can only be placed on one of the potential controller locations defined by $\mathcal{C}$:

$$\sum_{\phi \in \Phi} \sum_{c \in \mathcal{C}} a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c) = 1, \quad \forall v \in \mathcal{V}. \tag{4.6}$$

Assignment to active controllers. Each switch v can only to assigned to an active controller with ID ϕ that has been placed on a certain node c:

$$\sum_{v \in \mathcal{V}} a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c) \leq |\mathcal{V}| \cdot p_{\Phi,\mathcal{C}}(\phi, c), \quad \forall \phi \in \Phi, \forall c \in \mathcal{C}. \tag{4.7}$$

Location sharing. A switch v should be assigned to a controller with ID ϕ, if the controller shares the same location c with the switch. Note that this constraint can be omitted, when a switch is allowed to be assigned to any controller in the control plane. Thus,

$$p_{\Phi,\mathcal{C}}(\phi, c) = a_{\mathcal{V},\Phi,\mathcal{C}}(c, \phi, c), \quad \forall \phi \in \Phi, \forall c \in \mathcal{C}. \tag{4.8}$$

Control latency. The control path latency of a switch v, i.e., $l_{\mathcal{V}}(v)$, equals the shortest path latency between a switch v and the respective controller instance ϕ on node c:

$$\sum_{\phi \in \Phi} \sum_{c \in \mathcal{C}} a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c) \cdot \ell(v, c) = l_{\mathcal{V}}(v), \quad \forall v \in \mathcal{V}. \tag{4.9}$$

Switches in one control domain. For every two switches v and u, the following constraint ensures that they are in the same control domain if they are assigned to the same controller:

$$\sum_{c \in \mathcal{C}} a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c) \cdot a_{\mathcal{V},\Phi,\mathcal{C}}(u, \phi, c) = d_{\mathcal{V},\Phi}(v, u, \phi), \quad \forall v, u \in \mathcal{V}, \phi \in \Phi. \tag{4.10}$$

Switches in different control domains. The following constraint checks all controller instances from Φ with node v and u. If v and u are assigned to the same controller instance ϕ, the variable $\overline{d}_{\mathcal{V}}(v, u)$ is 0; otherwise, 1. Thus,

$$1 - \sum_{\phi \in \Phi} d_{\mathcal{V},\Phi}(v, u, \phi) = \overline{d}_{\mathcal{V}}(v, u), \quad \forall v, u \in \mathcal{V}. \tag{4.11}$$

Control latency of each hop. Not all hops along the flow's forwarding path trigger flow setup. Therefore, we iterate through every consecutive switch pair along a flow forwarding path $\Omega(s^f, d^f)$. If

[5]Notably, our model can be easily extended to consider SDN switch with multiple controllers to cover the scenario of the reliable control plane.

one switch pair, e.g., v and u, leads to $\overline{d}_{\mathcal{V}}(v,u) = 1$ (meaning the flow enters a new control domain), node u will initiate a flow setup request with control latency $l_{\mathcal{V}}(u)$, which is denoted as $\tau_{\mathcal{V}}(v,u)$. If $\overline{d}_{\mathcal{V}}(v,u) = 0$, the control latency $\tau_{\mathcal{V}}(v,u)$ will also be 0, because two nodes are in the same control domain. Thus,

$$l_{\mathcal{V}}(u) \cdot \overline{d}_{\mathcal{V}}(v,u) = \tau_{\mathcal{V}}(v,u), \quad \forall f \in \mathcal{F}, (v,u) \in \Omega(s^f, d^f). \tag{4.12}$$

4.4.3.4 Linearization

Two constraints, i.e., Eq. (4.10) and Eq. (4.12), are composed of multiplication of linear variables. We apply linearization techniques by introducing auxiliary constraints to them before we can use linear optimization problem solver, e.g., Gurobi [197] and CPLEX [198]. Eq. (4.10), which has a product of two binary variables, is replaced by:

$$d_{\mathcal{V},\Phi}(v,u,\phi) \leq a_{\mathcal{V},\Phi,\mathcal{C}}(v,\phi,c), \quad \forall v,u \in \mathcal{V}, \phi \in \Phi, \forall c \in \mathcal{C}; \tag{4.13a}$$

$$d_{\mathcal{V},\Phi}(v,u,\phi) \leq a_{\mathcal{V},\Phi,\mathcal{C}}(u,\phi,c), \quad \forall v,u \in \mathcal{V}, \phi \in \Phi, \forall c \in \mathcal{C}; \tag{4.13b}$$

$$d_{\mathcal{V},\Phi}(v,u,\phi) \leq a_{\mathcal{V},\Phi,\mathcal{C}}(v,\phi,c) + a_{\mathcal{V},\Phi,\mathcal{C}}(u,\phi,c) - 1, \quad \forall v,u \in \mathcal{V}, \phi \in \Phi, \forall c \in \mathcal{C}. \tag{4.13c}$$

Eq. (4.12), which has a product of one binary and one non-binary variable, is replaced by the following four constraints:

$$\tau_{\mathcal{V}}(v,u) \leq \overline{d}_{\mathcal{V}}(v,u) \cdot \overline{l_{\mathcal{V}}}, \quad \forall f \in \mathcal{F}, (v,u) \in \Omega(s^f, d^f); \tag{4.14a}$$

$$\tau_{\mathcal{V}}(v,u) \leq l_{\mathcal{V}}(u), \quad \forall f \in \mathcal{F}, (v,u) \in \Omega(s^f, d^f); \tag{4.14b}$$

$$\tau_{\mathcal{V}}(v,u) \geq l_{\mathcal{V}}(u) + \overline{d}_{\mathcal{V}}(v,u) - 1, \quad \forall f \in \mathcal{F}, (v,u) \in \Omega(s^f, d^f); \tag{4.14c}$$

$$\tau_{\mathcal{V}}(v,u) \geq 0, \quad \forall f \in \mathcal{F}, (v,u) \in \Omega(s^f, d^f). \tag{4.14d}$$

In order to avoid an empty solution space, we define a constant $\overline{l_{\mathcal{V}}}$ as the upper bound of control latency of all switches $l_{\mathcal{V}}(v), \forall v \in \mathcal{V}$. For its value, we use the worst-case end-to-end forwarding latency in the topology.

4.4.3.5 Baseline Models

The previous model (referred to as the CTR-SW model) allows the adaptation of both controller locations (variable $p_{\Phi,\mathcal{C}}(\phi,c)$) and switch assignment (variable $a_{\mathcal{V},\Phi,\mathcal{C}}(v,\phi,c)$) in accordance to traffic patterns. To demonstrate its advantage, we provide three other models.

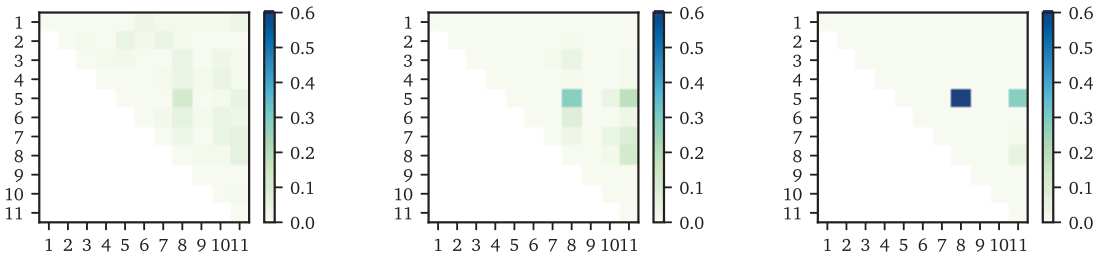
CTR model. The control domain remains intact; only controller is allowed to change its location inside the control domain. Spectral clustering [113] is applied to decide the components of each control domain.

SW model. Switches can be reassigned to different controllers whose locations are intact and decided by optimization of average control latency [46]. This model is considered mainly in the literature of control plane traffic load balancing [114], [115].

STATIC model. As a naive model, it represents a completely static control plane: the placement is optimized with the CTR-SW model for random traffic.

Table 4.4: Evaluation settings of single-period DCPP.

Parameters	Values
Flow density $\rho_{\mathcal{F}}$	0.05, 0.3, 0.6, 0.9
Traffic intensity	Log-normal distribution with mean=1, var=0.8 [Gbps]
Data rate per flow	50 Mbps
Topology	Abilene (11 nodes), AttMpls (24 nodes)
Runs per model	100
Evaluated models	CTR-SW, CTR, SW, STATIC



(a) $\rho_{\mathcal{F}} = 0.05$. (b) $\rho_{\mathcal{F}} = 0.3$. (c) $\rho_{\mathcal{F}} = 0.9$.

Figure 4.4: Controller placement distribution for Abilene network topology of the single-period DCPP model. Each square denotes one combination of two controller locations, i.e., $K = 2$. The brightness shows how often the combination is considered as optimum among all simulation runs. One figure per flow density $\rho_{\mathcal{F}}$.

4.4.4 Results Evaluation and Analysis

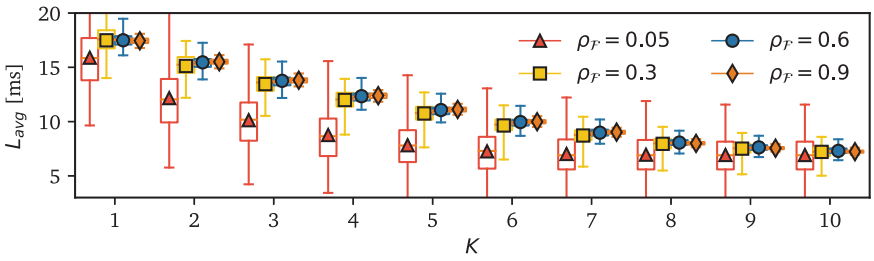
We develop a framework in Python for single-period DCPP and all subsequent optimization studies in this thesis. Gurobi [197] is selected as the optimization problem solver.

4.4.4.1 Evaluation Settings

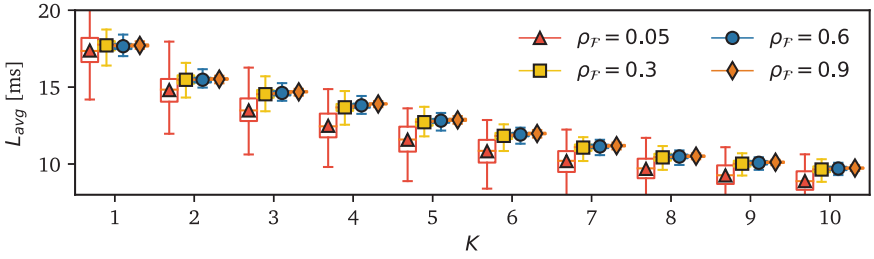
The parameter settings are given in Table 4.4. The network topologies under study are Abilene and AttMpls (ATT North America), both taken from the Topology Zoo data set [116]. We evaluate the model for different number of controllers: K ranges from 1 to the total number of nodes of the topology $|V|$. Flow density ρ_f is introduced as a parameter to define the level of sparseness of new flow distribution, i.e., all new flows defined as *flow profile*, in the data plane. Each flow profile consists of $|V|^2 \cdot \rho_f$ unique Source-Desitination (S-D) pairs in random. For each S-D pair, the number of new flows is the quotient of total traffic volume (following log-normal distribution [117]) and average data rate per flow.

4.4.4.2 Results

How does flow density influence the controller placement? Figure 4.4 portrays the controller placement distribution for Abilene network topology of our proposed model. The x-axis and y-axis represent the selections of two controller locations. When $\rho_{\mathcal{F}}$ is low, more than half of the controller location combinations are optimal at least once among all simulation runs. For denser flow distributions (e.g., $\rho_{\mathcal{F}} = 0.9$), the locations of two controllers converge to node 4 and 7, one in the center and the

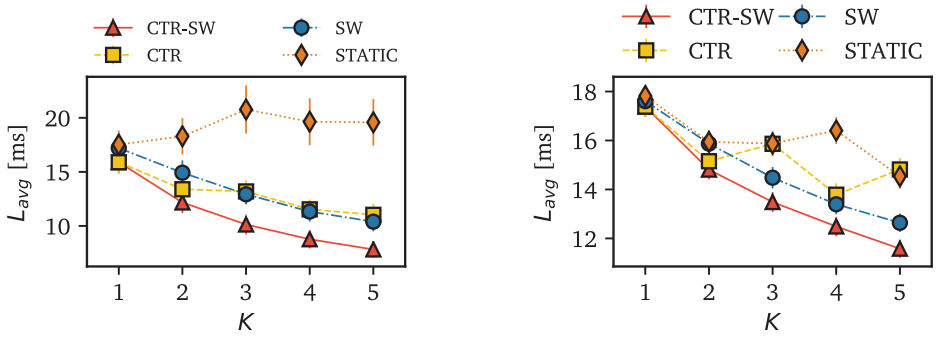


(a) Abilene.



(b) AttMpls.

Figure 4.5: Optimal average flow setup time (in milliseconds [ms]) of CTR-SW for different flow densities as a function of number of controllers $K \in [1, 10]$; one figure per network topology. Marker indicates the average of all simulation runs.

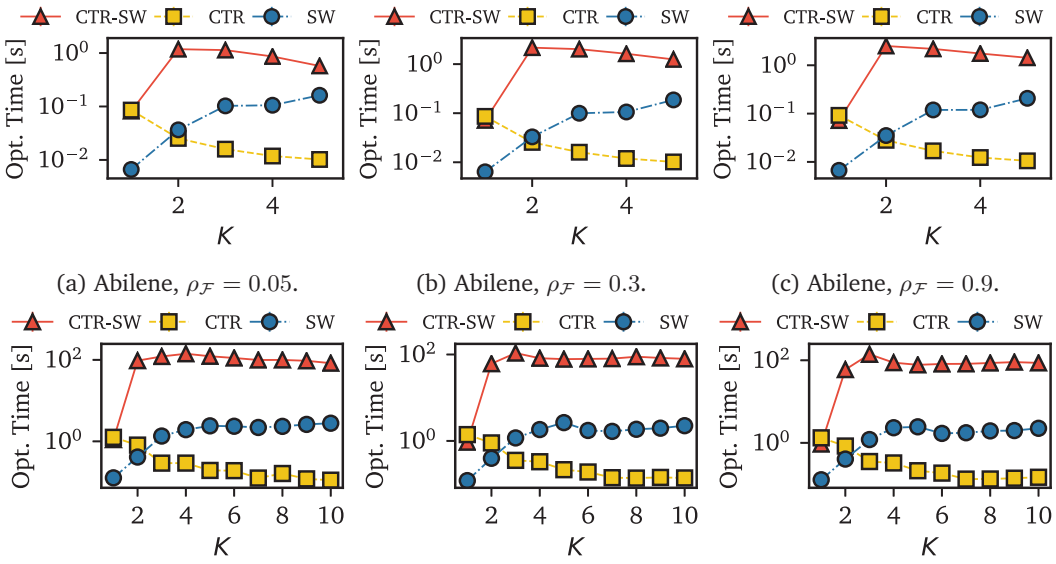


(a) Abilene. (b) AttMpls.

Figure 4.6: Optimal average flow setup time (in milliseconds [ms]) of different models; one figure per network topology. Flow density $\rho_{\mathcal{F}} = 0.05$ which represents a scarce flow distribution. Marker indicates the average of all simulation runs, and error bar indicates 99.9% confidence interval.

other in the west side of the topology. Indeed, graph feature has a critical impact on the placement decisions [118]: when $\rho_{\mathcal{F}}$ grows, all flows tend to distribute equally in the network topology and therefore nodes in the center with higher node degree are more likely to be selected.

Figure 4.5 shows the objective of the average flow setup time denoted as L_{avg} with respect to the number of controllers K. In general, more controllers lead to smaller L_{avg}: with larger K each flow has a higher chance of traversing more control domains, but each involved flow setup has shorter latency. Adding controllers yet yields less than linear reduction in the objective L_{avg}. A saturation point is observed in Figure 4.5a when $K \geq 9$. Further, as flow density $\rho_{\mathcal{F}}$ increases, the distribution of L_{avg} is narrower, and the mean also increases slightly, because more flows make it harder to satisfy them at the same time.



(a) Abilene, $\rho_{\mathcal{F}} = 0.05$. (b) Abilene, $\rho_{\mathcal{F}} = 0.3$. (c) Abilene, $\rho_{\mathcal{F}} = 0.9$.

(d) AttMpls, $\rho_{\mathcal{F}} = 0.05$. (e) AttMpls, $\rho_{\mathcal{F}} = 0.3$ (f) AttMpls, $\rho_{\mathcal{F}} = 0.9$

Figure 4.7: Optimization runtime (in seconds [s]) of different models. One figure per network topology and flow density $\rho_{\mathcal{F}}$. Optimization of CTR-SW takes the longest time due to the number of variables it considers.

What is the advantage of adapting to dynamic flows? To answer this question, we compare the objective of all four models. Figure 4.6 depicts the comparison of both topologies for $\rho_{\mathcal{F}} = 0.05$. The three models that are able to adapt, i.e., CTR-SW, CTR and SW, outperform STATIC, which sticks to the placement of a random flow profile throughout all simulation runs. For Abilene, L_{avg} of STATIC is even twice as that of CTR-SW of Abilene, indicating a significant improvement. Compared with partial adaptation as in CTR and SW, CTR-SW always guarantees the best L_{avg}. In the meantime, the gap between CTR-SW and SW does not change much, because of the positive correlation between the average control latency objective and our objective. CTR suffers from poor performance in some cases, e.g., when $K = 3$ for AttMpls, due to the impact of spectral clustering which targets equal-sized clusters.

Optimum in terms of L_{avg} comes with non-negligible optimization time. Figure 4.7 illustrates the optimization time of the four models. CTR-SW takes for $K > 1$ about 10 times and 100 times that of $K = 1$. It is also more complex to solve comparing with CTR and SW. Nevertheless, there is no obvious difference for problem instances with different $K \geq 2$ for CTR-SW, meaning more controllers do not induce a harder problem.

To conclude, how flows are distributed in the data plane plays a critical role in the controller placement. For a series of flow profiles, if flows are equally distributed with small perturbation, it would not be necessary to adapt the control plane as the optimal state does not change. On the contrary, it makes more sense to re-optimize when the flow distribution changes notably from time to time, which is more likely in modern SD-WAN [83]. Our proposed model of CTR-SW indeed provides minimum average flow setup time. When optimization is time-bounded, SW model is recommended because of its performance in terms of L_{avg} and smaller optimization time.

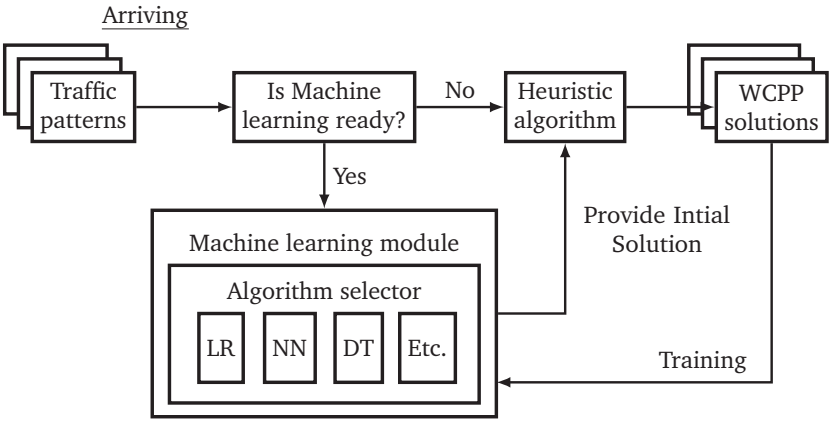


Figure 4.8: Workflow of data-driven network optimization with ML. Traffic patterns change over time. The heuristic algorithm module first generates a number of solutions, which will be used to train the ML module. For different available ML algorithms, the one which best fits the current setup will be selected. After training, the ML module processes new traffic patterns and creates initial solutions, which could be further improved by the heuristic algorithm module.

4.4.5 Towards Algorithm-Data Driven DCPP Optimization

Optimizing the control plane is a time-consuming task, therefore it is critical to propose efficient solution techniques. Meanwhile, optimization of the models introduced in the previous section naturally generates a wealth of data which can be potentially useful for future optimization, which, however, has remained untapped so far. This section initiates the study of data-driven network optimization with ML for DCPP subject to time-varying traffic patterns. In particular, it exploits the possibility to *learn from past executions of network algorithms*. Previous study [119] shows the potential to extend network algorithms of static networking placement problems by ML with promising performance. This section moves one step further and demonstrates that data produced by algorithm during past executions can be leveraged effectively to improve and speed up similar solutions in the future, by reducing algorithm's search space.

4.4.5.1 The Machine Learning Approach

Figure 4.8 illustrates the workflow of the data-driven network optimization approach for DCPP.

Problem Formulation

We consider a simplified version of DCPP (similar to the k-median problem [46]) without losing generality. The dynamic traffic is now consolidated as the volume of flow setup requests $\mathcal{R}(v)$, defined as a mapping $\mathcal{R}(n) \rightarrow [0, \infty), v \in \mathcal{V}$, from the underlying flow profile $\mathcal{F}$. Given a distribution $R(\cdot)$, the problem's objective is to determine the location of controllers, which minimize the weighted control latency:

$$\text{minimize} \quad \sum_{\phi \in \Phi} \sum_{c \in \mathcal{C}} \sum_{v \in \mathcal{V}} R(v) \cdot \ell(v, c) \cdot a_{\mathcal{V}, \Phi, \mathcal{C}}(v, \phi, c), \tag{4.15}$$

subject to the constraints (4.5), (4.6) and (4.7).

Algorithm 1 Greedy Algorithm

Input: Number of controllers K
Output: $\mathcal{C}$

```
1: Controller Set C ← ∅
2: while |C| < K do
3:     n ← argmin_n(F(C ∪ {n}))
4:     C ← C ∪ {n}
5: end while
```

Algorithm 2 Objective function Calculation $\mathcal{F}(*)$

Input: Latencies between nodes $\mathcal{L}$, traffic load $\mathcal{R}$, set of switches $\mathcal{N}$, controller set $\mathcal{C}$

```
1: Get switch to controller mapping A
2: obj ← 0
3: for n ∈ N do
4:     obj ← obj + R(n) · L(n, A(n))
5: end for
6: obj ← obj/|N|
7: return obj
```

Algorithm 3 Local Search Algorithm

Input: Number of controllers K
Output: $\mathcal{C}$

```
1: Random generate controller set C, s.t. |C| = K
2: Non-controller node set C̄ ← N \ C
3: obj* ← F(C), stopflag ← False
4: repeat
5:     obj** ← ∞, C** ← ∅
6:     for c ∈ C do
7:         for c̄ ∈ C̄ do
8:             C ← (C − {c}) ∪ {c̄}
9:             if F(C) < obj** then
10:                obj** ← F(C), C** ← C
11:            end if
12:        end for
13:    end for
14:    if obj* > obj** then
15:        obj* ← F(C**), C ← C**
16:    else
17:        stopflag ← True
18:    end if
19: until stopflag
```

Greedy and Heuristic Algorithm

Greedy algorithm provides a simple idea to address the optimization problem. Algorithm 1 incrementally inserts controllers into a set until the number K is reached. Each insert is performed to make sure that the objective function Eq. (4.15) stays optimal. Algorithm 2 describes the process of calculating the value of Eq. (4.15), which serves as a subroutine for other algorithms. The local search algorithm has been proved to be a promising heuristic for *k*-median [120] that outperforms the greedy algorithm. We introduce Algorithm 3 based on local search. During initialization, it randomly generates a controller set and calculates the corresponding objective. Line 5 to Line 18 are repeated until no further dominant solutions in terms of the objective function value can be found. Each time we perform $|\mathcal{C}| \cdot (|\mathcal{V}| - |\mathcal{C}|)$ local searches, from which the best one will be compared with the incumbent. A local search move represents swapping a node in set $\mathcal{C}$ with a node in set $\mathcal{V} \setminus \mathcal{C}$.

DCPP as Multi-label classification task

Multi-label classification[6] [110] is applied to predict controller locations. Table 4.1 summarizes an interpretation of the symbols. One observation represented by the feature vector $\mathbf{x}^{(i)}$ is associated with a *set* $\widetilde{\mathcal{I}}^{(i)} \subseteq \mathcal{I}$ of labels. Each traffic distribution $\mathcal{R}^{(i)}$ is a feature vector $\mathbf{x}^{(i)}$, i.e., $\mathbf{x}_j^{(i)} = \mathcal{R}^{(i)}(n_j)$. The node identifiers in network topology are labels; the i^{th} label $l_i \in \mathcal{I}$ identifies the i^{th} substrate node $n_i \in \mathcal{N}$. The controller placement for the traffic distribution $\mathcal{R}^{(i)}$ is represented by the labeling $\widetilde{\mathcal{I}}^{(i)}$, identifying the substrate nodes on which controllers are placed. Labels for different traffic distributions might change, yet classifiers generally do not have the capabilities to output varying sets. Therefore,

[6] Multi-label classification differs from multi-class classification, as in multi-class classification, exactly *one* label $l^{(i)} \in \mathcal{I}$ is associated with one observation $\mathbf{x}^{(i)}$. Multi-label classification is used, for example, in computer vision for sentiment analysis, where different images are associated with possibly different sentiments [121].

the labeling is represented by a binary vector $\mathbf{z}^{(i)}$, where the j^{th}-component $\mathbf{z}_j^{(i)}$ is set to one, if and only if $l_j \in \widetilde{\mathcal{I}}^{(i)}$, i.e., a controller is placed on node n_j. The prediction of a classifier is represented by a binary vector $\mathbf{y}^{(i)}$, which is calculated based on $\mathbf{x}^{(i)}$, i.e., $\mathbf{y}^{(i)} = h\left(\mathbf{x}^{(i)}\right)$, where $h(\cdot)$ represents a classifier. Three different multi-label classifiers, namely DT, NN and LR, are considered.

DT employs the *binary relevance* approach [122]. One tree for each possible label $l \in \mathcal{I}$ is built and trained independently from all others. A prediction $\mathbf{y}$ is a binary vector, where the i^{th} component is one if the respective tree detects the corresponding label. In theory, it is possible that more than K components in $\mathbf{y}$ are set to one. In this case, we randomly select K of the entries.

LR also predicts whether a controller is placed on each node. It produces a value between zero and one for each node, which can be interpreted as the *probability* of a controller to be located on that node. We take the K labels with highest probability as controller locations. To achieve this, the weighted sum of all input values is passed through a Sigmoid function. For each label, a distinct set of weights for the weighted sum is used. The weights are jointly updated using the gradient of a loss function, which is in our case the Bernoulli cross entropy loss. Since the weights are jointly updated, this approach does *not* amount to binary relevance, and is able to exploit correlations between single labels in labelings [122].

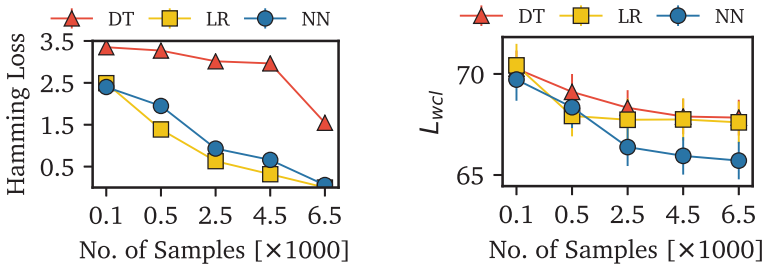
NN can be viewed as LR, where the input undergoes a sequence of nonlinear transformations when propagates through the network. The output of the last hidden layer is then used as input to LR. NN is thus also able to exploit label correlations, and additionally nonlinear dependencies in the inputs [122].

4.4.5.2 Results Evaluation and Analysis

We evaluate 6 network topologies with different sizes from the Topology Zoo data set [116]: AttMpls (with 25 nodes), Bics (33), Cernet (40), Uninett2010 (74), Deltacom (99) and Cogentco (180). For each topology, we test different Ks ranging from 5 to 20, with an interval of 5. The traffic on each node is randomly generated following a uniform distribution $U(1, 100)$. For NN classifier, we create 1 hidden layer with $|\mathcal{C}|$ neurons. Besides, it uses Sigmoid as activation function, and ADAM [123] with step rate of 0.01 and regularization factor of 10^{-5} for training. NN-LS represents the local search algorithm (described in Algorithm 3) with NN prediction and Greedy (GDY) denotes the greedy algorithm (described in Algorithm 1). Local search generates data samples by solving $7\,000$ problem instances for each substrate. $6\,500$ samples are used for training the algorithms and 500 samples are used for evaluation. Parameter tuning for the neural network was performed on a separate data set, with $6\,500$ samples for training and 500 samples for validation. Samples were obtained on the Bics substrate with K fixed to 3. We use *Hamming Loss* as defined in Eq. 4.3 between ground truth and classifier prediction to measure predictive accuracy, i.e., how many controllers are inaccurately predicted.

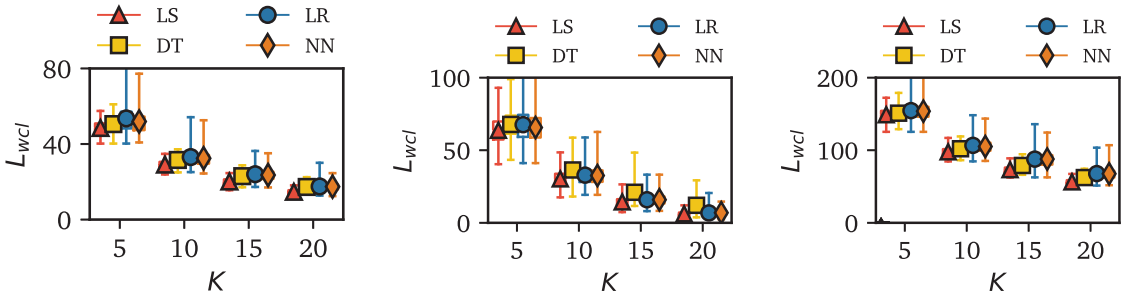
How much data is needed? Figure 4.9 shows Hamming Loss on the test set, as well as the weighted average control latency (denoted as metric L_{wcl}) as a function of the training set size for the Bics network topology. For the weighted average control latency in Figure 4.9b, the means with $95\,\%$ confidence intervals are shown. As the number of training samples increases, both Hamming Loss and L_{wcl} decrease. Interestingly, LR outperforms NN in terms of Hamming Loss, which is not the case of L_{wcl}. The difference in the prediction accuracy between DT and LR also cannot translate to L_{wcl}. Furthermore, DT and LR yield nearly the same objective starting from $4\,500$ samples.

Which ML algorithm is better? Figure 4.10 compares the three ML algorithms for three topologies and all Ks in terms of L_{wcl}, with LS as a baseline. NN and LS give competitive results. DT is outperformed



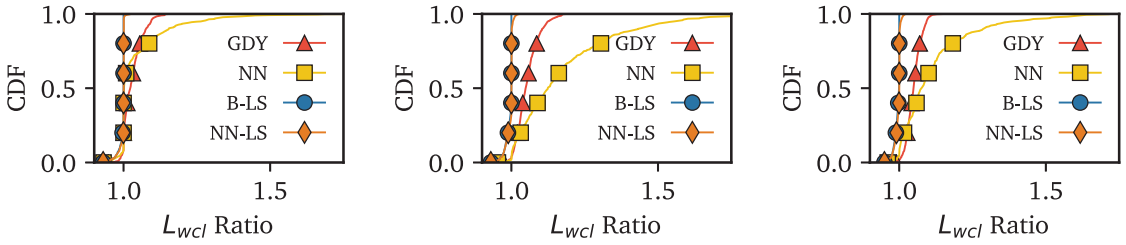
(a) Prediction accuracy. (b) Objective function.

Figure 4.9: The prediction evaluation with different training set sizes for Bics network topology ($K = 5$).



(a) Deltacom. (b) Bics. (c) Cogentco.

Figure 4.10: The comparison of different ML algorithms for different network topologies in terms of the optimization objective.



(a) Uninett2010, $K = 5$. (b) Uninett2010, $K = 10$. (c) Deltacom, $K = 10$.

Figure 4.11: The comparison of different algorithms for two particular setups in terms of the optimization objective.

by NN and LS on Bics, however, DT outperforms NN and LS on Cogentco in Figure 4.10c. Especially the variability of L_{wcl} obtained with the solutions of DT on Cogentco is small, in fact almost as small as the variability of LS. The respective differences of DT compared to LS and NN become more pronounced as K increases. Therefore, an exact prediction of controller placement is still hard to achieve. Only NN results are shown in the subsequent studies.

How much can we gain from leveraging data-driven approach? Since the performance of LS depends on the initial solution, we design a straw-man case namely B-LS: LS runs 10 times with different random initial solution, and the one that performs the best is the final solution. Figure 4.11 compares two particular problem setups. We quantify the performance gain of different algorithms in comparison to LS, i.e. $L_{\text{wcl}}(\text{alg})/L_{\text{wcl}}(\text{LS})$ for each sample. The cumulative distributive function is plotted of all 500 samples in the test set. In general, the performance of NN-LS is on a par with that of LS, indicating that NN provides good initial solutions for local search. For B-LS, we conclude that the

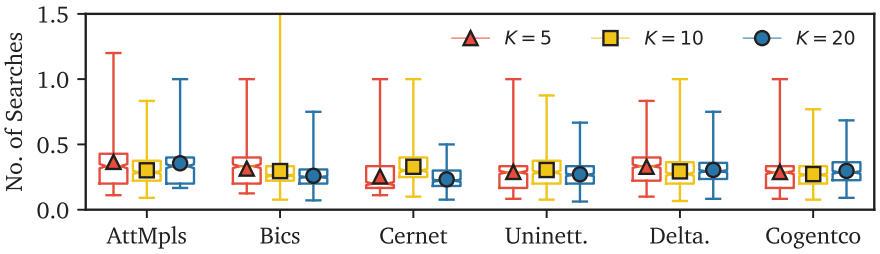


Figure 4.12: Comparison of all evaluated topologies in terms of the runtime saving.

objective cannot be significantly improved with different random initial solutions. Notably, different setups provide distinct comparisons for NN and GDY.

How much can we gain from leveraging data-driven approach? Figure 4.12 compares normalized algorithm runtime savings. For each sample, the number of searches in NN-LS is divided by that in LS. For all topologies NN-LS takes only one third the number of searches for most samples, independent of K. For a heuristic algorithm that starts with an initial feasible solution, applying ML prediction can result in solutions of decent quality and reduce runtime significantly.

To conclude, we demonstrate that ML can indeed be used to learn from past algorithm's solutions and can be applied directly to address control plane optimization problems. DCPP is phrased as a multi-label classification problem, where input is traffic intensity at each node in the topology, and the output is a set of node labels representing the controller locations. The predicted placement can be used either directly as a solution, or as an initial solution for other heuristic algorithms. Our approach shows promising results: NN prediction as initial solution for heuristic could save considerable amount of algorithm runtime.

4.5 Multi-Period Dynamic Controller Placement Problem

This section introduces the optimization of the dynamic control plane that is able to adapt itself during the course of its operation in multiple time-slots. In contrast to the single-period DCPP, decision variables, objective functions, and constraints of multi-period DCPP need to additionally consider the time-slot [124]. The single-period model can be used as a basis; thus this section only introduces the additional variables and constraints in detail. Moreover, the processing time of flow setup request, e.g., Packet-In defined in OpenFlow [180], at controller is considered in modeling the end-to-end flow setup time.

4.5.1 Network Model: Adaptable Dynamic Control Plane

Now the network model considers processing time (i.e., packet sojourn time) of each flow setup request on top of forwarding time. Processing time consists of queuing and service delays. Without losing generality, each controller is assumed to have one thread (i.e., one server), and the flow setup request arrivals follow a Poisson process [94]. Therefore, each controller can be modeled as an M/M/1 queue [97]. This model has been validated to approximate the processing time of a controller with one switch [97] and multiple switches [125], and applied in previous work of DCPP for SD-WAN [92]. Nevertheless, other more accurate mathematical models can be incorporated to extend the problem

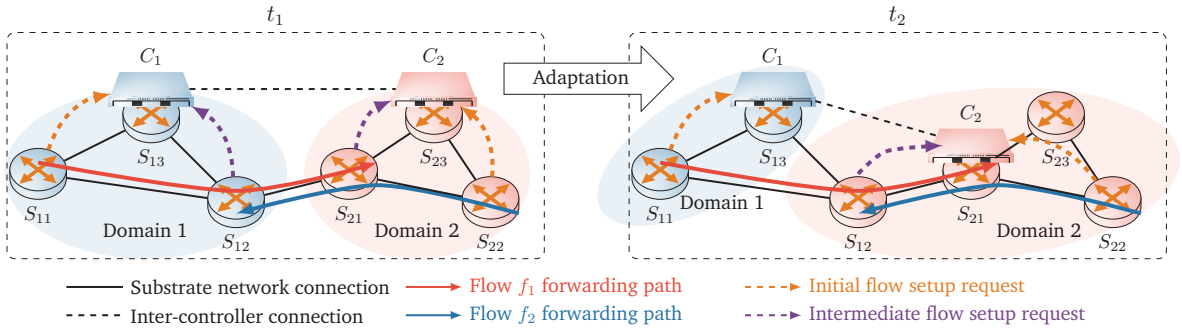


Figure 4.13: Illustration of control plane adaptation when a new demand of traffic distribution arrives. At t_2, controller C_2 migrates from the location of S_{23} to the location of S_{21}, and then S_{12} is reassigned from C_1 to C_2. For the two flows F_1 and F_2, the second controller placement at t_2 has less incurred control load (3 setup requests) compared with the first one at t_1 (4 setup requests).

formulation. The expected processing time $\mu_{\Phi}^{(t)}(\phi)$ of a controller ϕ is calculated as the follows:

$$\mu_{\Phi}^{(t)}(\phi) = \frac{1}{\theta_{\phi} - \lambda_{\Phi}^{(t)}(\phi)}, \tag{4.16}$$

where θ_{ϕ} represents the processing capacity of a controller and $\lambda_{\Phi}^{(t)}(\phi)$ represents the load of a controller in terms of the number of flow setup requests.

4.5.2 Modeling Control Plane Reconfiguration

Figure 4.13 illustrates a reconfiguration of the control plane. At t_1, each control domain consists of three switches. Two representative flows belonging to the old flow profile f_{t_1} (other flows are neglected in the figure) are sketched, each of them initiates two flow setup requests. When a new flow profile is present, single-period placement model, as introduced in Section 4.4, outputs a new optimal control plane state, where controller C_2 is first migrated to the location of S_{21} and then switch S_{12} is reassigned to C_2. This section models the reconfiguration process that incurs time T_{mig}, which is a critical factor to consider while evaluating the flexibility (introduced in Chapter 3). A control plane adaptation consists of controller migration and switch reassignment that are introduced in Section 2.1. To recap, controller migration time is composed of a transmission component $T_{\text{ctr\_tran}}$, as the data store needs to be transmitted from old location to new location, and a forwarding component $T_{\text{ctr\_prop}}$. Switch re-assignment, which incurs T_{sw}, takes place after controller migration, and it involves several rounds of control information exchange, in which forwarding latency plays the major role.

4.5.2.1 Controller Migration

We assume that the control plane is fully distributed and all controller instances have access to the network state, e.g., flows and network topology. The state is maintained in each controller's local data store [35]. During controller migration, the data store is transferred from the old controller to the new controller with a preserved bandwidth. We also assume that controllers can migrate simultaneously.

4.5.2.2 Switch Re-Assignment

After the controller starts running in the new location, switch also stops the old control channel and establish a new one with the new controller. OpenFlow specification [180], however, does not provide a

Table 4.5: Additional variables of multiple-period DCPP on top of the ones introduced in Table 4.3. All variables have superscript to represent decisions in each time-slot t.

Notation	Description
$\upsilon_{\mathcal{V}}^{(t)}(v)$	non-negative variable representing the amount of flow setup requests generated from a switch $v \in \mathcal{V}$ at time t
$\upsilon_{\mathcal{V},\Phi,\mathcal{C}}^{(t)}(v,\phi,c)$	non-negative variable representing the amount of flow setup requests generated from a switch $v \in \mathcal{V}$ assigned to controller with ID $\phi \in \Phi$ placed at node $c \in \mathcal{C}$ at time t
$\lambda_{\Phi}^{(t)}(\phi)$	non-negative variable representing the total amount of load in terms of the number of Packet-In messages on the controller with ID $\phi \in \Phi$ at time t
$\mu_{\Phi}^{(t)}(\phi)$	non-negative variable representing the expected Packet-In processing time of the controller with ID $\phi \in \Phi$ at time t
$\delta_{\mathcal{V}}^{(t)}(v,u)$	non-negative variable representing the necessary controller processing time if the flow goes from $v \in \mathcal{V}$ to $u \in \mathcal{V}$, and representing the controller processing time when the flow originates at v if $v = u$

native mechanism in reassigning SDN switch to different controller instances and leaves it as an open question to network providers. Nevertheless, during the re-assignment process of a switch, the liveness and safety properties should be guaranteed [84]. On one hand, a switch is always connected to at least one active controller. A controller that has issued Packet-Out and Flow-Mode messages as a response of Packet-In from a switch should not be turned off before the switch finishes the corresponding flow operation. On the other hand, a Packet-In should be responded exactly once and duplicated Flow-Mod messages to a switch may cause the same entry being pushed into the flow table repeatedly.

4.5.3 Problem Formulation

4.5.3.1 Problem Input and Variables

Multi-period DCPP considers $|T|$ time-slots in total. All variables have superscript "(t)" to represent the values at a particular time-slot t. For example, $\mathcal{F}^{(t)}$ represents the set of new flows at time-slot t. To support multi-period DCPP, five more types of variables are required, which are summarized in Table 4.5. Notably, $\upsilon_{\mathcal{V}}^{(t)}(v)$ is not a problem input, but a variable depending on other variables. For the same flow profile $\mathcal{F}^{(t)}$, different controller locations $p_{\Phi,\mathcal{C}}(\phi,c)$ and switch assignments $a_{\mathcal{V},\Phi,\mathcal{C}}(v,\phi,c)$ can result in different load distributions.

4.5.3.2 Objective Function

The problem is formulated as multi-period cost minimization problem: the total cost of operating the dynamic control plane at each time-slot $t \in T$ consists of the operational cost $C_F^{(t)}$ and the reconfiguration cost $C_M^{(t)} + C_R^{(t)}$.

Operational Cost

We define the operational cost as the average end-to-end flow setup time considering processing time of flow setup request $\delta_{\mathcal{V}}^{(t)}(v,u)$. Thus,

$$C_F^{(t)} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}^{(t)}} \Big[\underbrace{2 \cdot l_{\mathcal{V}}(s^f) + \delta_{\mathcal{V}}(s^f, s^f)}_{\text{Initial flow setup time } C_{F_1}^{(t)}} + \underbrace{2 \cdot \sum_{(v,u) \in \Omega(s^f, d^f)} \tau_{\mathcal{V}}(v,u) + \delta_{\mathcal{V}}(v,u)}_{\text{Intermediate flow setup time } C_{F_2}^{(t)}} + \underbrace{\ell(s^f, d^f)}_{\text{Forwarding time } C_{F_3}^{(t)}} \Big], \tag{4.17}$$

where $C_{F_1}^{(t)}$ and $C_{F_2}^{(t)}$ denote initial and intermediate flow setup time, respectively. The flow forwarding time $C_{F_3}^{(t)}$ is fixed for each flow and does not pose any impact on the decision variables, therefore can be left out in the problem formulation.

Adaptation Cost

Control plane adaptation in the face of traffic distribution (represented as flow profile) change consists of controller migration and switch reassignment, each induces a cost. The controller migration cost is modeled as the total latency induced by migrating controllers, given by:

$$C_M^{(t)} = \sum_{\phi \in \Phi} \sum_{c^{t-1} \in \mathcal{C}} \sum_{c^t \in \mathcal{C}} \Big[p_{\Phi,\mathcal{C}}^{(t-1)}(\phi, c^{t-1}) p_{\Phi,\mathcal{C}}(\phi, c^t) \ell(c^{t-1}, c^t) \Big], \tag{4.18}$$

where the forwarding latency $\ell(c^{t-1}, c^t)$ between the previous location c^{t-1} and the current location c^t is the factor that decides the migration time of each controller.

The switch reassignment cost is modeled as the total latency of reassigning switches from one controller instance to another controller instance, given by:

$$C_R^{(t)} = \sum_{v \in \mathcal{V}} \sum_{\phi \in \Phi} \sum_{c^{t-1} \in \mathcal{C}} \sum_{c^t \in \mathcal{C}} \Big[\Big[\ell(c^{t-1}, v) + \ell(c^t, v) \Big] \cdot a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c^{t-1}) a_{\mathcal{V},\Phi,\mathcal{C}}(v, \phi, c^t) \Big].$$

For each switch, the sum of its old and new control latency contributes to its reassignment delay. Note that the switch can be reassigned only after the controllers have been migrated. As an example in Figure 4.13, the controller migration cost is $\ell(S_{21}, S_{23})$ and the switch reassignment cost is $\ell(S_{12}, S_{13}) + \ell(S_{12}, S_{21})$.

Intuitively, the operational cost $C_F^{(t)}$ only depends on the decision variables at t. The adaptation cost $C_M^{(t)}$ and $C_R^{(t)}$, however, are functions of the difference of decision variables between $t-1$ and t. Furthermore, the scaling of controller instances can contribute to the adaptation cost [94]. Scaling controller instances vertically, e.g., increase the number of CPU cores of an instance, requires reboot of the Virtual Machine (VM) [126], which leads to temporary in-activeness of that instance and incurs high cost. Horizontal scaling, on the other hand, can take place beforehand, i.e. new controller instances are instantiated before they are actually being used. The cost in this case mainly involves the deployment of new VMs and the higher inter-controller synchronization, which can also be explored in the future.

Objective

With the two cost factors introduced above and the traffic profiles $\mathcal{F}^{(t)}, \forall t \in T$ as input, the objective is formulated as the follows:

$$\text{minimize} \quad \sum_{t \in T} \gamma_F \cdot C_F^{(t)} + \gamma_M \cdot C_M^{(t)} + \gamma_R \cdot C_R^{(t)}, \tag{4.19}$$

where γ_F, γ_M and γ_R denote the weighting factors for operational, controller migration, and switch reassignment costs, respectively.

4.5.3.3 Constraints

The additional constraints model the controller's processing time, which relates to the number of flow setup requests it needs to process. They accumulate the load of each controller in terms of the number of flow setup requests, and then calculate the processing time as the sojourn time accordingly by using queuing theory.

Build up flow setup requests at switch. It sums up the total number of flow setup requests, consisting of the initial and the intermediate ones. Function $\lceil x \rceil_v$ returns 1 if $x = v$, otherwise 0. Thus,

$$\sum_{f \in \mathcal{F}^{(t)}} \left[\lceil s^f \rceil_v + \sum_{(p,q) \in \Omega(s^f, d^f)} \lceil q \rceil_v \right] = \upsilon_{\mathcal{V}}^{(t)}(v), \quad \forall v \in \mathcal{V}. \tag{4.20}$$

Requests consolidation at controller. To consolidate the load from a control domain to its controller, only switches within the domain are considered as follows:

$$\upsilon_{\mathcal{V}}^{(t)}(v) \cdot a_{\mathcal{V},\Phi,\mathcal{C}}^{(t)}(v, \phi, c) = \upsilon_{\mathcal{V},\Phi,\mathcal{C}}^{(t)}(v, \phi, c), \quad \forall v \in \mathcal{V}, \forall \phi \in \Phi, \forall c \in \mathcal{C}. \tag{4.21}$$

Expectation of processing time. The following constraint models the expected controller processing time if a new flow originates at one switch:

$$\sum_{\phi \in \Phi} \sum_{c \in \mathcal{C}} \mu_{\Phi}^{(t)}(\phi) \cdot a_{\mathcal{V},\Phi,\mathcal{C}}^{(t)}(v, \phi, c) = \delta_{\mathcal{V}}^{(t)}(v, v), \quad \forall v \in \mathcal{V}. \tag{4.22}$$

For simplicity, we introduce two new variables $\mu_{\Phi}^{(t)}(\phi)$ and $\lambda_{\Phi}^{(t)}(\phi)$, defined as:

$$\mu_{\Phi}^{(t)}(\phi) = \frac{1}{\theta_\phi - \lambda_{\Phi}^{(t)}(\phi)}, \quad \phi \in \Phi; \text{ and} \tag{4.23}$$

$$\lambda_{\Phi}^{(t)}(\phi) = \sum_{c \in \mathcal{C}} \sum_{v \in \mathcal{V}} \upsilon_{\mathcal{V},\Phi,\mathcal{C}}^{(t)}(v, \phi, c), \quad \phi \in \Phi. \tag{4.24}$$

.

Expected processing time of each hop. Similar to Eq. (4.12), not all hops along the flow's forwarding path trigger processing of controller. Thus,

$$\sum_{\phi \in \Phi} \mu_{\Phi}^{(t)}(\phi) \cdot \sum_{c \in \mathcal{C}} a_{\mathcal{V},\Phi,\mathcal{C}}^{(t)}(u, \phi, c) \cdot \overline{d}_{\mathcal{V}}^{(t)}(v, u) = \delta_{\mathcal{V}}^{(t)}(v, u), \quad \forall f \in \mathcal{F}, (v, u) \in \Omega(s^f, d^f). \tag{4.25}$$

Algorithm 4 RHC

Input: ω, $\mathcal{F}$
1: **for** $t \in T$ **do**
2: Solve Problem **P'** over $(t, t+\omega)$ with SA-based algorithm;
3: Update the control plane state;
4: **end for**

Algorithm 5 FHC

Input: ω, $\mathcal{F}$
1: $T' \leftarrow \{t | t \bmod (\omega+1) = 1, t \in T\}$;
2: **while** $t < |T'|$ **do**
3: Solve the Problem **P'** over $(t, t+\omega)$ with SA-based algorithm;
4: Update the control plane state;
5: **end while**

4.5.3.4 Linearization

High order constraints, i.e., Eq. (4.21), (4.22) and (4.25), can be linearized without loss of optimality, following the approach that is described in Section 4.4.3.4. For Eq. (4.23), however, we need to apply piece-wise linear approximation and define the following linear relationship between the expected processing time of a flow setup request at a controller $\mu_{\Phi}^{(t)}(\phi)$ and the total number of flow setup requests the controller needs to process $\lambda_{\Phi}^{(t)}(\phi)$,

$$\mu_{\Phi}^{(t)}(\phi) = \begin{cases} \frac{1}{(\theta_\phi - \lambda_1)\theta_\phi}\lambda + \frac{1}{\theta_\phi}, & 0 \leq \lambda < \lambda_1; \\ \frac{1}{(\theta_\phi - \lambda_1)(\theta_\phi - \lambda_2)}\lambda - \frac{\lambda_1 + \lambda_2 - \theta_\phi}{(\theta_\phi - \lambda_1)(\theta_\phi - \lambda_2)}, & \lambda_1 \leq \lambda < \lambda_2. \end{cases} \tag{4.26}$$

Three segment points 0, λ_1 and λ_2 define two levels of controller load: low $[0, \lambda_1)$ and high $[\lambda_1, \lambda_2)$. Note that λ_2 is smaller than controller capacity θ_ϕ to model the spare capacity of controller, e.g., $\frac{\lambda_2}{\theta_\phi} \in [0.85, 0.95]$ [94].

We refer to the original optimization problem as Problem **P** and the new optimization one, which replaces Eq. (4.23) with Eq. (4.26), as Problem **P'**. Indeed, the approximation overestimates the processing time of controller and therefore does not guarantee that the global optimum of Problem **P'** equals to the global optimum of Problem **P**. We have the following proposition, whose proof is in Appendix A.

Proposition 1 *Suppose the controller capacity θ_ϕ is $1\,000$, and the segment points reside at 0, 700 and 900. Solving the linearized problem **P'** optimally achieves 1.79-approximation of the original problem **P**.*

4.5.4 Design of Efficient Algorithms

4.5.4.1 Look-ahead Control Scheme

We leverage look-ahead control to address our cost optimization problem. With only a limited knowledge of future input [108], i.e., input within a look-ahead window with size ω, look-ahead control decomposes an offline problem into a series of online sub-problems, each with a small number of time-slots (equals to ω). We introduce two types of look-ahead control: RHC [107] in Algorithm 4 and FHC [94] in Algorithm 5. RHC solves the optimization problem at each time-slot t over the look-ahead window $(t, t+\omega)$. Each optimization takes place given the system state of the last time stamp $t-1$. Thereafter, only the decision variables of the first time-slot t are applied and the remaining decision variables, i.e., of $t+1, ..., t+\omega$, are discarded. FHC, on the other hand, keeps all the decision variables of time-slot $t, .., t+\omega$ and directly jumps over the current look-ahead window ω. Figure 4.1 illustrates the differences

Algorithm 6 SA-Based Algorithm

Input: ω

1: Initialize R_c, R_b, obj_b, $cnt_b \leftarrow 0$, $temp \leftarrow T_i$
2: **while** $temp \geq T_s$ **do**
3: $r_i \leftarrow 0$ ▷ Run id
4: **for** $r_i < R$ **do**
5: Randomly call RS, SS, or RC and get neighbor result R_n
6: $obj_c, obj_n \leftarrow$ EVALUATE(R_c, R_n)
7: **if** $obj_b \geq \min(obj_c, obj_n)$ **then**
8: $cnt_b \leftarrow 0$ ▷ Best solution updated
9: Copy the better one of R_c and R_n to R_b
10: **else**
11: $cnt_b \leftarrow cnt_b + 1$ ▷ Best solution kept
12: **end if**
13: $\delta \leftarrow (obj_c - obj_n)/obj_n$
14: **if** $\delta > 0$ **then**
15: Copy R_n to R_c ▷ Neighbor is better
16: **else**
17: $prob \leftarrow \exp(\delta/temp)$ ▷ Probability of acceptance
18: $r \leftarrow U(0, 1)$ ▷ Random number
19: **if** $r < prob$ **then**
20: Copy R_n to R_c ▷ Accept worse neighbor
21: **end if**
22: **end if**
23: **if** $cnt_b > CNT$ **then**
24: Return R_b ▷ Early stop
25: **end if**
26: **end for**
27: $temp \leftarrow temp \cdot \alpha$ ▷ Update temperature
28: **end while**
29: Return R_b ▷ Normal stop

between the two alternatives. FHC guarantees shorter running time by solving fewer optimization problems ($\lceil |T|/(\omega + 1) \rceil$) compared to RHC ($|T|$).

4.5.4.2 Simulated Annealing

Algorithm 6 describes an SA-based algorithm with four parameters determine the total number of random searches (i.e., iterations, one iteration from Line 5 to Line 22): initial temperature T_i, stop temperature T_s, temperature update ratio α, and number of searches per temperature R. The symbol R_c, R_n and R_b represent the current, the neighbor and the best result found respectively, each with objective function value obj_c, obj_n and obj_b. Symbol $delta$ denotes the difference of objective function values between the current and the neighbor result. Symbol cnt_b denotes the number of iterations with which the best result does not change.

In each iteration, the solution space of the controller placement problem at one time-slot (defined as a *problem slice*) is explored with one of the three following procedures under equal probabilities (Line 5). *(i)* Procedure REASSIGNSWITCH (RS) randomly selects one switch from a control domain with size larger than 1 and reassigns it to another control domain. *(ii)* Procedure SWAPSWITCH (SS) randomly selects two switches from two control domains and swaps their controllers. *(iii)* Procedure

RELOCATECONTROLLER (RC) randomly selects a controller and change its location to that of another switch of its control domain. Procedure EVALUATE calculates the objective function value. If the look-ahead window size ω is non-zero, the algorithm randomly selects a *problem slide* at one time-slot $t \in \{0, 1, ..., \omega - 1\}$. An early-stop mechanism (Line 24) can expedite the algorithm by returning the best solution R_b, if it is not improved for CNT steps. Otherwise, the algorithm continues until the current temperature $temp$ is lower than T_s.

4.5.4.3 Complexity Analysis

Considering the worst-case (i.e., without early-stop), the SA-based algorithm needs for RHC $|T|\lceil R \cdot \log_\alpha \frac{T_s}{T_i} \rceil$ or for FHC $\lceil T/(\omega+1) \rceil \cdot \lceil R \cdot \log_\alpha \frac{T_s}{T_i} \rceil$ iterations, which are independent from the number of switches and controllers. In each iteration, procedure RS can go through all controllers and switches to find a switch that can be reassigned, resulting in $O(|\Phi| + |\mathcal{V}|)$ complexity. Procedure SS can search over all possible pairs of controllers, which indicate a complexity of $O(|\Phi|^2)$. For Procedure RC, all controllers may need to be examined, resulting in a complexity of $O(|\Phi|)$. The remaining steps of an iteration contribute to $O(1)$. Therefore, the overall worst-case complexity is $O(|\Phi| + |\mathcal{V}|) + O(|\Phi|^2) + O(|\Phi|) + O(1) = O(|\Phi|^2 + |\mathcal{V}|)$.

4.5.5 Results Evaluation and Analysis

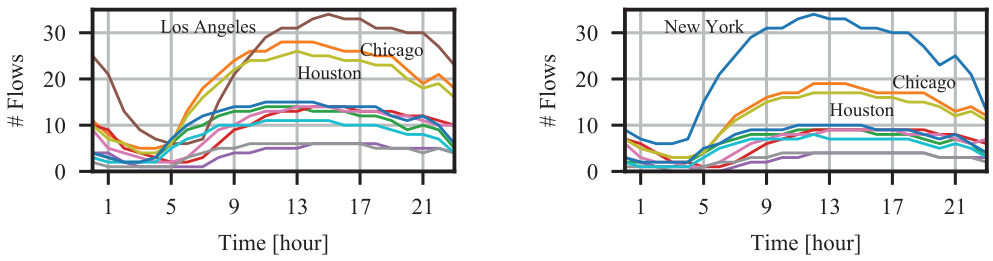
4.5.5.1 Evaluation Settings

We evaluate three network topologies from the Topology Zoo [116]: Abilene, AttMpls, and OS3E. The number of controllers K varies between 2 and 5 and its impact on the evaluation metrics is analyzed. We use the same traffic model as in Section 4.4 to generate synthetic random traffic for $|T| = 30$. Note that there is a positive correlation between the size of the topology and the number of generated new flows (flow setup requests). To avoid overloading the controllers, we set the controller capacity for the three topologies as $1\,000$, $5\,000$ and $10\,000$, respectively. The segment points for linear approximation are 0, 0.7 and 0.9 times the respective controller capacity.

The following algorithms are under comparison:

1. `STA`: We fix the controller locations and keep the switch assignment, which is the optimal solution of minimizing the average control latency for the first time-slot and does not consider control plane reconfiguration;
2. `CNPA`: We take another state-of-the-art algorithm adapted from [92] which is designed based on topology clustering. We need to amend the original algorithm, so that the clustering can take the dynamics of the flows into account.
3. `OPT(`ω`)`: Due to huge computational complexity, it is not realistic to optimize for the whole time horizon, i.e., the full time-slot set T.
4. `RHC(`ω`)`: Similarly, we solve the sub-problems with our proposed online algorithm based on RHC and SA with different look-ahead window sizes.
5. `FHC(`ω`)`: We solve the cost minimization problem with FHC as the online algorithm wrapper.

Realistic Traffic. In the SD-WAN scenario, we assume that data plane traffic demand volume between source and destination node depends on both node's population data [31], [199]. There is shifting of



(a) Traffic originated from New York.

(b) Traffic originated from Los Angeles.

Figure 4.14: Daily traffic patterns of different source nodes in a US WAN topology: one on the east coast and the other on the west coast. For each source node, the curves corresponding to different target nodes shift. On the other hand, the peak points among all curves of the two source nodes appear at different time.

the traffic demand curve between the same source node to different destination nodes, and between different source nodes to the same destination nodes. Figure 4.14 shows the traffic patterns of two source nodes. The y-axis is the traffic volume in terms of the number of flows. Since New York and Los Angeles are the two cities with the largest population, the highest curves represent the traffic between them, i.e., brown curve NY-LA in Figure 4.14a and blue curve LA-NY in Figure 4.14b. The maximum points of the these two curves, however, correspond to different time-slots: NY-LA at 15:00 and LA-NY at 12:00.

4.5.5.2 Results

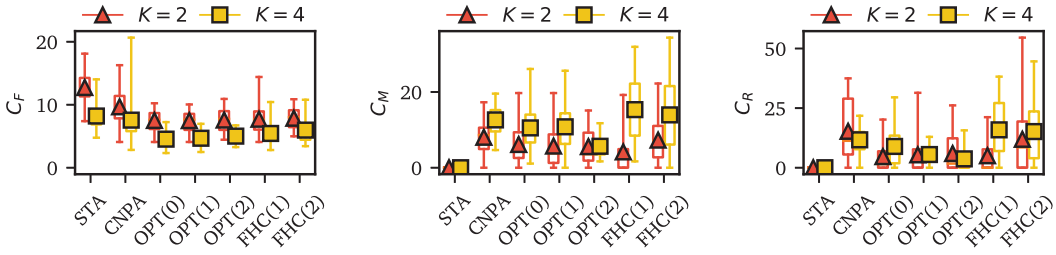
After performing a parameter study, we select the following parameter combination: $T_i = 0.5$, $T_s = 0.001$, $R = 500$ and $\alpha = 0.95$, for the subsequent evaluations of the SA-based algorithms.

How good are our algorithms? Figure 4.15 plots the three cost factors on Abilene network topology with $\rho_{\mathcal{F}} = 0.05$ and K equals 2 and 4. We set the reconfiguration cost with low priority (i.e., $\gamma_F = 1$, $\gamma_M = \gamma_R = 0.1$) and compare `STA`, `CNPA`, `OPT(0/1/2)` and `FHC(1/2)`. `STA` has the worst flow setup performance on average, but enjoys zero controller and switch reconfiguration due to its static nature. Compared to `OPT(0/1/2)`[7] and `FHC(1/2)`, `CNPA` can achieve similar performance in terms of controller reconfiguration, but at the sacrifice of worse the other two cost factors. `FHC(1/2)` can achieve similar flow setup performance compared with `OPT(1/2)`. However, when it comes to the reconfiguration, `FHC(1/2)` does not always promise smaller reconfiguration latency. We raise the priority of reconfiguration (i.e., $\gamma_F = 1$, $\gamma_M = \gamma_R = 0.5$) and compares them again in Figure 4.16. As a proper response to the higher priority, `OPT(0/1/2)` pushes C_M and C_R down, which is not revealed for `FHC(1/2)`.

Figure 4.17 shows the detailed overall cost values for Abilene topology (i.e., $K = 4$) for all time-slots in T. We can observe that for low flow densities, the curves of `OPT(0/1)` and `FHC(0/1)` intertwine and there is no absolute winner. `CNPA`, however, always delivers the worst overall cost values, on average 2 times that of the other algorithms, and in the worst case, 4 times. When the flow density becomes larger, the trend of `CNPA` does not change. It can be seen that the difference between `OPT(0)` and `FHC(0/1)` is marginal, and they are slightly better than `OPT(1)`.

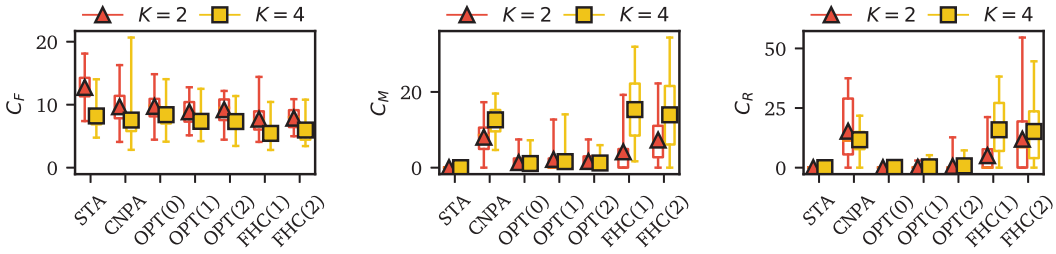
How long should we look into the future? This is an interesting question for all types of look-ahead control schemes. Intuitively, larger window size leads to lower overall cost objective, because it can

[7] `OPT(0/1/2)` represents all the three cases of `OPT(0)`, `OPT(1)` and `OPT(2)`. Same for the notation of other algorithms.



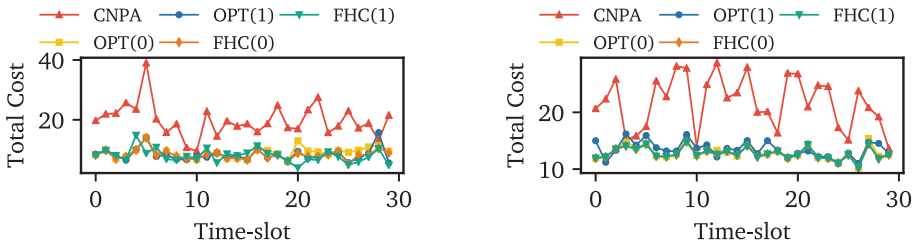
(a) Operational cost. (b) Controller migration cost. (c) Switch reassignment cost.

Figure 4.15: Performance of cost minimization of various algorithms with $\gamma_F = 1$, $\gamma_M = 0.1$, $\gamma_R = 0.1$, and $\rho_{\mathcal{F}} = 0.05$ in Abilene network topology. Box-plots are drawn with 30 runs of each case (and for the subsequent results unless specified).



(a) Operational cost. (b) Controller migration cost. (c) Switch reassignment cost.

Figure 4.16: Performance of cost minimization of various algorithms with $\gamma_F = 1$, $\gamma_M = 0.5$, $\gamma_R = 0.5$, and $\rho_{\mathcal{F}} = 0.05$ in Abilene network topology.

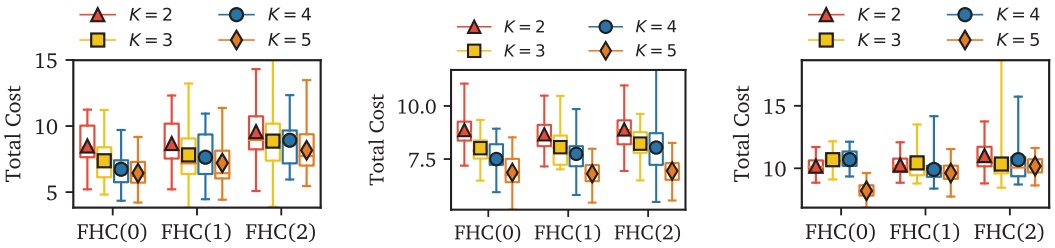


(a) Low flow density $\rho_{\mathcal{F}} = 0.05$. (b) High flow density $\rho_{\mathcal{F}} = 0.3$.

Figure 4.17: Overall cost in each time-slot with $\gamma_F = 1$, $\gamma_M = 0.5$, $\gamma_R = 0.5$ and $K = 4$ in Abilene network topology.

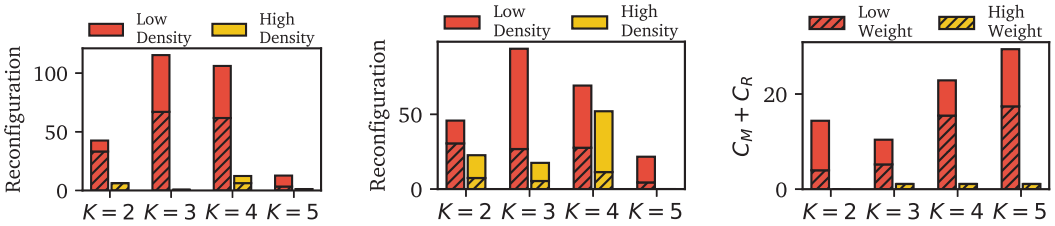
achieve a better trade-off between operational cost and reconfiguration cost by considering more time slots. However, this intuition is not reflected in our evaluation results. Figure 4.18 compares the total cost of different look-ahead window sizes for different topologies with $K \in 2, 3, 4, 5$. We observe that the total cost objective increases slightly with a larger ω due to the positive correlation between the complexity of the online problem and the window size. When the complexity increases, our proposed algorithm takes longer to converge and is likely to output a less optimal solution. Nevertheless, when we look into the three cost coefficients, we notice that operational cost is the only one that actually decreases (Figure 4.15a and Figure 4.16a).

Do we need frequent reconfigurations? The frequency of control plane reconfiguration impacts the induced cost, and it is not always required to have many reconfigurations. Figure 4.19a to Figure 4.19b plot the reconfiguration costs for two network topologies with synthetic random traffic. The lower



(a) Abilene. (b) AttMpls. (c) OS3E.

Figure 4.18: Comparison of total cost with $\gamma_F = 1$, $\gamma_M = 0.1$, $\gamma_R = 0.1$, and $\rho_{\mathcal{F}} = 0.05$ comparing different look-ahead window sizes ω from 0 to 2, different number of controllers K, and different topologies. Results obtained using FHC algorithm.



(a) AttMpls. (b) OS3E. (c) Abilene (Realistic Traffic).

Figure 4.19: Reconfiguration cost of different traffic distribution with cost coefficient γ_F, γ_M, γ_R equals to 1, 0.1 and 0.1 respectively (over 30 runs) obtained from different topologies for (a) and (b). In (c), we compare $\gamma_F = 1$, $\gamma_M = 0.1$, $\gamma_R = 0.1$ (low weight) and $\gamma_F = 1$, $\gamma_M = 0.5$, $\gamma_R = 0.5$ (high weight).

(shaded) and upper part of each bar represents C_M and C_R, respectively. In general, we can observe that when the traffic distribution is more evenly distributed (i.e., high flow density), the intention of the reconfiguration becomes less obvious. The control plane can stay unchanged for most flow profiles with good flow setup performance. For Abilene and low flow density, more controllers lead to higher reconfiguration cost. Nevertheless, the trend becomes different for the other two topologies, where the cost reaches the maximum at $K = 3$ and drops afterwards.

For the case of random traffic, there is no spatial or temporal correlation among the flows of different source and destination pairs. Therefore, it represents the worst case in terms of controller and switch reconfiguration cost. We apply the traffic introduced before on the Abilene network topology and plot the reconfiguration cost in Figure 4.19c. Compared with the reconfiguration cost of 500 for random traffic, realistic traffic only triggers a cost of 20 on average. The two bars for each K represent low and high weights for reconfiguration cost, respectively. The results confirm our previous conclusion: with high weighting factors, our proposed algorithm can push down the value of the particular cost component.

To conclude, we formulate multi-period DCPP as a cost optimization problem considering both operational and adaptation cost. Leveraging look-ahead control, RHC and FHC algorithm show their advantage of a fair trade-off between two types of costs. Compared with the state-of-the-art algorithms, they can reduce the flow setup up time by 30% and the cost of control plane adaptation by 20%. When a low correlation exists inside data plane traffic, the control plane is expected to perform frequent reconfigurations to maintain a lower operational cost.

4.6 Summary and Discussion

In this chapter, we discuss the architecture of the dynamic control plane in SDN and its optimization opportunities, i.e., DCPP problems. We observe a research gap that has been overlooked so far: the control plane's ability to adapt according to varying data plane traffic (dynamic flows). The placement of controllers and assignment of switches are critical to establish flow connectivity, and without adapting to new traffic distributions, static controller placement can add significant overhead of flow setup time. With mathematical programming, we propose single-period DCPP which concentrates on minimizing the average flow setup time of all new flows within current flow profile. DCPP is based on an exact modeling of end-to-end flow setup latency due to flow setup requests, flow installation, and controller processing. Furthermore, we make a first move in the spirit of data-driven network optimization: we leverage ML techniques and feed historic algorithm data into classifiers that can make predictions for future similar problem instances.

We then extend the study of control placement to a scenario with multiple time-slots: controller placement adapting from one time-slot to another time-slot incurs adaptation latency. Such latency is particularly important while evaluating flexibility of the dynamic control plane, where a long adaptation is deemed as an inflexible move because network instability during adaptation can lead to service interruptions or network outages. We modeled it as a cost factor, comprising of controller migration and switch reassignment, together with another cost factor, i.e., average flow setup time, that is derived from the single-period scenario. We formally propose multi-period DCPP with the objective of minimizing the weighted sum of both cost factors of control plane management over multiple time-slots. Because of the problem's intractability nature, we leverage the scheme of look-ahead control and propose efficient online algorithms based on SA. Comprehensive evaluations demonstrate the effectiveness of cost minimization within an acceptable optimization time. The proposed optimization model and algorithms enable network operators to evaluate the trade-offs between operational and adaptation cost. We also notice that control plane adaptation is inevitable when data plane traffic has a tendency to change, e.g., diurnal patterns of shifting of traffic curves every couple of hours, which has become a common property for modern networks.

In the next chapter, we will continue with the dynamic control plane use case. Based on the "flexibility as a measure" introduced in Chapter 3, we will study how to evaluate and optimize the control plane's flexibility.

Chapter 5

Towards flexible SDN control plane

5.1 Introduction and Motivation

5.1.1 Motivation, Problem Scope, and Research Challenges

In Chapter 4, we model the adaptation cost as the difference between two consecutive control plane states, i.e., the difference of controller location and switch assignment variables. Taking the adaptation cost into account, the multi-period DCPP model optimizes the control plane states in a way that the joint objective function is minimized. However, another essential aspect in the definition of flexibility, i.e., adaptation time, is not explicitly considered. A flexible networking system envisions, the time it takes to update, e.g., the configuration of flows and functions, should be smaller than a use case-specific threshold. State-of-the-art research concentrates more on the ability to adapt, neglecting the analysis of the adaptation in terms of time. Therefore, we start by addressing this issue in this chapter: based on a discussion of the control plane adaptation techniques, we evaluate the time of controller migration and switch reassignment. The evaluation of adaptation time is the primary building block of the quantification framework that we present afterward to compare different design choices of the control plane. In particular, we refer to the flexibility metric that is introduced in Chapter 3 that takes the time and cost into account for successful migrations to handle traffic changes. By comparing the numeric values, we can reach several conclusions about the flexibility of specific control plane settings, i.e., the number of controllers, as well as the savings of cost.

Further, we try to answer another intriguing research question – *can we optimize for flexibility?* – which to our best knowledge has seldom been addressed in the literature. The optimization is critical while planning the infrastructure because it is not realistic to update the planning decisions every now and then, e.g., deployment of DCs and interconnection of substrate nodes. In comparison, the operation decisions, e.g., flow routing and function placement, can be updated during operation without obvious restrictions, and actually, this is the enabler of flexibility. In our use case, network operators have the freedom to decide where to deploy a limited number of DCs, whose locations are fixed throughout the operation of the control plane. This problem sounds trivial: a naive approach could place DCs close to large cities where most of the traffic originate and terminate. However, in a multi-controller scenario, inter-domain flows would also trigger flow setup requests at intermediate switches [7], [91] that are not necessarily in the proximity of large cities, not to mention the flows variate from time to time. In this regard, we propose a mathematical programming model called FLEXDC that maximizes the flexibility of the control plane under the constraints of adaptation time and cost. To improve efficiency, we also design two heuristics: one borrows the idea of the static controller placement problem, and the other

extracts traffic features with smaller problem input. Evaluation over real network topologies suggests that the *(i)* flexibility depends on the adaptation time and cost constraints that need to be met, and *(ii)* our proposed heuristics can achieve near-optimal performance in some cases with apparent runtime saving.

5.1.2 Key Contributions

The contributions presented in this chapter are based on our work in [8] and [13], and can be summarized as the following:

1. An evaluation procedure is introduced to calculate the adaptation time of the dynamic control plane, which comprises the latency of controller migration and switch reassignment [8].
2. A quantification framework is presented to compare different design choices in terms of flexibility offered by the dynamic control plane [8]. The design choices comprise the number of controller instances, the selection of migration time constraint, and the network topology.
3. Given the observation that the flexibility of the dynamic control plane depends heavily on the DC locations that are fixed, a mathematical programming model called FLEXDC is proposed to optimize their locations and increase the flexibility [13].

The rest of the chapter is organized as follows. Section 5.2 introduces the background of this chapter. In Section 5.3, the quantification framework is elaborated with a use case study of DCPP. Further, Section 5.4 presents the mathematical model to optimize the flexibility of the dynamic control plane. Finally, Section 5.5 concludes this chapter.

5.2 Background

The section introduces background on technique details to migrate controllers and reassign switches at runtime, which are two enablers of control plane adaptation. The subsequent quantification framework and optimization model are based on these techniques.

5.2.1 Controller Migration

The SDN controller is a software running with provisioned virtual CPU, memory, network bandwidth, and system cache in the commodity server. Different levels of virtualization enable options of VM or container, and both offer appropriate techniques for migration. Further, controller migration can be realized by leveraging the scalability feature of the controller platform itself.

5.2.1.1 VM Migration

VM migration exploits live or non-live patterns to reallocate a complete virtual machine across physical servers to successfully gain benefits of load balancing, power efficiency, fault tolerance, and system maintenance [127]. Live migration guarantees continuous service provisioning to the hosted software during the VM memory transfer process, which satisfies the requirement of the control plane. *Pre-copy* method copies complete memory footprint before resuming VM in the target server, whereas *post-copy* captures and transfers a minimum state, such as CPU registers and I/O states, to the target server before resuming the VM.

Service downtime is the most important metric to consider for the sake of the control plane's availability and responsiveness; however, a short period of downtime during which the service experiences a longer delay is inevitable. Such downtime is caused mainly by *(i)* unplugging and plugging virtual interfaces and storage and *(ii)* the update of forwarding rules in the underlying network infrastructure. Measurement studies demonstrate the service downtime that ranges between tens and hundreds of milliseconds in a server cluster [128]. Therefore, the inclusion of dedicated communication links for data transfers can reduce the service degradation experienced by the control plane [127].

5.2.1.2 Container Migration

Container techniques offer light-weight virtualization and isolation of applications sharing the same Linux kernel space. As one popular representative, Docker [200] enables layered storage inside containers, which allows fast packaging and migration based on common layers and Linux CRIU (Checkpoint/Restore in Userspace) [129]. Container images are available in a centralized image database. Before migration, the target server fetches the container base image from the database. Afterward, the thin container layer on top of the base image, together with the runtime memory footprints are transferred during the migration. Compared with VM, container migration has smaller data transfer, thus creates less burden on the network infrastructure. However, current CRIU is still limited, and it can induce downtime up to seconds [129].

5.2.1.3 State Synchronization

The third option leverages the scalability feature of controller platforms, e.g., ONOS [35] and ODL [130], to achieve migration. The distributed control plane architecture manages state synchronization between the controller instances for the sake of global view maintenance and consistent network updates. The most popular technique in this regard is *distributed key-value store* [131], which keeps information such as topology composed of devices and end-hosts, their capabilities, link latency, and bandwidth, as well as current network configuration and state in terms of flow table rules and counters. Scalability allows dynamically adding or removing controller instances in accordance with the performance and availability requirements. Before migration, the control plane adds the target server, in which a new controller instance is activated, into the cluster. The distributed database is then automatically synchronized to the new controller instance. Finally, the controller instance in the source server can be disabled.

5.2.2 Switch Reassignment

Switch reassignment techniques were originally proposed to balance flow setup requests among multiple controller instances [114], [115]. The reassignment requires proper coordination between the source and target controllers and the switch itself. To model the involved latency, we adopt the mechanism proposed by Dixit et al. [84], which makes use of different switch roles in OpenFlow. An OpenFlow-enabled switch is typically connected to one *master* controller, which has write permission of the flow table rules, and several *slave* controllers, which can only read table rules and statistics. OpenFlow also allows multiple controllers in *equal* mode and share write permission in the same switch. Figure 5.1 depicts the mechanism in four phases. In Phase 1, the target controller requests to be in the equal mode to enable its contact with the switch and prepares for migration. In Phase 2, the source controller adds a pre-defined flow rule in the switch via Flow-mod message, followed by a Barrier-request to make sure that all pending events (e.g., control plane packets) are processed by the switch. Afterward, the source

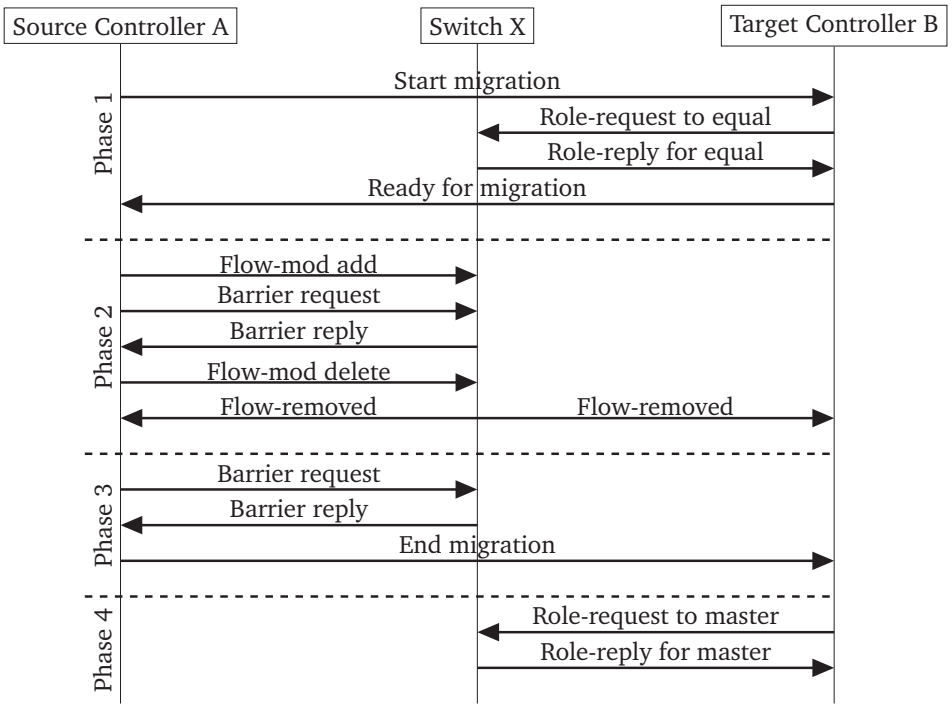


Figure 5.1: Message exchanges for switch reassignment in the dynamic control plane: switch X is reassigned from controller A to controller B. The figure is adapted from [84].

controller removes the previously added flow rule, and the switch confirms the removal by informing both controllers, which labels the migration of the switch to the target controller. In Phase 3, one more Barrier-request is pushed to the switch from the source controller to ensure that the pending events that happen in Phase 2 are processed. Upon receiving the confirmation, the source controller notifies the target controller about the end of the migration. Finally, Phase 4 requests to take over the controller completely by asking to be the master of the switch. This mechanism ensures that no control plane packets get lost or duplicated during the migration process.

5.3 Quantifying the Flexibility of the Control Plane

The adaptation of the dynamic control plane includes the migration of controllers, i.e., the change of their placement, as well as the change of switch to controller assignments. The two components significantly affect the control plane's performance, including its flexibility. Based on the technique enablers introduced in the previous section, this section takes the overall adaptation time into account that is needed for successful adaptations to handle traffic changes. It also studies the impact of adaptation time threshold on different control plane parameters.

5.3.1 Workflow Proposal

Figure 5.2 illustrates the workflow of the flexibility quantification framework for the dynamic control plane. To generate demand changes, the traffic generator randomly creates a number of flow profiles $\mathcal{F}^{(t)}, t \in T$ following a distribution and some parameters, e.g., flow density $\rho_{\mathcal{F}}$. Each flow profile represents a single-period DCPP instance and needs to be optimized towards the operational objective e.g., average flow setup time L_{avg}. After the optimization of all flow profiles is ready, flexibility will be

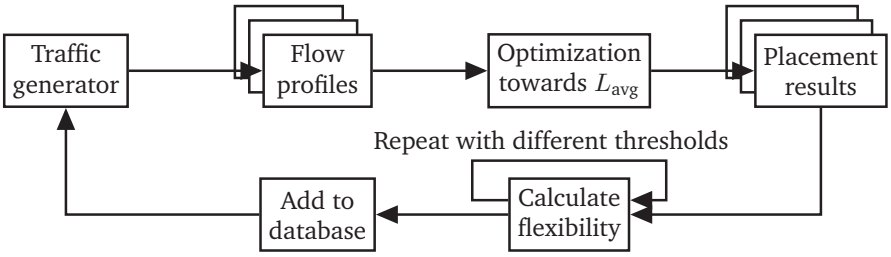


Figure 5.2: Workflow of flexibility quantification framework. Traffic generator generates a series of flow profiles, which are input to the optimization model (introduced in Section 4.4) to get the optimal placement results. A flexibility value is calculated with one adaptation time threshold. The calculation is repeated with varying thresholds to derive patterns.

Algorithm 7 Total Adaptation Time Calculation

Input: last controller set Φ_{old}, current controller set Φ_{new}, switch set $\mathcal{V}$, size of data store in controller σ, link bandwidth η, shortest path latency $\ell(u, v)$
Output: Total adaptation time T_{at}

set of propagation latencies of the controllers to be migrated $\mathcal{T}_{\text{ctr\_prop}} = \emptyset$
$T_{\text{ctr\_prop}} = 0$, $T_{\text{ctr\_tran}} = 0$
for each c_{new} in C_{new} **do**
  if $c_{\text{new}} \notin C_{\text{old}}$ **then**
    find the $c_{\text{old}} \in C_{\text{old}}$ such that the latency $\ell(c_{\text{old}}, c_{\text{new}})$ is minimal.
    insert $\ell(c_{\text{old}}, c_{\text{new}})$ into $\mathcal{T}_{\text{ctr\_prop}}$
  end if
end for
$T_{\text{ctr\_prop}} = \max(\mathcal{T}_{\text{ctr\_prop}})$ ▷ Parallel control migrations
$T_{\text{ctr\_tran}} = \sigma/\eta$
set of reassignment times of switches $\mathcal{T}_{\text{sw}} = \emptyset$
for each $v \in V$ **do**
  if v is assigned to a different controller instance **then**
    get the master controller c of sw
    insert $\ell(v, c)$ into $\mathcal{T}_{\text{sw}}$
  end if
end for
$T_{\text{sw}} = \max(\mathcal{T}_{\text{sw}})$ ▷ Parallel switch reassignments
$T_{\text{at}} = T_{\text{ctr\_prop}} + T_{\text{ctr\_tran}} + T_{\text{sw}}$

calculated, given the order that the profiles are generated. The controller placement result of the first flow profile $\mathcal{F}^{(0)}$ becomes the initial control plane state. Starting from the second profile, $\mathcal{F}^{(1)}$, the new optimal control plane state is compared with the old one; if there is a difference in terms of controller locations or switch assignment, the adaptation is triggered and the corresponding time is calculated.

Algorithm 7 describes the calculation process consisting of controller migration and switch reassignment. In SD-WAN, the latency of controller migration depends on two factors involved in synchronizing the internal states of the data plane: propagation and transmission. Whereas the transmission latency is relatively fixed, the propagation latency is proportional to the geographical difference between the source and target node. This thesis assumes the adaption scheme as follows. For each new controller location (i.e., target node), it examines all old locations and selects the one with the least propagation latency as the source node; the corresponding propagation latency is inserted

Algorithm 8 Flexibility Measure Calculation

Input: adaptation time threshold $\mathcal{C}_{\text{at}}$
Output: flexibility measure $\varphi(\mathcal{A}(T,\infty))$
1: randomly create an order of all flow profiles
2: total number of migration requests $n_{\text{req}} = 0$
3: total number of migration success $n_{\text{suc}} = 0$
4: **for** each flow profile in the order **do**
5: $n_{\text{req}} = n_{\text{req}} + 1$
6: get optimal controller set C_{new} and switch assignment set A_{new}
7: **if** this is the first flow profile **then**
8: $C_{\text{old}} = C_{\text{new}}$, $A_{\text{old}} = A_{\text{new}}$
9: continue
10: **end if**
11: **if** $C_{\text{new}} = C_{\text{old}}$ and $A_{\text{new}} = A_{\text{old}}$ **then**
12: $n_{\text{suc}} = n_{\text{suc}} + 1$
13: **else**
14: call Algorithm 7 to calculate T_{at}
15: **if** $T_{\text{at}} < \mathcal{C}_{\text{at}}$ **then**
16: $n_{\text{suc}} = n_{\text{suc}} + 1$
17: $C_{\text{old}} = C_{\text{new}}$, $A_{\text{old}} = A_{\text{new}}$
18: **end if**
19: **end if**
20: **end for**
21: $\varphi(\mathcal{A}(T,\infty)) = n_{\text{suc}}/n_{\text{req}}$

into a set $\mathcal{T}_{\text{ctr\_prop}}$ (Line $3-8$). Afterwards, $T_{\text{ctr\_prop}}$ is the maximum of $\mathcal{T}_{\text{ctr\_prop}}$. The transmission latency $T_{\text{ctr\_tran}}$ is calculated as the ratio of data store size σ and link bandwidth reserved for migration η. The number of active controllers does not impact σ. Regarding switch reassignment, this thesis assumes applying the protocol in Section 5.2.2 which induces six Round-Trip Times (RTTs) between the switch and its new controller. For each switch that need to be re-assigned, the propagation latency between it and its new controller is calculated and inserted into a set $\mathcal{T}_{\text{sw}}$ (Line $12-17$). Switch reassignments can take place in parallel, therefore the maximum of $\mathcal{T}_{\text{sw}}$ is passed to T_{sw}. Finally, the overall adaptation time T_{at} is the sum of $T_{\text{ctr\_prop}}$, $T_{\text{ctr\_tran}}$, and T_{sw} (Line 19).

Recall the interpretation and the measure of flexibility in Section 3. It represents the *timely support of changes in the network requirements* and is defined as follows:

$$\varphi(\mathcal{A}(T,C)) = \frac{\#\text{ achievable demand changes under } T \text{ and } C}{\#\text{ defined demand changes}}, \tag{5.1}$$

where T and C represent the time and cost constraint of adaptation. We assume the adaptation cost is unbounded, i.e., $C = \infty$, and predefine a value $\mathcal{C}_{\text{at}}$ for the time threshold. For each new demand, the overall adaptation time T_{at} is compared against $\mathcal{C}_{\text{at}}$: the current adaptation is successful only if $T_{\text{at}} \leq \mathcal{C}_{\text{at}}$. Notably, $T_{\text{at}} = 0$ means that the control plane automatically supports the new demand change without adaptation. Following a successful adaptation, the control plane state changes to the new optimum; otherwise, it stays as the old one, i.e., $S^{(i-1)}$. A detailed description of the calculation is shown in Algorithm 8.

Table 5.1: Simulation parameter settings to evaluate the flexibility of the dynamic control plane.

Parameters	Values
Number of controllers	1, 2, 3, 4
Number of flow S-D pairs	6 (Abilene), 14 (Germany)
Flow number distribution	$\ln\mathcal{N}$ with mean of 20
Total data store size of a controller σ	100 Mega Byte
Migration Link Bandwidth η	10 Gbps
# RTT of re-assigning a switch	6
Number of demand changes per run	100
Number of runs	50

5.3.2 Results Evaluation and Analysis

This section investigates the impact of a control plane design choice, namely the number of controllers, on the flexibility metric for different control plane topologies.

5.3.2.1 Evaluation Settings

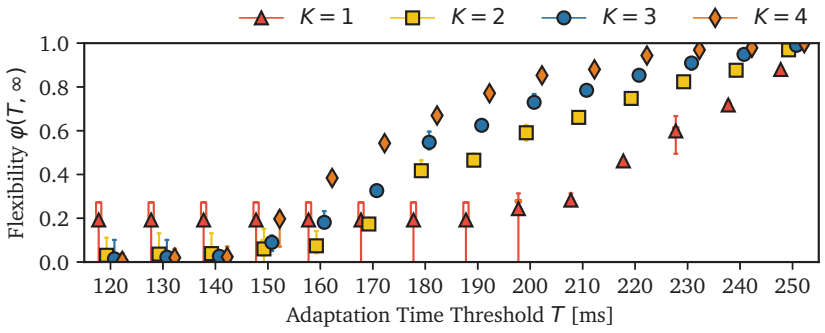
We evaluate two network topologies: Abilene (from Topology Zoo [116]) and Germany (from the SNDLib topology database [132]). To produce the demands, we first generate 100 flow profiles with the same traffic model as in Section 4.4. For each flow profile, we optimize the single-period DCPP model (introduced in Section 4.4) for a different number of controllers. The 100 flow profiles are shuffled to represent dynamics further and then fed into Algorithm 8 with the adaptation time threshold. To convey the relation between $\mathcal{C}_{\text{at}}$ and our flexibility measure, we choose $\mathcal{C}_{\text{at}}$ in a way that it covers the range from almost no demand to nearly all demands can be supported, which is $[120, 250]$ for Abilene and $[85, 105]$ for Germany. Detailed parameter settings are listed in Table 5.1.

5.3.2.2 Results

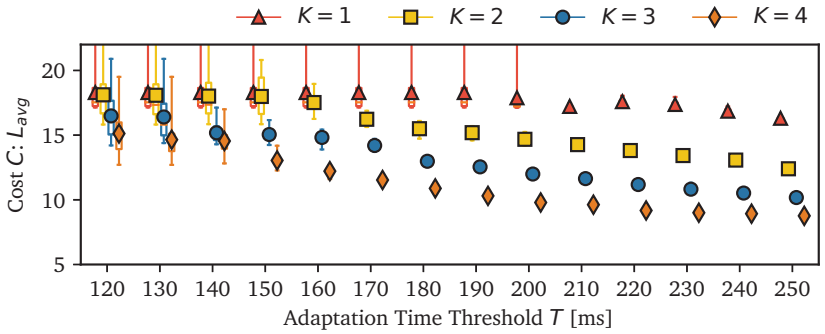
Next, we demonstrate and analyze the evaluation results from different perspectives.

Comparison of flexibility. Evaluation results of Abilene and Germany are shown as box-plots in Figure 5.3a and Figure 5.4a respectively. We can observe that for both network topologies, the flexibility increases as the adaptation time threshold $\mathcal{C}_{\text{at}}$ increases because of a larger $\mathcal{C}_{\text{at}}$ guarantees more successful migrations. More controllers lead to less inter-controller distance and less control latency of switch, which also provides higher flexibility. When $\mathcal{C}_{\text{at}}$ is small (less than 170 milliseconds for Abilene and 95 milliseconds for Germany), the flexibility of one controller $K = 1$ is higher than that of all other cases. This is because the single controller tends to stay in one optimal location, even for different flow profiles, which requires no migration time. In other words, a flexible system does not always need to adapt its internal state to support new demands, as long as the cost and time constraints are satisfied.

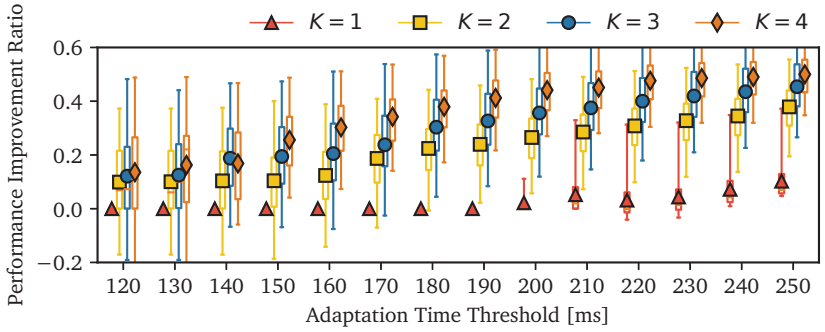
Comparison of cost aspect. We use the average flow setup time L_{avg} as an indicator for the operational cost and plot the comparison in Figure 5.3b and Figure 5.4b. In general, higher cost emerges if the adaptation time threshold $\mathcal{C}_{\text{at}}$ cannot be fulfilled by the adaptation process, where the new optimal placement will be discarded, and the average flow setup time will be degraded. The distribution of data



(a) Flexibility.



(b) Cost.

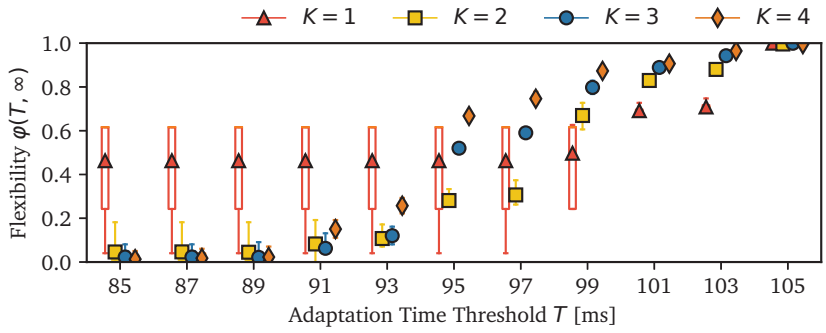


(c) Performance improvement.

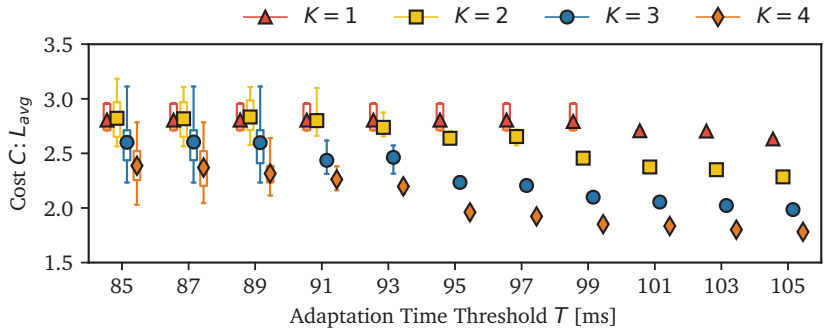
Figure 5.3: Flexibility measure, cost and performance improvement of Abilene network topology comparing different number of controllers and different $\mathcal{C}_{\text{at}}$. For small $\mathcal{C}_{\text{at}} < 150$, $K = 1$ is more flexible than the other cases.

is more condensed when $\mathcal{C}_{\text{at}}$ increases. In general, the cost decreases as the control plane becomes more flexible. Given a single controller, a large $\mathcal{C}_{\text{at}}$ generates a marginal reduction in cost. When $\mathcal{C}_{\text{at}}$ is small, L_{avg} of different number of controllers in most simulation runs are close. $K = 1$ even outperforms $K = 2$ in some cases, because under a stringent $\mathcal{C}_{\text{at}}$, 2 controllers may remain in a placement for the following flow profiles that degrade control latencies. However, as $\mathcal{C}_{\text{at}}$ increases, the average flow setup time of $K = 4$ outperforms the other three cases in every run. Comparing with $K = 1$, $K = 4$ drops from 15 milliseconds to less than 10 milliseconds, which is a 30% decrease in cost.

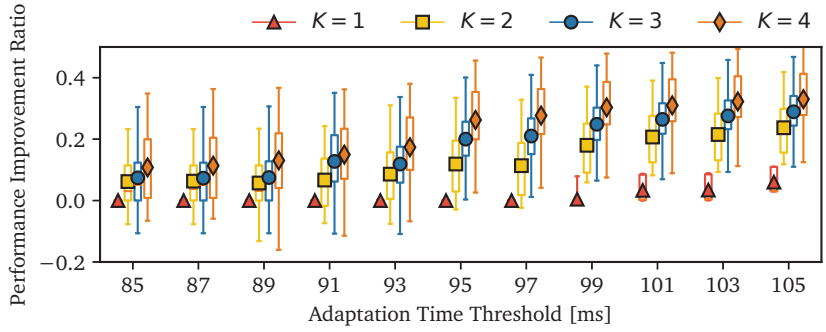
Benefit of the dynamic control plane. Another observation helps to answer the question of why a system should be flexible and adapt to dynamic inputs. We compare the average flow setup time of the dynamic control plane, which can adapt but with a time constraint, with that of the static control plane. The comparisons are shown in Figure 5.3c and Figure 5.4c. For the static scenario, we keep the optimal controller placement from the first flow profile for all subsequent flow profiles. It is clear that the more



(a) Flexibility.



(b) Cost.



(c) Performance improvement.

Figure 5.4: Flexibility measure, cost and performance improvement of Germany network topology comparing different number of controllers and different $\mathcal{C}_{\text{at}}$. For small $\mathcal{C}_{\text{at}} < 95$, $K = 1$ is more flexible than the other cases.

controllers are deployed, the stronger need for the system's adaptation with respect to changing traffic input. The improvement in performance from being dynamic can be as significant as 60% for Abilene and 40% for Germany. Note that the static placement can outperform the dynamic placement that could not adapt under strict migration time $\mathcal{C}_{\text{at}}$. This happens if the initial static placement is by chance better than the placement solutions that the dynamic placement remains at while not being able to adapt to the demands.

Every millisecond counts. Comparing the flexibility of two different network topologies, we notice the difference between the range of the adaptation time threshold. The geographical coverage of Germany is smaller than Abilene, which makes the propagation delay contributes less than the transmission delay to the overall migration time ($T_{\text{ctr\_tran}} \leq 25$ milliseconds and $T_{\text{ctr\_tran}} = 80$ milliseconds). Because of this, φ of $K = 4$ drops from higher than 0.9 to lower than 0.2 while $\mathcal{C}_{\text{at}}$ is relaxed for 10 milliseconds.

Therefore, it becomes more crucial to reduce the amount of data while migrating a controller and shrink the transmission delay to satisfy more demands under a tight time constraint.

To conclude, the above observations show that $K = 1$ provides more flexibility when the migration time $\mathcal{C}_{\text{at}}$ is small. It proposes not only higher flexibility, but also a satisfying performance in many cases. However, more controllers outperform the single controller given a higher $\mathcal{C}_{\text{at}}$, as they show a more flexible control plane as well as an improved flow setup performance.

5.4 Optimizing the Flexibility of the Control Plane

After evaluating the flexibility of the control plane, this section moves one step further and sheds light on the optimization towards flexibility. The optimization takes place during the *design phase* in the life-cycle of a flexible networking system. In this phase, the design parameters, e.g., static facility locations and physical link connections, are decided by network designers and stay fixed during the operation of the networking system. A common approach to determine such parameters is to consider the usual metrics of networks such as latency and reliability. However, the interest for flexibility also brings in a new metric that we can consider.

In this regard, this section continues the study of the dynamic control plane and considers a realistic condition, i.e., limited DCs locations. DCs hosts the controllers and enables their migrations during operation. However, almost all related works assume that each node of the network topology has a DC installed, which is far from realistic. Indeed, the number of DCs is limited, and in the *design phase*, it is critical to decide the number of DCs and their locations that can facilitate the control plane to achieve higher flexibility. Accordingly, this section introduces a mathematical programming model called FLEXDC. On top of the time aspect, this model incorporates flow setup performance as the cost aspect. The flexible control plane is assumed to reduce the cost factors by adapting itself under an adaptation time constraint. Two heuristic methods are proposed to decide the DC locations efficiently. A comprehensive evaluation is performed on real network topologies to analyze the impact of cost and time constraints as well as the performance of proposed heuristic methods.

5.4.1 Problem Formulation

This section introduces the decision variables, constraints, and objective function to build the optimization problem FLEXDC. In addition to the single- and multi-period DCPP, the FLEXDC model needs to additionally consider the static DC placement over the whole planning horizon.

5.4.1.1 Problem Input and Variables

We consider that all nodes $v, \forall v \in \mathcal{V}$ are candidates for DCs, i.e., $\mathcal{D} \subseteq \mathcal{V}$. The controllers can only be placed on the nodes where DCs locate. The set of possible controller locations is always a subset of the set of possible DC locations, i.e., $D \subseteq C$. Note that for the sake of simplicity, the set of possible DC locations D is the same as the set of nodes V, but in reality, the two sets can be different. We have three parameters to reveal our thresholds of flow setup performance ($\mathcal{C}_{\text{cl}}$), controller's node ($\mathcal{C}_{\text{l}}$), and latency of control plane adaptation ($\mathcal{C}_{\text{at}}$). The additional notations defined in this section are summarized in Table 5.2. The new variables are described in Table 5.3. In each time-slot t, three binary variables $\mathcal{B}_{\text{cl}}^{(t)}$, $\mathcal{B}_{\text{l}}^{(t)}$, and $\mathcal{B}_{\text{at}}^{(t)}$ denote the fulfillment of the cost and adaptation time constraints.

Table 5.2: Additional notations of FLEXDC.

Notation	Description
$\mathcal{D}$	Set of DCs where $\mathcal{D} \subseteq \mathcal{V}$
K_{DC}	Number of DCs with $K_{\text{dc}} \geq K$
$\mathcal{T}^t_{\text{sw\_assign}}$	Non-negative variable representing the switch reassignment delay in time-slot t
$\mathcal{C}_{\text{at}}$	Threshold of adaptation time constraint
$\mathcal{C}_{\text{cl}}$	Threshold of flow setup constraint
$\mathcal{C}_{\text{l}}$	Threshold of controller's load constraint

5.4.1.2 Constraints

The new constraints of OPEFLEX model the additional layer of DCs between the nodes and the controllers. This set of constraints constitute the basic model of placing controllers and DCs, as well as assigning switches to the controllers in each time-slot t.

Number of DCs. We ensure that the number of DCs (i.e., the number of selected DC nodes) is equal to K_{DC} throughout all time-slots T:

$$\sum_{d \in \mathcal{D}} p_{\mathcal{D}}(d) = K_{\text{DC}}. \tag{5.2}$$

Number of controllers. Each DC $d \in \mathcal{D}$ that is fixed on node $d \in \mathcal{V}$ can host only one controller in each time-slot $t \in T$:

$$\sum_{c \in \mathcal{C}} p^{(t)}_{\mathcal{C}}(c) = K. \tag{5.3}$$

Placing controllers on DC node. In each time-slot $t \in T$, each controller $c \in \mathcal{C}$ must be placed on a node where there is a DC. In other words, controllers can only move among fixed DC. Thus,

$$p^{(t)}_{\mathcal{C}}(c) \leq p_{\mathcal{D}}(d), \quad \forall c \in \mathcal{C}, \forall d \in \mathcal{D}, \forall t \in T. \tag{5.4}$$

Controller locations. The helper variable $pp^{(t)}_{\mathcal{C}}(c_{\text{old}}, c)$ will be set to 1 if there is one controller at c_{old} in time-slot $t-1$ and one controller at c in time-slot t. Therefore,

$$pp^{(t)}_{\mathcal{C}}(c_{\text{old}}, c) = p^{(t-1)}_{\mathcal{C}}(c_{\text{old}}) \times p^{(t)}_{\mathcal{C}}(c), \quad \forall c_{\text{old}}, c \in C,\ t \in T. \tag{5.5}$$

Different from the multi-period DCPP model of Section 4.5, FLEXDC does not specify the cost and time aspects as optimization objectives. Instead, they are translated to respective constraints of fulfillment, and the degree of fulfillment (i.e., the number of fulfilled constraints) is turned into the optimization objective.

Flow setup performance. The performance indicator is the *average end-to-end flow setup time* $C^{(t)}_F$ introduced in Eq. (4.17). $C^{(t)}_F$ represents the time it takes to forward the first packet of a new flow to its destination, which incurs one or more flow setups along the forwarding path. It is a critical because of the additional communication and processing latency of each new flow. FLEXDC simplifies $C^{(t)}_F$ as the average control latency of each node weighted by the number of new flows originate from it, defined as

Table 5.3: Variables of FLEXDC.

Notation	Description
$p_{\mathcal{D}}(d)$	Binary variable representing if a DC is placed on the location $d \in D$
$p_{\mathcal{C}}^{(t)}(c)$	Binary variable representing if a controller is placed on the location $c \in \mathcal{C}$ in time-slot t
$a_{\mathcal{V},\mathcal{C}}^{(t)}(v,c)$	Binary variable representing if a switch $v \in \mathcal{V}$ is assigned to a different controller other than $c \in \mathcal{C}$ in time-slot $t-1$ and assigned to c in time slot t
$l_{\mathcal{V}}(v)$	non-negative variable representing the control path latency of a switch $v \in \mathcal{V}$
$pp_{\mathcal{C}}^{(t)}(c_{\text{old}},c)$	Binary variable representing the location of two controllers in two consecutive time-slots
$\mathcal{L}_{\text{prop}}^{(t)}(c)$	Non-negative variable representing the propagation delay of migrating a controller $c \in C$
$T_{\text{tran}}^{(t)}$	Non-negative variable representing the transmission latency of controller migration in time-slot t
$T_{\text{prop}}^{(t)}$	Non-negative variable representing the propagation latency of controller migration in time-slot t
$p_{\mathcal{C},\text{SOS}}^{(t)}(c_{\text{old}},c)$	Binary SOS variable representing if a migration happens between two locations c_{old} and c
$\mathcal{R}_{\text{sw}}^{v,c,t}$	Binary variable representing if the assigned controller $c \in C$ of node $v \in V$ in time-slot t is different from the controller in time-slot $t-1$
$T_{\text{sw}}^{(t)}$	Non-negative variable representing the time of switch reassignment in time-slot t
$T_{\text{at}}^{(t)}$	Non-negative variable representing the overall adaptation time in time-slot t
$T_{\text{cl}}^{(t)}$	Non-negative variable representing the average weighted control latency in time-slot t
$L_{\text{l}}^{(t)}(c)$	Non-negative variable representing the amount of flow setup requests served by a controller $c \in \mathcal{C}$
$\mathcal{B}_{\text{at}}^{(t)}$	Binary variable representing if the adaptation time constraint is fulfilled with $T_{\text{at}}^{(t)} \leq \mathcal{C}_{\text{at}}$
$\mathcal{B}_{\text{cl}}^{(t)}$	Binary variable representing if the flow setup constraint is fulfilled with $T_{\text{cl}}^{(t)} \leq \mathcal{C}_{\text{cl}}$
$\mathcal{B}_{\text{l}}^{(t)}$	Binary variable representing if the load of controller constraint is fulfilled with $L_{\text{l}}^{(t)}(c) \leq \mathcal{C}_{\text{l}}, \forall c \in \mathcal{C}$

follows:

$$T_{\text{cl}}^{(t)} = \frac{1}{|F_t|} \sum_{f \in \mathcal{F}^{(t)}} l_{\mathcal{V}}(f[s]), \quad \forall t \in T. \tag{5.6}$$

Load of controllers. Following the queuing theory, the expect sojourn time of flow setup request from a switch would increase and approach infinity, when the arrival rate of setup requests approximates the controller's capacity. A new flow profile with different traffic distribution can overload a certain controller if the previous controller placement retains. Therefore, it incurs a high cost in terms of the consolidated load of controller, which is defined as the total number of flow setup requests, including both initial and intermediate ones, that the controller needs to process in time-slot t. Thus,

$$L_1^{(t)}(c) = \sum_{f \in \mathcal{F}^{(t)}} a_{\mathcal{V},\mathcal{C}}^{(t)}(f[s], c) + \sum_{f \in \mathcal{F}^{(t)}} \sum_{(u,v) \in f} a_{\mathcal{V},\mathcal{C}}^{(t)}(v, c) \times \overline{d}_{\mathcal{V}}(u, v), \quad \forall t \in T, \ \forall c \in C. \tag{5.7}$$

Similarly, it can be simplified to only count the load in terms of the initial flow setup requests, as followed,

$$L_1^{(t)}(c) = \sum_{f \in \mathcal{F}^{(t)}} a_{\mathcal{V},\mathcal{C}}^{(t)}(f[s], c), \quad \forall t \in T, \ \forall c \in C. \tag{5.8}$$

Controller migration mapping. Following the migration pattern in Section 5.3, each controller in time-slot t selects the closest location of a controller in time-slot $t-1$ and produces its own propagation delay. To model this selection procedure, we need to find the one-to-one mapping of the controllers of the previous time-slot $t-1$ and current time-slot t. We leverage Spatial Ordered Set (SOS) variables in this regard. In a set of SOS variables, at most *one* item can take a non-zero value, whereas the remaining items are set to 0. We use binary SOS variable $p_{\mathcal{C},\text{SOS}}^{(t)}(c_{\text{old}}, c)$ to denote if a migration happens between old c_{old} and new controller location c. The following constraint enforces only one migration between a pair of old and new locations:

$$\sum_{c_{\text{old}} \in \mathcal{C}} p_{\mathcal{C},\text{SOS}}^{(t)}(c_{\text{old}}, c) = 1, \quad \forall c \in C, \ t \in T. \tag{5.9}$$

For example, $p_{\mathcal{C},\text{SOS}}^{(3)}(1, 5) = 1$ means that the controller on node 5 in time-slot 3 is migrated from node 1 in time-slot 2. In the meantime, $p_{\mathcal{C},\text{SOS}}^{(3)}(c_{\text{old}}, 5) = 0, \forall c_{\text{old}} \in \mathcal{C} \setminus \{1\}$, which forbids the migration from old locations other than 1.

The transmission delay $T_{\text{tran}}^{(t)}$ is defined as the time of transmitting the VM or the data store of a controller with the migration bandwidth. It's obvious that $\mathcal{T}_{\text{ctr\_tran}}$ does not dependent on the controller placement and is always a constant, therefore it can be ignored in Eq. (5.15).

Propagation delay of a controller migration. If a controller migrates from c_{old} to c, the propagation delay $\mathcal{L}_{\text{prop}}^{(t)}(c)$ is the shortest-path latency between c_{old} and c, i.e., $\ell(c_{\text{old}}, c)$, otherwise it is set to a large number $\mathbb{M}$ (meaning an illegal case). Therefore,

$$\mathcal{L}_{\text{prop}}^{(t)}(c) = \sum_{c_{\text{old}} \in \mathcal{C}} p_{\mathcal{C},\text{SOS}}^{(t)}(c_{\text{old}}, c) \times \Big[pp_{\mathcal{C}}^{(t)}(c_{\text{old}}, c) \times \ell(c_{\text{old}}, c) + (1 - pp_{\mathcal{C}}^{(t)}(c_{\text{old}}, c)) \times \mathbb{M} \Big], \quad \forall c \in C, t \in T. \tag{5.10}$$

Migration of controllers in parallel. Because the migrations of different controllers occur in parallel, only the longest migration decides the overall migration time. Therefore,

$$T_{\text{prop}}^{(t)} \geq \mathcal{L}_{\text{prop}}^{(t)}(c) \times p_{\mathcal{C}}^{(t)}(c), \quad \forall c \in C, \ \forall t \in T. \tag{5.11}$$

Note that variable $T^{(t)}_{\text{prop}}$ does not depend on any controller instance.

Mapping of switch reassignment. Helper variable $\mathcal{R}^{v,c,t}_{\text{sw}}$ represents if switch v is assigned to a different controller other than c in slot $t-1$ and assigned to c in slot t, and it is defined as the following:

$$\mathcal{R}^{v,c,t}_{\text{sw}} = a^{(t)}_{\mathcal{V},\mathcal{C}}(v,c) \times (1 - a^{(t-1)}_{\mathcal{V},\mathcal{C}}(v,c)), \quad \forall v \in V,\ \forall c \in C,\ \forall t \in T. \tag{5.12}$$

Reassignment in parallel. Switch reassignments can also occur simultaneously with the overall delay $T^{(t)}_{\text{sw}}$ defined as the maximum of all switch reassignment delays in time-slot t. Therefore,

$$T^{(t)}_{\text{sw}} \geq \sum_{c \in C} \mathcal{R}^{v,c,t}_{\text{sw}} \times \ell(v,c) \times a^{(t)}_{\mathcal{V},\mathcal{C}}(v,c), \quad \forall v \in V,\ \forall t \in T. \tag{5.13}$$

Note that $T^{(t)}_{\text{sw}}$ is defined only for valid $t-1$ and t. For the time-slot $t=0$, since $t-1$ does not exist, we have $\mathcal{T}^{0}_{\text{ctr\_prop}} = 0$. But for the switches, we define $T^{0}_{\text{sw\_assign}}$ as the maximum control latency as followed,

$$T^{(t)}_{\text{sw}} \geq \sum_{c \in C} \ell(v,c) a^{(t)}_{\mathcal{V},\mathcal{C}}(v,c), \quad \forall v \in V,\ t = 0. \tag{5.14}$$

Buildup of control plane adaptation time. Control plane adaptation triggered by new flow profile consists of *(i)* controller migration and *(ii)* switch reassignment. Switches are reassigned after controllers are migrated to their new locations. Therefore,

$$T^{(t)}_{\text{at}} = T^{(t)}_{\text{prop}} + T^{(t)}_{\text{tran}} + T^{(t)}_{\text{sw}}, \quad \forall t \in T, \tag{5.15}$$

where $T^{(t)}_{\text{prop}}$, $T^{(t)}_{\text{tran}}$, and $T^{(t)}_{\text{sw}}$ denotes the controller propagation, transmission delay, and switch reassignment delay respectively. The propagation delay of a controller migration is the latency of the shortest-path between its old (at $t-1$) and new (at t) location.

Linearization. The formulation of FLEXDC has the following non-linear constraints: Eq. (5.10), Eq. (5.11), Eq. (5.12), Eq (5.13). Similar to the single- and multi-period DCPP models, we apply linearization techniques and create auxiliary constraints to make them compatible in linear solvers.

5.4.1.3 Objective Function

Cost of Flow Setup

For the cost constraint in terms of flow setup performance, if the average flow setup time $T^{(t)}_{\text{avg}}$ (or average weighted control latency $T^{(t)}_{\text{cl}}$) is not greater than the predefined threshold $\mathcal{C}_{\text{cl}}$, we set the variable $\mathcal{B}^{(t)}_{\text{cl}}$ to 1. Therefore,

$$\mathcal{B}^{(t)}_{\text{cl}} = \begin{cases} 1, & \text{if } T^{(t)}_{\text{cl}} \leq \mathcal{C}_{\text{cl}}; \\ 0, & \text{otherwise.} \end{cases} \tag{5.16}$$

As if-condition expression cannot be directly supported by linear solvers, we linearize with the following two equations:

$$\mathcal{C}_{\text{cl}} - T^{(t)}_{\text{cl}} \leq \mathcal{B}^{(t)}_{\text{cl}} \times \mathbb{M}, \quad \forall t \in T, \tag{5.17}$$

$$\mathcal{B}^{(t)}_{\text{cl}} \times (\mathcal{C}_{\text{cl}} - T^{(t)}_{\text{cl}}) \geq 0, \quad \forall t \in T, \tag{5.18}$$

where $\mathbb{M}$ is a very large number.

Cost of Controller's Load

For the cost constraint of load of controller, the fulfillment binary variable $\mathcal{B}_{\text{l}}^{(t)}$ is set to 1 only if the load of each controller $c \in \mathcal{C}$ is not larger than the predefined threshold $\mathcal{C}_{\text{l}}$. Therefore,

$$\mathcal{B}_{\text{l}}^{(t)} = \begin{cases} 1, & \text{if } L_{\text{l}}^{(t)}(c) \leq \mathcal{C}_{\text{l}}, \quad \forall c \in \mathcal{C}; \\ 0, & \text{otherwise.} \end{cases} \tag{5.19}$$

Its linear equivalences are as follows:

$$\mathcal{C}_{\text{l}} - L_{\text{l}}^{(t)}(c) < \mathcal{B}_{\text{l}}^{(t)} \times \mathbb{M}, \quad \forall t \in T, \tag{5.20}$$

$$\mathcal{B}_{\text{l}}^{(t)} \times (\mathcal{C}_{\text{l}} - L_{\text{l}}^{(t)}(c)) \geq 0, \quad \forall t \in T, \tag{5.21}$$

where $\mathbb{M}$ is a very large number.

Time of Adaptation

For the adaptation time constraint, the fulfillment binary variable $\mathcal{B}_{\text{at}}^{(t)}$ is set to 1 only if the overall adaptation time of the control plane is not larger than the predefine threshold $\mathcal{C}_{\text{at}}$. Therefore,

$$\mathcal{B}_{\text{at}}^{(t)} = \begin{cases} 1, & \text{if } T_{\text{at}}^{(t)} \leq \mathcal{C}_{\text{at}}; \\ 0, & \text{otherwise.} \end{cases} \tag{5.22}$$

Its linear equivalences are as follows:

$$\mathcal{C}_{\text{at}} - T_{\text{at}}^{(t)} \leq \mathcal{B}_{\text{at}}^{(t)} \times \mathbb{M}, \quad \forall t \in T, \tag{5.23}$$

$$\mathcal{B}_{\text{at}}^{(t)} \times (\mathcal{C}_{\text{at}} - T_{\text{at}}^{(t)}) \geq 0, \quad \forall t \in T. \tag{5.24}$$

where $\mathbb{M}$ is a very large number.

Objective

After defining three fulfillment variables, the objective of FLEXDC is to decide the static locations of DCs (i.e., the variables $p_{\mathcal{D}}(d), \forall d \in \mathcal{D}$) so as to maximize the number of successful adaptations triggered by a set of demands. Each demand is defined as a flow profile $\mathcal{F}^{(t)}, t \in T$. A successful adaptation means $\mathcal{B}_{\text{cl}}^{(t)} = \mathcal{B}_{\text{l}}^{(t)} = \mathcal{B}_{\text{at}}^{(t)} = 1$. In other words, partial fulfillment, e.g., satisfying two cost constraints but violating the adaptation time constraint, is deemed as an adaptation failure. Thus, the objective is formally defined as follows:

$$\text{minimize} \quad \sum_{t \in T} \mathcal{B}_{\text{cl}}^{(t)} \times \mathcal{B}_{\text{l}}^{(t)} \times \mathcal{B}_{\text{at}}^{(t)}, \quad \forall t \in T,$$

Besides the locations of DCs, the control plane state, i.e., controller locations and switch assignment, in each time-slot t are also revealed in variable $p_{\mathcal{C}}^{(t)}(c)$ and $a_{\mathcal{V},\mathcal{C}}^{(t)}(v, c)$.

5.4.2 Heuristic Methods for DC Selection

The complexity of FLEXDC is enormous, which induces unbearable optimization's runtime for large-size topology and a large number of slots. Therefore, we propose three methods to decide DC locations efficiently with less knowledge of flow profiles.

5.4.2.1 DC Selector with Random Selection (RANDOMDC)

This is a straw-man method that selects DC locations in a purely random manner. It takes the number of DCs K_{DC} and the network topology G with edges E and vertices V as input. It is a baseline against other methods.

5.4.2.2 DC Selector towards Minimum Average Control Latency (ACLDC)

This heuristic method borrows the output of CPP, which optimizes the controller locations towards minimal average control latency of all switches in the network topology. The controllers' locations will be used for that of DCs. For example, if 5 DCs need to be placed, i.e., $K_{\text{DC}} = 5$, the corresponding CPP instance of 5 controllers is optimized. However, while evaluating operating the control plane, $K \leq 5$. For huge topology size, heuristic algorithms with acceptable optimization gaps can also be applied, such as the proposal in [92].

5.4.2.3 DC Selector with Half input (HFDC)

The number of time-slots $|T|$ is one factor that contributes to the complexity of FLEXDC. That means shrinking the solution space can reduce the optimization runtime. Thus, we propose another heuristic method, which only considers the first half of the time-slots, i.e., $|T|/2$. Note that this number can be amended depending on the traffic input. This method is more promising when the cyclic pattern can be observed in data plane traffic [31].

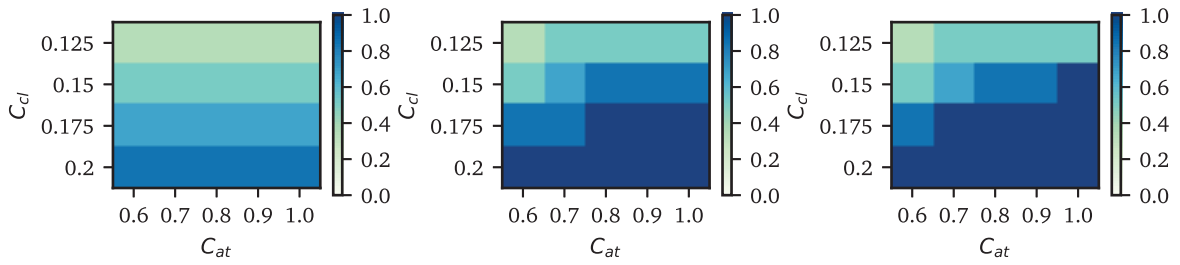
5.4.3 Results Evaluation and Analysis

This section investigates the impact of the static DC placement on the control plane's theoretical maximal flexibility. The results also show the adaptation time, the control latency, and the controller's load constraint can affect the success of control plane adaptations.

5.4.3.1 Evaluation Settings

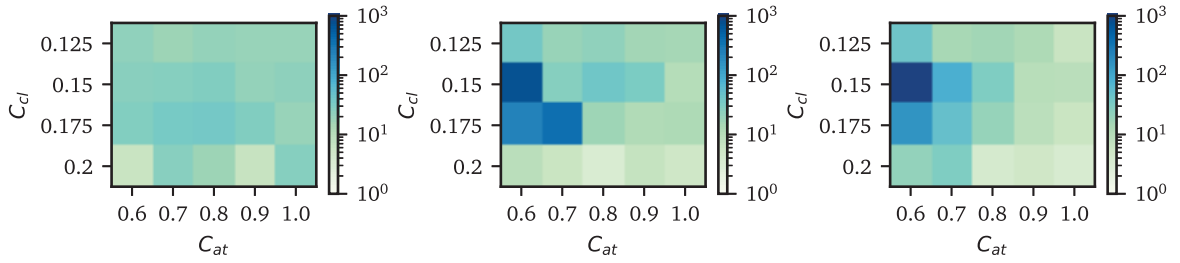
For FLEXDC, the flexibility value is calculated as the ratio between the number of adapted demand changes, i.e., the objective function value, and the number of demand changes $|T|$. However, the three proposed efficient DC placement methods (RANDOMDC, ACLDC, and HFDC) do not return the number of adapted demand changes automatically. Therefore, the following flexibility evaluation process is performed. For a set of DC locations returned by a heuristic method, we optimize the same mathematical programming model defined by FLEXDC but with DC location variables $p_{\mathcal{D}}(d)$ fixed accordingly. For example, if the set of DC locations returned by HFDC is $\{0, 3, 8\}$, we need to explicitly set $p_{\mathcal{D}}(0) = p_{\mathcal{D}}(3) = p_{\mathcal{D}}(8) = 1$ and the others to 0 before feeding the optimization problem to the linear optimizer. The controller placement and switch assignment will be optimized for each time-slot to maximize the number of adapted demand changes.

We evaluate two network topologies from the Topology Zoo [116]: Abilene and AttMpls. We set $|T| = 5$ to represent a short planning period and $|T| = 10$ to represent a long planning period. In each time-slot t, the traffic generator creates a random number of data plane flows that follows a uniform distribution $U(1, 10)$ (Abilene) and $U(1, 100)$ (AttMpls). We variate the number of DCs K_{DC} between 2 and 6, fix the number of controllers $K = 2$. To analyze the impact of different threshold combinations, i.e., adaptation time constraint, control latency, and controller's load constraint, we use



(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.5: Heatmap of flexibility with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (FLEXDC, Abilene network topology, $|T| = 5$). One figure per K_{DC}.



(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.6: Heatmap of runtime in seconds with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (FLEXDC, Abilene, $|T| = 5$). One figure per K_{DC}.

the following procedure to decide their respective ranges. For latency thresholds $\mathcal{C}_{\text{at}}$ and $\mathcal{C}_{\text{cl}}$, we multiply the maximal inter-node forwarding delay with the list of $[0.125, 0.15, 0.175, 2.0]$ and $[0.6, 0.7, 0.8, 0.9, 1.0]$ respectively, and for controller's load threshold $\mathcal{C}_{\text{l}}$, we use the list of $[30, 40, 50, 60, 70]$.

Our evaluation mainly considers two performance metrics: flexibility and runtime. The runtime is the time it takes to return the DC locations, which includes the time of optimizing controller placement and DC locations for FLEXDC. For a better demonstration, the performance metrics drawn in 2-D heatmaps: we fix one of the three thresholds with a large value (so that it does not impact the flexibility metric) and variate the thresholds of the other two (values as indicated above) on the two axes. Darker color indicates higher flexibility and longer runtime. We only show the cases with $K_{\text{DC}} = 2$, 4, and 6 from left to right; the observed trend is consistent for other cases.

5.4.3.2 Evaluation Results

What is the impact of adaptation time and control latency constraints? We start by analyzing the performance of OPTFLEX. Figure 5.5 illustrates the impact of adaptation time and average weighted control latency constraints. For 4 and 6 DCs, an increase of the adaptation time constraint threshold (ATC for short) with fixed weighted control latency constraint threshold (CLC for short) results in an increase of the flexibility, but this observation is not valid for 2 DCs. This is because 2 DCs would not allow any migration of 2 controllers, thus allowing longer migrations does not help. However, fixing ATC and increasing CLC leads to higher flexibility for all three cases. Another observation is that more DCs tends to increase the flexibility for the same constraint threshold combination. Meanwhile, compared with $K_{\text{DC}} = 5$ (not shown in the figure) and $K_{\text{DC}} = 6$, the increase is not obvious compared with the smaller number of DCs. This complies the effect of *diminishing return* [46]. For most of the threshold

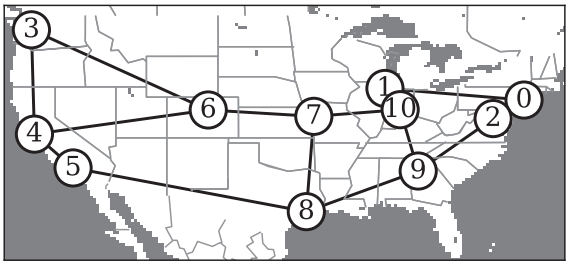


Figure 5.7: Topology of Abilene network.

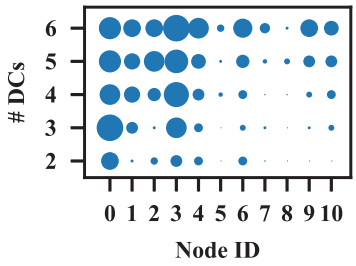


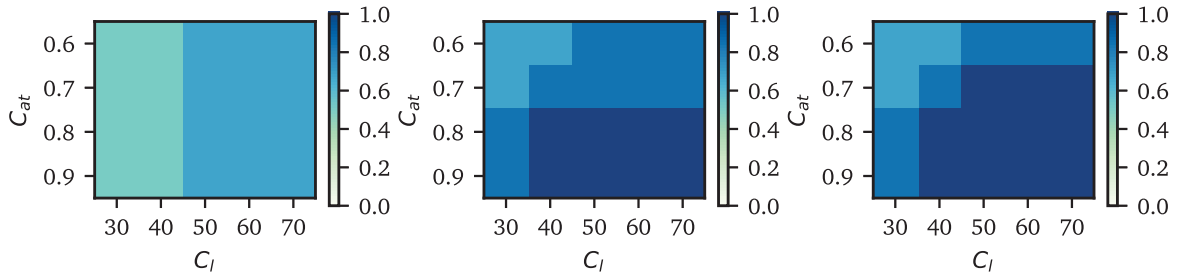
Figure 5.8: Summary of DC locations of 20 parameter combinations. The sizes of the circles denote the empirical distribution of the possibility that a node is selected to host a DC.

combinations, 6 DCs would be an *over-design* that does not bring much benefit but induces obvious higher cost (in terms of CAPEX, e.g., DC deployment cost, and OPEX, e.g., DC operational cost).

How long does the optimization of OPTFLEX take? The model is indeed complex, which is revealed in its runtime. Figure 5.6 shows the respective runtime in seconds in logarithmic scale. The increase of runtime are on par with the increase of K_{DC} for most threshold combinations. For $K_{\text{DC}} = 4$ and 6, a tight parameter combination, e.g., $\mathcal{C}_{\text{at}} = 0.6$ and $\mathcal{C}_{\text{cl}} = 0.15$, can result an optimization that takes more than 100 times longer than a relaxed combination, e.g., $\mathcal{C}_{\text{at}} = 0.8$ and $\mathcal{C}_{\text{cl}} = 0.2$. This means it is hard to optimize for flexibility with tight adaptation cost and time constraints.

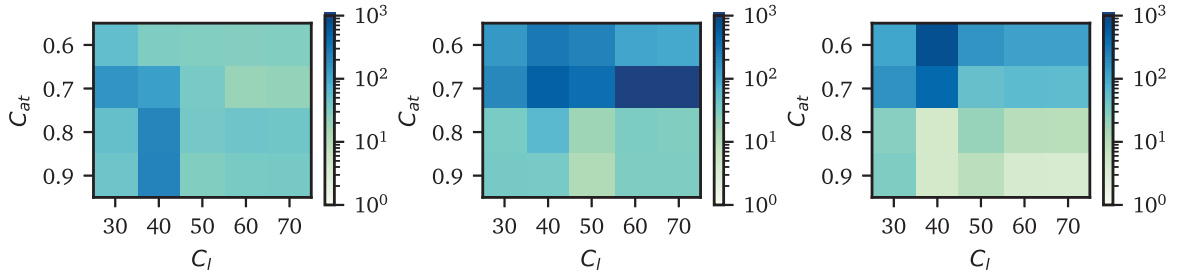
Does the DC selection show a certain pattern? To illustrate the pattern, Figure 5.7 shows the network topology of Abilene with node IDs, and Figure 5.8 illustrates the normalized distribution of DC locations of 20 threshold combinations (corresponding to 20 grids in the heatmap). The sizes of the circles denote the empirical distribution of the possibility that a node is selected to host a DC. In general, some nodes (e.g., 0 and 3) are more likely to be selected than others (e.g., 5 and 8). When less DCs are available, their location distribution is more concentrated on some nodes. The most selected DC locations for $K_{\text{DC}} = 2$ are (2, 3) and (0, 4), which are pushed towards two sides in the topology as controller migration is not allowed to happen. Therefore, taking graph features into account can reduce the size of the solution space and decrease the optimization runtime.

What is the impact of adaptation time and controller's load constraints? Figure 5.9 shows the impact of controller's load constraint (short for LC) and ATC, with fixed CLC. Similar to Figure 5.5, ATC does not impact flexibility for $K_{\text{DC}} = 2$. For all three cases of K_{DC}, increasing LC results in higher flexibility. The observation of runtime (not shown in the figure) is also similar to Figure 5.6. Again, we observe that tight parameter combination induces longer optimization runtime.



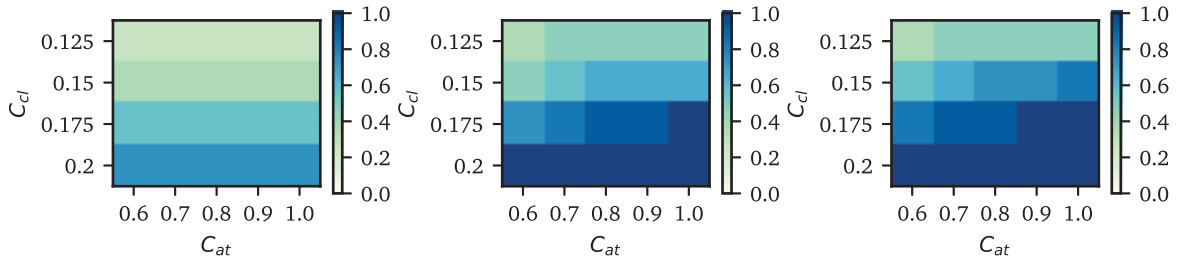
(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.9: Heatmap of flexibility with different controller's load threshold $\mathcal{C}_{\text{l}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (FLEXDC, Abilene, $|T| = 5$). One figure per K_{DC}.



(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.10: Heatmap of flexibility with different controller's load threshold $\mathcal{C}_{\text{l}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (FLEXDC, Abilene, $|T| = 5$). One figure per K_{DC}.

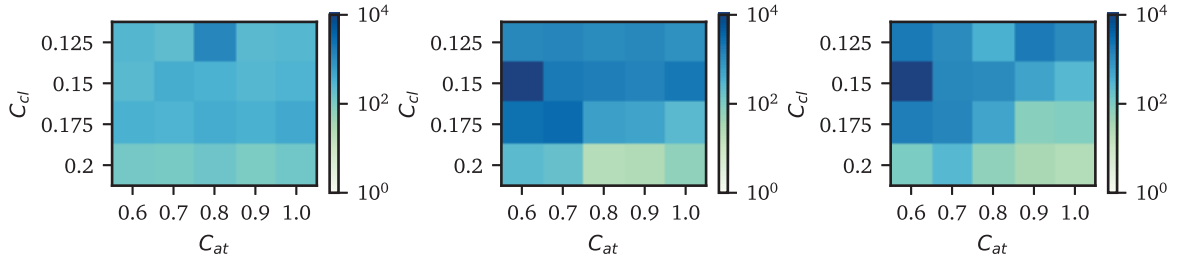


(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.11: Heatmap of flexibility with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (FLEXDC, Abilene network topology, $|T| = 10$). One figure per K_{DC}.

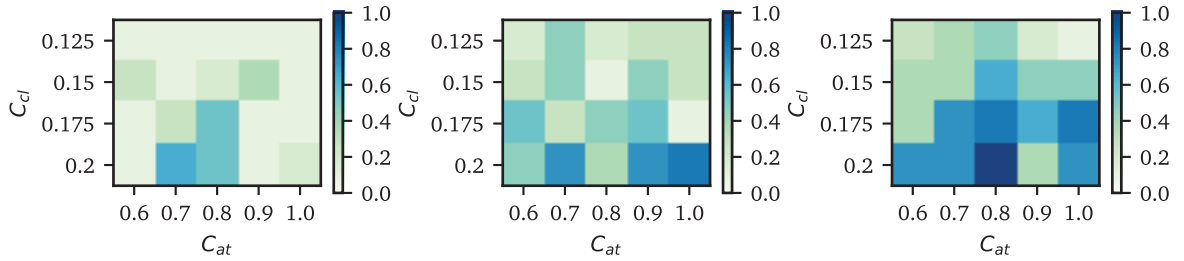
How does OPTFLEX perform for long planning period? To demonstrate this, we report the flexibility and runtime of $|T| = 10$ in Figure 5.11 and Figure 5.12. The general observation of different threshold combinations is similar to $|T| = 5$. Longer runtime is intuitive because of a larger input set. Regarding the flexibility, since the flows of different time-slots are randomly generated without interdependence, it becomes less likely to find an optimal DC placement to satisfy the constraints. Nevertheless, if the flows follow a diurnal pattern or are predictable, the flexibility would increase for both $|T| = 5$ and $|T| = 10$.

What is the performance of RANDOMDC? We now study the performance of the proposed efficient methods. As a straw-man approach, RANDOMDC selects DC locations randomly from the network topology, which takes negligible time. Because there is a respective set of DC locations for each threshold combination, Figure 5.13 looks rather irregular and does not show any meaningful pattern. For several



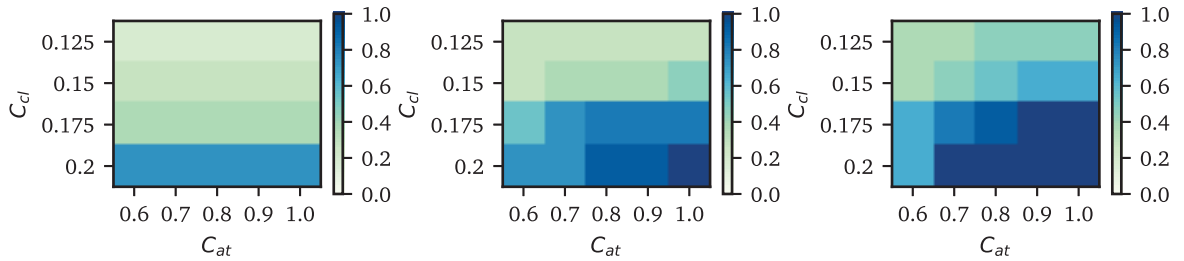
(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.12: Heatmap of runtime in seconds with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (FLEXDC, Abilene network topology, $|T| = 5$). One figure per K_{DC}.



(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.13: Heatmap of flexibility with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (RANDOMDC, Abilene network topology, $|T| = 10$). One figure per K_{DC}.

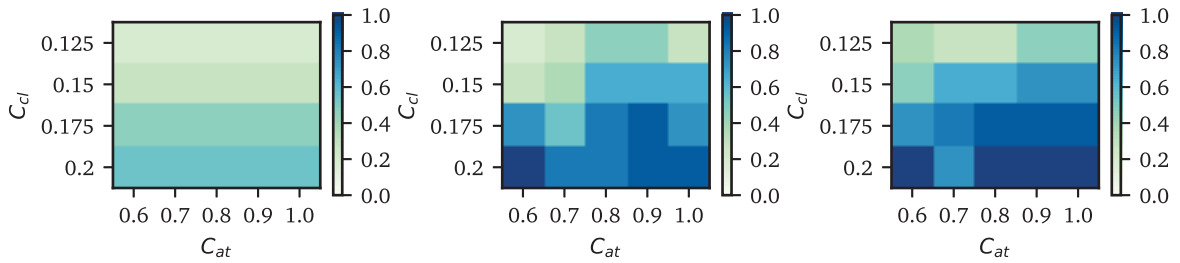


(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.14: Heatmap of flexibility with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (ACLDC, Abilene network topology, $|T| = 10$). One figure per K_{DC}.

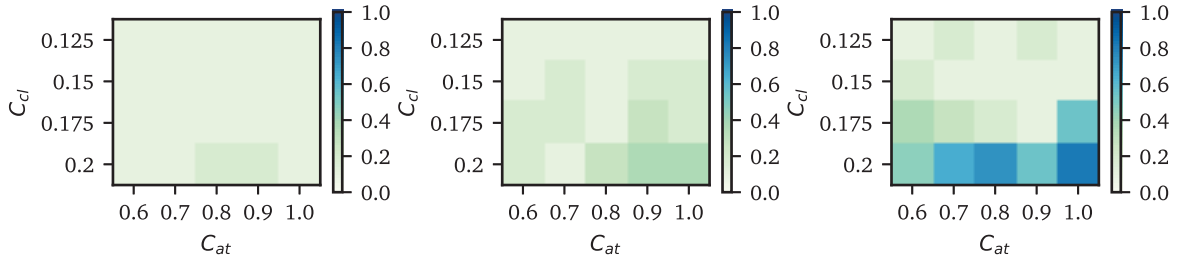
parameter combinations, the flexibility is nearly 0, whereas it can be more than 0.5 if optimized with FLEXDC. The trend here is higher K_{DC} tends to increase flexibility.

What is the performance of ACLDC? As the first heuristic method, this method takes K_{DC} as input and optimizes the locations towards the minimum average control latency as if all DCs are home to controllers. The runtime does not depend on the constraint combination, and it is in the magnitude of seconds for both topologies. Figure 5.14 depicts the achieved flexibility. Comparing with that of FLEXDC, its gap is minor for relaxed constraint combinations. In addition, ACLDC can achieve at least the same flexibility for almost all threshold combinations with 4 DCs (see the heatmap in the middle in Figure 5.14) with RANDOMDC with 6DCs (see the heatmap on the right in Figure 5.13), which is a significant cost saving.



(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.15: Heatmap of flexibility with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (HFDC, Abilene network topology, $|T| = 10$). One figure per K_{DC}.



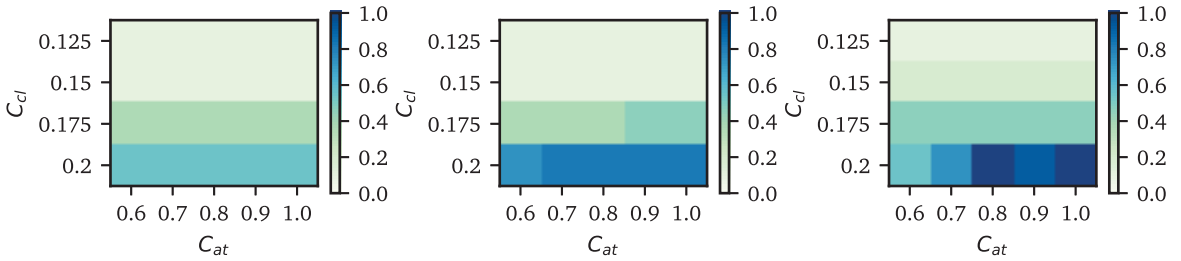
(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.16: Heatmap of flexibility with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (RANDOMDC, AttMpls network topology, $|T| = 10$). One figure per K_{DC}.

What is the performance of HFDC? HFDC only optimizes the first 5 time-slots and output the optimal DCs which are evaluated on all 10 time-slots. Therefore, we expect its runtime to be similar to the case of $|T| = 5$ which is shown in Figure 5.6. Figure 5.15 illustrates that the flexibility is worse in most cases compared with ACLDC. This is because the traffic pattern we have mentioned before. We envision that for traffic that is periodic or has a certain trend, HFDC would be a good candidate to save runtime.

The impact of larger network topology. We now evaluate AttMpls as a large network topology. First, the runtime of FLEXDC on AttMpls takes more than 10 hours for a single loose threshold combination, making it barely useful for the large-scale simulation study. Therefore, we only show the results of efficient methods. Figure 5.16 reports the achieved flexibility with RANDOMDC, which is worse in most cases compared with Figure 5.13. This is because as a larger network topology, AttMpls has more candidates for DC locations that does not support adaptation of the changing demands. In comparison, ACLDC is more promising to deliver high flexibility, as depicted in Figure 5.17. Meanwhile, we can observe that increasing K_{DC} and relaxing CLC result in higher flexibility, whereas varying ATC does not have an obvious impact.

To conclude, the threshold of constraints plays a crucial role in restricting the maximum achievable flexibility. The runtime of FLEXDC can hinder its usability for large $|T|$ and topology size, giving way to our proposed heuristic methods. The optimal DC locations of FLEXDC depends on the cost factors that we employ. When the constraints are not tight, ACLDC is the right candidate for FLEXDC, whereas when the traffic has a cyclic pattern, HFDC shows its advantage by only taking the flow profiles of the first cycle. Furthermore, both ACLDC and HFDC can achieve the same flexibility but with less number of DCs, comparing with RANDOMDC.



(a) $K_{\text{DC}} = 2$. (b) $K_{\text{DC}} = 4$. (c) $K_{\text{DC}} = 6$.

Figure 5.17: Heatmap of flexibility with different control latency threshold $\mathcal{C}_{\text{cl}}$ and adaptation time threshold $\mathcal{C}_{\text{at}}$ (ACLDC, AttMpls network topology, $|T| = 10$). One figure per K_{DC}.

5.5 Summary and Discussion

Following the research endeavor of the previous chapter, we focus on how to evaluate the flexibility of the dynamic control plane and how to optimize the flexibility accordingly in this chapter. We provide a detailed procedure to calculate the time the control plane takes to adapt itself in the face of a new traffic distribution, which comprises two processes: controller migration and switch reassignment. The calculation of the flexibility metric follows the mathematical definition in Chapter 3. Our evaluation on real network topologies reveals that in case only a short time is allowed for control plane adaptation, the number of controllers has almost no influence on the flexibility. If we tolerate longer migration time, as expected, a higher number of controllers leads to more flexibility. The evaluation also underlines the need for the dynamic control plane addressing changing traffic conditions and shows that the incurred cost decreases with higher flexibility.

For the second part of this chapter, we initiate the study of optimizing for flexibility. In particular, we consider the design phase of managing a flexible networking system, which is as important as operating the system but neglected in the literature. To design the flexible control plane, we first need to decide where to install the DCs, so that the control plane can adapt without violating the cost and time constraints. In this regard, we propose a mathematical programming model OPTFLEX, which plans the static DC locations over a time horizon and, at the same time, returns the controller placement and switch assignment in each time-slot. To increase the efficiency of the decision process, we also design heuristic methods ACLDC and HFDC. The former leverages the static CPP model to return an approximation of optimal DC locations, and the latter takes advantage of reducing the solution space. The evaluation shows the significant impact of adaptation cost and time constraints on the achieved flexibility. It demonstrates the necessity to come up with planning models and underlying techniques that can reduce the incurred cost and time while adapting the system in the face of new demand.

So far we have discussed the optimization opportunities of the control plane from the theoretical point of view. In the next chapter, we will move on to the practical part of this thesis and explore the reconfiguration of the data plane enabled by PDP.

Chapter 6

Towards Flexible Programmable Data Plane

6.1 Introduction

6.1.1 Motivation, Problem Scope, and Research Challenges

Previous two chapters discuss the opportunities in terms of the control plane resource management optimization and assume that the data plane abides the specification of OpenFlow without the ability of further customization. This chapter continues the road-map towards flexible softwarized networks by leveraging the flexibility envisioned by the novel PDP paradigm. In order to better understand the background of PDP, it is necessary to go beyond flow routing – where SDN shows superiority – and look at other types of NFs and the history of NFV.

Emerging applications and changing user demands are challenging today's communication networks: applications like Augmented Reality (AR) [133] and Vehicle-to-everything (V2X) [134] require ultra low latency, while big data applications introduce new dimensions of networking traffic in terms of scale and dynamicity. Such demands stand in stark contrast to the inflexible nature of the legacy infrastructure. Hardware-based routers and middleboxes, which cannot be easily upgraded or reconfigured, hinder the network operators from updating their infrastructure. Accordingly, NFV has emerged to deploy NFs as software running in off-the-shelf commodity servers. As a result, operators can flexibly instantiate, configure, migrate, and terminate NFs according to dynamic system conditions [135]; hence operators can efficiently adapt the infrastructure.

However, a pure software solution, as proposed by recent NFV approaches, raise the following concerns. First, it is hard for a software-based NF to achieve the line-rate processing target, especially with small packets [54]. Second, many NFs, such as DPI and packet en/decryption, are compute-intensive. The appliance of general-purpose CPUs – which are not explicitly designed for those tasks – is not cost-effective from the techno-economic perspective [136]. By leveraging both programmable software and hardware, PDP opens the gate of adaptable data plane with runtime reconfiguration capability while maintaining packet processing performance, and therefore becomes a promising research direction [137], [138]. With the data plane programming language P4, network operators can use the domain-specific abstraction to implement a variety of novel applications, including Inband Network Telemetry (INT) [139], advanced traffic engineering and load balancing schemes [140], and consensus protocols [141]. However, how to efficiently manage a P4 data plane that consists of various types of P4 devices is still missing in the literature, not to mention how to enable its runtime reconfiguration.

In the meantime, many PDP operations are stateful: the forwarding devices maintain states for the network services, e.g., registers for forwarding port mapping and meters for traffic rate control. In order to provide a predictable data plane behavior, we need to ensure state consistency at any time. For example, in autonomous driving, network services provisioned to the vehicles should be migrated together with their states (e.g., service ports and IP addresses), to guarantee connectivity and QoS. Therefore, how to automatically recognize all the states in a PDP program written in P4 that should be maintained during reconfiguration: an aspect which to the best of our knowledge has also not been addressed in the literature so far. State recognition, however, is challenging because of the full range of possible notions of states (and access methods) in P4-based PDPs.

The above-mentioned research challenges need to be addressed in order to fully leverage the flexibility offered by PDP.

6.1.2 Key Contributions

The contributions presented in this chapter are based on our work in [10] and [11], and can be summarized as the following:

1. An NFV management architecture P4NFV is proposed for the P4-enhanced data plane to manage different types of P4 devices [10]. The architecture achieves the design goal of abstraction, flexibility, and consistency while offering the capability to reconfigure the functionality of forwarding devices at runtime without service disruption.
2. Based on a characterization of states in the P4 data plane, P4STATE analyzer is presented as a suite of algorithms that analyze P4 programs and identify the states that need to be maintained during functionality reconfiguration to avoid data plane inconsistency [11].
3. The performance of the P4 data plane realized with different targets during reconfiguration is comprehensively measured to understand the potential overhead in terms of the packet processing latency during reconfiguration and the latency of reconfiguration [10].

The rest of the chapter is organized as follows. Section 6.2 introduces the background of general NFV management architectures and data plane state. In Section 6.3, an architecture called P4NFV is proposed to manage P4-enhanced data plane while enabling flexible reconfiguration. To maintain the consistency during the reconfiguration, P4STATE is presented in Section 6.4 to return necessary state variables that need to be synchronized. Section 6.5 evaluates the performance of reconfiguration of different P4 targets. Finally, Section 6.6 summarizes this chapter.

6.2 Background and Related Work

This section first introduces the general background on NFV management architectures, followed by the approaches of data plane reconfiguration. For the sake of state management during reconfiguration, it offers a taxonomy of data plane states in the context of P4. In the end, it presents information about state management and analyzers for network functions.

6.2.1 NFV Management Architectures

The technology concept NFV enables fast and cost-efficient NF provisioning; hence, it complements SDN and PDP towards more flexible softwarized networks. NFV transfers the NFs from vendor-specific and proprietary hardware devices to software running in off-the-shelf commodity servers that are equipped with standard processing (i.e., CPU), memory (i.e., RAM), and storage components (i.e., hard disks). As a common practice, network services are provided in VMs, each performing respective operations such as routing, firewall, or load balancing. In this way, the cost to implement a software function is mostly smaller than the hardware counterpart; besides, the speed to scale up and down a particular network service is also faster, thanks to the general interfaces to the commodity services which do not need many man-hours.

The management of VNF, its life-cycle (from instantiation till termination) in particular, is the fundamental task of the NFV management architecture; resources of processing, memory, and storage are allocated to each NF instance and adapted at runtime to satisfy the respective service requirement that might be highly dynamic. NFV orchestration is often considered together with NFV management, and is responsible for realizing network services after the VNFs are instantiated by the NFV manager. A variety of NFV architecture proposals have been suggested to tackle various aspects of resource management in NFV [142]–[145], with focus on, e.g., automation of NF provisioning, NF latency reduction, dynamic resource scheduling, and dynamic service function chaining.

Because of the complexity of softwarized network construction, vConductor [142] proposes to automate the NF provisioning procedure, while considering various resource scheduling and infrastructure fault isolation. The trend of pushing NFs to the network edge and steering the traffic using a centralized SDN controller simplifies the deployment procedure, and even further increases the resource usage efficiency and reduces management cost and latency. Following this trend, NetFATE [143] deploys NFs not only in high-performance DC servers but also in the nodes that are close to end-users. NFVnice [144] dynamically schedules resources allocated to service chains and thus enables fair share of CPU to NFs. Further, Callegati el al. propose to orchestrate and chain NFs in a DC and demonstrate the high dynamism and flexibility of function chaining compared with legacy hardware infrastructure [145].

However, because NFV generally relies on software data plane implemented as high-performance packet Input/outputs (I/Os) to userspace and virtualization of packet processing. For example, SoftNIC [146] uses KVM [201] for virtualization and Intel DPDK [192] for packet I/Os, whereas ClickOS [147] uses Xen [202] and netmap [148] instead. As the requirements of today's networking services can be more strict in terms of throughput and latency, pure software data plane may not be able to satisfy them due to its limited packet processing capability. Indeed, the measurement study conducted by Niu et al. [54] demonstrates that software data plane can achieve line-rate only for medium to large packets with sizes larger than 128 Bytes, even with a low CPU clock cycle. For small packets, the line-rate is hard to reach for a typical CPU with 2.6 GHz clock cycle, which indicates the performance bottleneck. Besides, packets processing with software incurs high and highly variable latency [149]. For instance, Ananta [150] – a software load balancer – can have processing latency from 200 microseconds to 1 millisecond at the load of 100K packets per second (pps). Hence, software data plane is not competent in all scenarios, making it important to consider the traffic characteristic and requirement before wide adoption. Meanwhile, hybrid data plane with hardware enhancement is also a promising direction to pursue.

We envision that the benefits of PDP with P4 can address the potential issue of performance bottleneck while maintaining the capability to reconfigure itself in a flexible manner. Unfortunately, the proposed architectures above all assume that NFs are implemented as general software, which is not applicable to a data plane for P4 composed of software and hardware programmable devices.

6.2.2 Data Plane Reconfiguration Approaches

The reconfigurability of data plane can better satisfy dynamic networking applications, and user demands at a lower cost. In the meantime, it introduces new challenges, such as disruption during reconfiguration and state management. In the following, we report on works that target reconfigurations enabled with and without P4.

6.2.2.1 Reconfiguration without P4

Several works [151]–[153] tackle the problem of how to reconfigure the data plane with minimal service disruption. The Split/Merge approach [151] considers the dynamic scaling of VNFs. A hypervisor abstracts the states of VNFs and manages their redistribution upon creating or destroying VNF replicas. OpenNF [152] is a control plane architecture that manages both VNF state and networking forwarding state. Special APIs and a combination of events and forwarding updates can redistribute the packet processing across a group of VNFs. Zave et al. design a protocol named Dysco [153] to enable dynamic service chaining. When the sequence of NFs changes, the protocol reconfigures the data plane packets of the corresponding TCP session with a small disruption.

6.2.2.2 Reconfiguration Leveraging P4

The reconfiguration capability of P4 has been leveraged to achieve data plane virtualization in the works such as HyPer4 [154], HyperV [155], and P4Visor [156]. Like in other virtualization scenarios, the objective is to enable the sharing of networking resources between multiple tenants (the ones that receive virtual resources, i.e., a portion of the physical networking resources). These approaches introduce a hypervisor that enables multiple P4 programs to run in an isolated way on the same packet processing entity. Upon initialization, each processing entity is configured with all necessary P4 programs. A table is used to dispatch the tenant networking traffics to the P4 programs of the tenants respectively. By updating specific table entries, the hypervisor can even turn on and off the programs at runtime. We leverage a similar approach to accommodate multiple P4 programs simultaneously on one network processing entity. In contrast to the virtualization approaches, we focus on the capability to reconfigure the processing pipelines, i.e., to dynamically steer networking traffics between P4 pipelines running on the same networking entity.

6.2.3 Taxonomy of P4 Data Plane States

P4 enables the definition of states in the data plane, which calls for consistent mechanisms during reconfiguration. Tradition data plane update techniques would not apply in this new context due to the following concerns. To begin with, state variables inside P4 data plane can be updated at line-rate, e.g., up to Tbps [183], which invalidates the naive approach that synchronizes all variables to maintain consistency properties. As a result of inconsistent reconfiguration, network services can experience unexpected behaviors. For instance, a stateful firewall application tracking flow statistics [157] might fail to block some malicious flows because of the incomplete statistics represented as states. Second,

because each forwarding device in a P4 data plane can have a distinct set of NFs, they do not share an exact set of state variables. Hence, traditional techniques, often supported by the control plane, fail to locate all necessary states that need to be maintained [158].

In order to manage data plane reconfiguration in a consistent manner, it is critical to design an approach that automatically identifies the states. We start with a taxonomy study in this regard. The definition of states in a P4 data plane is quite broad [158], [159]. States include *(i)* table entries, *(ii)* stateful variables defined in the P4 specification, and *(iii)* (some) temporary variables defined in a program. We include the temporary variables only if they act like pointers that refer to stateful variables. Since the table entries can be recognized and maintained by the control plane during data plane reconfiguration without much effort, we do not explicitly consider them.

The P4 specification [190] defines three types of stateful variables: *register*, *meter*, and *counter*. All the variables need to be persistent, i.e., their values should persist beyond a single iteration of the packet processing loop [160]. We focus on the *register* variable that is commonly adopted in real P4 programs. Regarding meter and counter, we briefly discuss the mechanism to maintain their consistency later in the discussion Section 6.4.4.

6.2.3.1 Register Usage

We identify the following scenarios when *register* variables can be declared:

- a value that the processing of the following packets can access, e.g., packet counter[1] [68].
- a value that the control plane can access for making control decisions, e.g., the status of a port [161].
- a value that controls the packet processing in a P4 node, e.g., a flag enabling on-demand functionality [10].

The usage of a register is one of the factors indicating whether it should be transferred during data plane reconfiguration. As an example, Figure 6.1 shows the declaration (line 2) and read-access (line 5) of the content of register `flag_reg`. Note that in this example, we denote `flag_reg` as a **register type** and the content as a **register entry**. After surveying the public available P4 projects, we create Table 6.1 that gives an overview of the P4 programs that leverage registers in their implementation. A more detailed description of all P4 programs is elaborated in our online repository [21].

Register access within a P4 program can be either read or write (both indirectly and directly). In Figure 6.1, the binary register value decides the following packet processing path: either line 6 or line 7. This is an indirect access in an if-conditional, i.e., a temporary variable that refers to the value of a register entry is evaluated. Register entries can *only* be directly accessed in actions. Figure 6.2 demonstrates an example, where a register entry is read and copied to one field in the user-defined custom metadata. Actions are always associated with tables; an action is called either based on the matching result of a table or when a packet processing path traverses a table. Meanwhile, the value of a register entry can impact the decision of an if-conditional.

[1] Similar to the usage of a native counter, but supported by more P4 targets.

```
control ingress(...) {
register<bit<1>>(32w1) flag_reg; ...; // register definition
  apply {
    bit<16> flag;
    reg.read(flag, 0);          // register read access
    if (flag == 1) {...;}       // one packet processing path
    else {...;}              // another packet processing path
  }
}
```

Figure 6.1: Register declaration and indirect register access defined in P4. The register value decides the subsequent packet processing path.

Table 6.1: Overview of Register's Usage in P4 Programs.

Program Name	LoC	Types of Reg.	# Reg. Ent.
heavy hitter	178	2	32
flowlet	203	2	16 384
netpaxos [162]	210	6	256 002
ndp [163]	223	2	4
hashpipe [68]	229	8	224
hula [140]	289	4	65 632
dapper [164]	535	22	86
sketchlearn [64]	646	32	8 192
linearroad [165]	789	11	6 096
netcache [166]	1 427	40	6 784

Note: The LoC calculation is based on programs written with P4-16 for the BMv2 simple switch target [181]. Programs written in P4-14 are translated to P4-16.

```
action read_register() {
  hash(reg_index, HashAlgorithm.crc16, (bit<32>)0, {...}, (bit<32>)65536);
  reg.read(meta.custom_metadata.val, reg_index);
}
```

Figure 6.2: Direct register access in an action defined in P4. Both `read` and `write` operations are supported.

6.2.3.2 Register Classification

We classify registers into two categories, namely *flow-based* and *device-based*. A *flow-based* register saves per-flow state and typically instantiates a large number of entries (e.g., 65 536), which can be migrated on the data and control plane. A *device-based* register saves the state that is device-specific. It has fewer entries compared with the flow-based counterpart but may need to be migrated with the help of the controller. The classification helps to coordinate the maintenance of various registers at runtime, i.e., decide how to migrate them.

Luo et al. [158] advocate that it is not necessary to migrate the flow-based register that is computed from the events of arriving packets, e.g., the *flowlet_id*, which denotes a flowlet and is hashed from the

header field tuple. However, we argue that in order to maintain the consistency requirement, we have to migrate those flow-based registers. For example, when the *flowlet_id* determines the egress port, the loss of its values might lead to the following packets of the same flowlet to be forwarded on a different path; hence, larger jitter and longer delays can be expected.

6.2.3.3 Register Migration

States can be migrated either through the data plane [158], where register values are carried as specific header fields, or through the controller [10], which performs direct register read/write. For the first approach, a *flow-based* register entry needs an exact flow to piggyback its value, which can lead to long migration latencies when flow patterns change quickly, and the expected flows do not show up on time. Longer latencies violate the original intention of flexible adaptation. For the second approach, since the state values need to be copied first from the source node to the controller and then copied to the destination node, we need to take into account the extra forwarding time from the nodes to the controller. Note that we assume the controller integrates the functionalities of both the control plane and the management plane.

No matter which approach is applied, we assume that the overall migration time increases with the number of register entries. The migration time is defined as the overall time spent on migrating all necessary states before the new function starts to work correctly. For the flow-based registers, i.e., the ones with indices calculated with hash functions, we may potentially create a huge number of (more than 2^{16}) entries, depending on an initial estimation of how many flows can show up in the network. However, not all entries are filled with effective values that need migration. In other words, an intuitive approach that strives to migrate all register values would induce a long overall migration time. Therefore, it is essential to propose an approach that recognizes only the necessary register values.

6.2.4 Data Plane State Management

The research in the state management of the data plane is quite abundant. Split/Merge [151] requires middleboxes to allocate and access all states through a customized shared library. OpenNF [152], however, transfers directly the serialized states between different middleboxes. From a diverse perspective, StatelessNF [167] requires middleboxes to create/read/update states in a central data store, which allows any middlebox to access any state at any time. SNAP [159] considers state allocation in a static scenario; the whole network is considered as a single switch and the location of states in the form of forwarding rules are optimized to enforce policy.

6.2.5 Network Function Program Analyzer

Many tools were proposed to analyze data plane programs in terms of performance or security. CASTAN [168] and BOLT [169] discover execution paths within the code of an NF and recognize potentially large resource consumption, e.g., CPU cycles and memory accesses. P4pktgen [170] analyzes a P4 program and generates input packets and table entries that cover all execution paths. Assert-P4 [171] and Vera [172] leverages assertions and symbolic execution to validate the general network correctness properties. Birnfeld et al. propose to only use Control Flow Graph (CFG) analysis to verify switch programs without the need of custom verification code [173]. However, the analysis for data plane reconfiguration is still missing in the literature.

6.3 Enabling Data Plane with Dynamic Reconfiguration

This section introduces a management architecture called P4NFV that aims at managing a heterogeneous data plane infrastructure comprising of software and hardware P4 forwarding devices. We assume a scenario where a network provider deploys NFs or whole function chains over time to process networking traffic of different network services.

6.3.1 Architecture Design of P4NFV

We start with enumerating the challenges of an NFV management architecture leveraging data plane reconfigurability, including abstraction, flexibility, and consistency. Thereafter, we introduce the components of P4NFV that can manage NFs implemented with P4. We also show that P4NFV complies with the guideline NFV architecture proposed by the European Telecommunications Standards Institute (ETSI) organization [203].

6.3.1.1 Design Goals

To support network dynamics, an NFV management architecture should consider the following three aspects.

Abstraction

In order to balance the trade-off among performance, investment, and revenue, network providers should be able to deploy heterogeneous devices, i.e., commodity server and hardware equipment, for NF provisioning. To ease the operation of the heterogeneous devices, abstraction should hide target-specific implementation details, APIs, and performance trade-offs among heterogeneous data plane platforms. Abstracting the infrastructure simplifies the procedure of managing both software and hardware resources. Realizing abstraction via an additional layer enables infrastructure providers to offer various NFs, from lite ones, such as packet forwarder, to advanced ones that are more compute-intensive, such as packet en/decryption and deep inspection.

Flexibility

The architecture should be able to cope with dynamics such as changing networking traffic conditions or changes in terms of network service requirements represented as SLAs. For instance, during the operation of an NF, the requirements of the NF in terms of QoS, as well as reliability and resilience, may alter over time. The above dynamics should be handled through proper NF management schemes, including feature upgrade, instance migration, parameter adaptation, etc. In other words, it should be possible to instantiate, (re-)configure, (re-)located, and upgraded each deployed NF in a flexible manner, with minimum interruption of operation [1].

Consistency

Besides abstraction and flexibility, a holistic architecture design should also integrate consistency. Consistency aims at two aspects, namely consistency during operation without any adaption and consistency when actually adapting existing NF deployments. First, during operation, performance consistency ensures that all abstracted resources provide the capacities as they claim without any unpredictable violations. This can be particularly challenging when facing heterogeneous P4-enabled

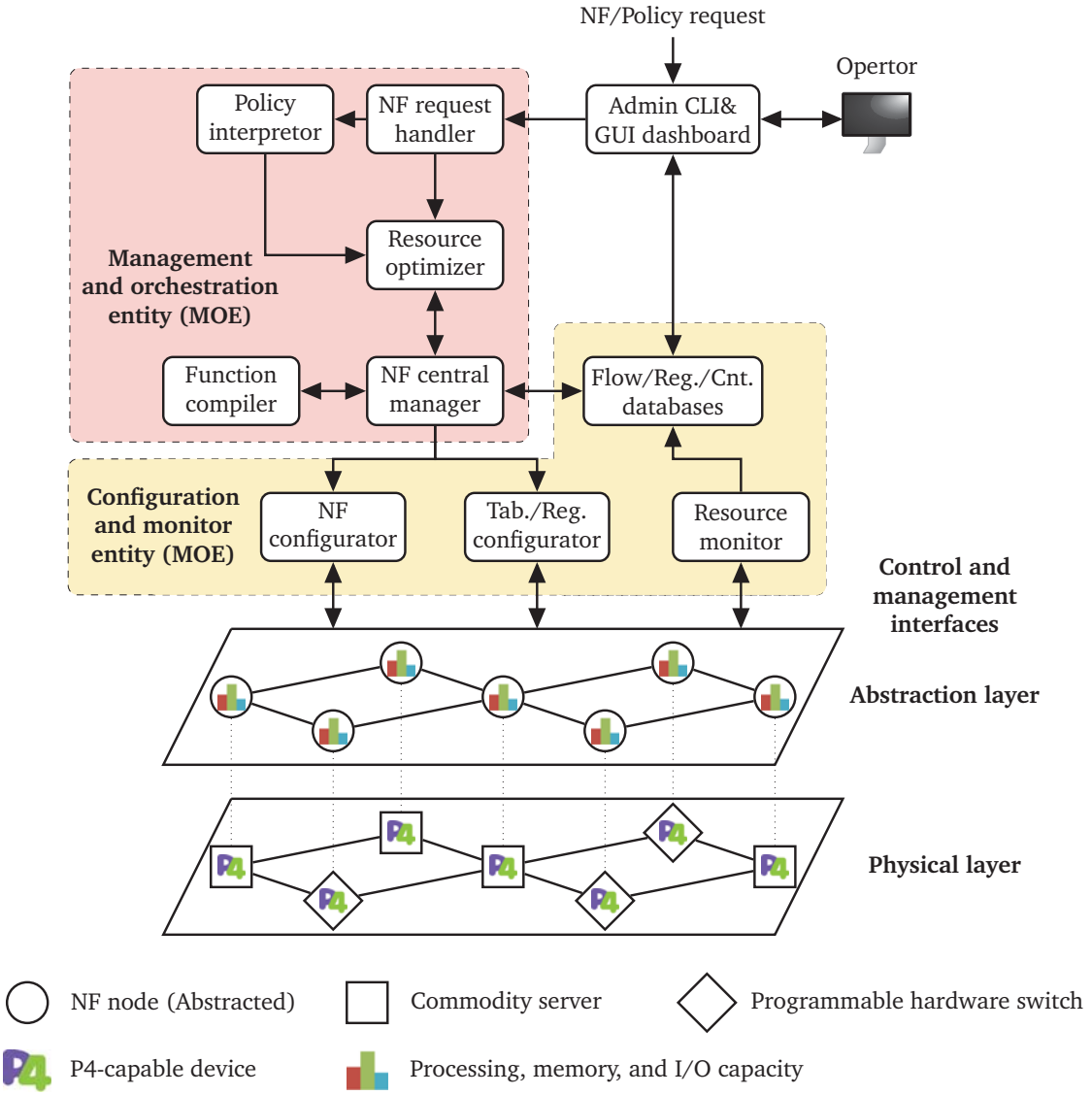


Figure 6.3: Illustration of P4NFV's architecture design. The physical layer consists of different P4 forwarding devices which are abstracted in the abstraction layer for the ease of management.

platforms with performance tradeoffs [54]. Second, when adapting the NF deployments, e.g., when migrating an NF from a hardware to a software target, the performance guarantees should still maintain. Moreover, logical consistency should always be preserved, even in case of reconfigurations; e.g., a stateful NF such as an L4 firewall should not allow malicious traffic during reconfigurations.

6.3.2 Architecture Description

Next, we introduce the P4NFV architecture by overviewing its components and clarifying how P4NFV realizes the design goals, as mentioned in the last section. Figure 6.3 illustrates the architecture of P4NFV. Logically, it comprises five components: *(i)* the Physical Layer, *(ii)* the Abstraction Layer, *(iii)* the Control and Management Interface (CMI), *(iv)* the Configuration and Monitor Entity (CME), and *(v)* the Management and Orchestration Entity (MOE).

Physical Layer

The physical layer consists of various types of packet processing entities, i.e., targets, that can be programmed with P4. As shown in Figure 6.3, squares represent commodity servers that can host software targets, e.g., BMv2 [181] and PISCES [60], and diamond squares represent programmable hardware targets, e.g., NetFPGA [36] and SmartNIC [182]. Note that the physical layer can be extended to incorporate other entities which are not programmable. For instance, commodity routers with configurable forwarding information bases can also be controlled in the architecture. Such hybrid infrastructure benefits from both deployment flexibility [136] and CAPEX/OPEX cost saving [143].

Abstraction Layer (NF Nodes)

This layer abstracts the underlying physical resources for packet processing. All the *abstracted* entities can implement P4 match-action pipelines. Specifically, each entity is abstracted as an NF node to host multiple NFs equipped with processing, memory, and I/O resources. Through the abstraction, various targets are modeled with their own performance characteristics, e.g., whether they can process packets in parallel. The performance models assist the network operator to decide the best target for a particular NF requirement and simplify the resource management process. To preserve the data plane's consistency, the resources are monitored by the upper components to alleviate anomalies, e.g., overload situations, which may cause data plane performance degradation.

Control and Management Interface (CMI)

This interface is the logical communication channel between *CME* (introduced later) and the underlying NF nodes. For different targets, the actual implementation of *CMI* can differ; however, the distinction is completely transparent to the above components in the P4NFV architecture. In order to connect the *Configurator*s and the *Resource Monitor* of *CME* with the underlying NF nodes, the control/management interface defines the logical communication channel, which is target- and protocol-independent. All the operations, e.g., pushing compiled P4 programs to NF nodes, populating match tables, fetching counter values, and reading/writing registers, passing through this interface.

Configuration and Monitor Entity (CME)

The components of the Configuration and Monitor Entity (CME) are the *Configurator*s, the *Resource Monitor* and the *Database*s. Based on the configurations received from *MOE*, the *Configurator*s take the responsibility of implementation of the NFs in the respective physical NF nodes.

The *Resource Monitor* collects the statistics from the *Abstraction Layer* periodically and notifies *MOE* in an event-based fashion, i.e., whenever any performance indicator, e.g., the load balancing factor or physical link utilization, violates a predefined threshold. For example, if a software firewall experiences sudden traffic volume increase, its CPU usage may rise dramatically and impacts the other NFs in the same server. In this case, the *Resource Monitor* captures the high CPU usage and thereafter informs *MOE* which triggers *Resource Optimizer* to determine a new location for hosting it. Failures in infrastructure, e.g., memory read/write anomaly, should also be observed and eliminated before potential performance degradation would happen. Besides, the *Resource Monitor* collects the values of registers and counters and keeps such state information in the *Databases*. With the help of registers, flow information can be stored, such as header fingerprint, average arrival rate, and forwarding state. The *Database* approach

has a clear advantage: it facilitates the maintenance of the global consistency of various NF instances during data plane reconfiguration.

NF Management and Orchestration Entity (MOE)

As the central component of P4NFV's architecture, *MOE* consists of the *NF Request Handler*, the *Resource Optimizer*, the *Policy Interpreter*, the *NF Central Manager*, and the *Function Compiler*. The network operator interacts with *MOE* through the *Admin Command-Line Interface (CLI) & Graphical User Interface (GUI) Dashboard* module offering different management operation possibilities: e.g., configuring global policies or checking the resource usage of nodes.

MOE automates the whole process of initiating, coordinating, and managing NFs. The *NF Request Handler* listens to new NF requests, as well as policy updates of existing NFs. Together with the *NF Request Handler* and the *Policy Interpreter*, the *Resource Optimizer* implements the requests with optimized configurations, including the P4 programs.

The configurations are directed to the *NF Central Manager*, and then passed to *CME*, which in turn implements them in the NF nodes. In the meantime, the *NF Central Manager* listens to the data plane status via *CME* and re-optimizes the NF deployment and configuration. The re-optimization can be triggered either by the operator manually or by the *NF Central Manager* itself. The *Function Compiler* is a set of compilers (e.g., P4C [191] for BMv2 and P4-SDNet [204] for NetFPGA) and is able to compile P4 program for different targets.

6.3.3 Mapping to the ETSI Architecture Framework

P4NFV is compliant with the official recommendation of the standardization group. ETSI proposed a guideline NFV reference architecture framework in [203] (portrayed in Figure 6.4). The framework defines the functional blocks and the reference points needed to support the infrastructure services in the operator's network. Within the context of NFV, the infrastructure services are referred to as the network services, which are provided by the NFs. To show the compliance, we map the components of P4NFV to blocks in the framework.

In the reference architecture, the Virtualized Infrastructure Manager (VIM) controls the virtualization process and exposes the Network Functions Virtualization Infrastructure (NFVI) to the other modules. In P4NFV, the *Resource Monitor* and the *Resource Optimizer* take over this part, and the abstraction layer corresponds to the NFVI.

The intermediate level in the reference architecture consists of various VNFs and Element Management Systems (EMSs). Each VNF is an implementation of a networking application running atop the NFVI resources. VNFs' instantiation and termination is controlled by the VNF Manager(s), represented by the *NF Central Manager* and *Function Compiler*. During a VNF's life-cycle, its management, such as fault recovery, performance monitoring and accounting [203], are handled by its corresponding *EMS*, which corresponds to the *NF Configurator*, *Table/Register Configuration* and *Resource Monitor* in P4NFV.

The NFV Orchestrator at the top-level realizes NF requests by coordinating other modules in the reference architecture. It is represented as the combination of the *NF Request Handler*, *Policy Interpreter* and *Resource Optimizer*.

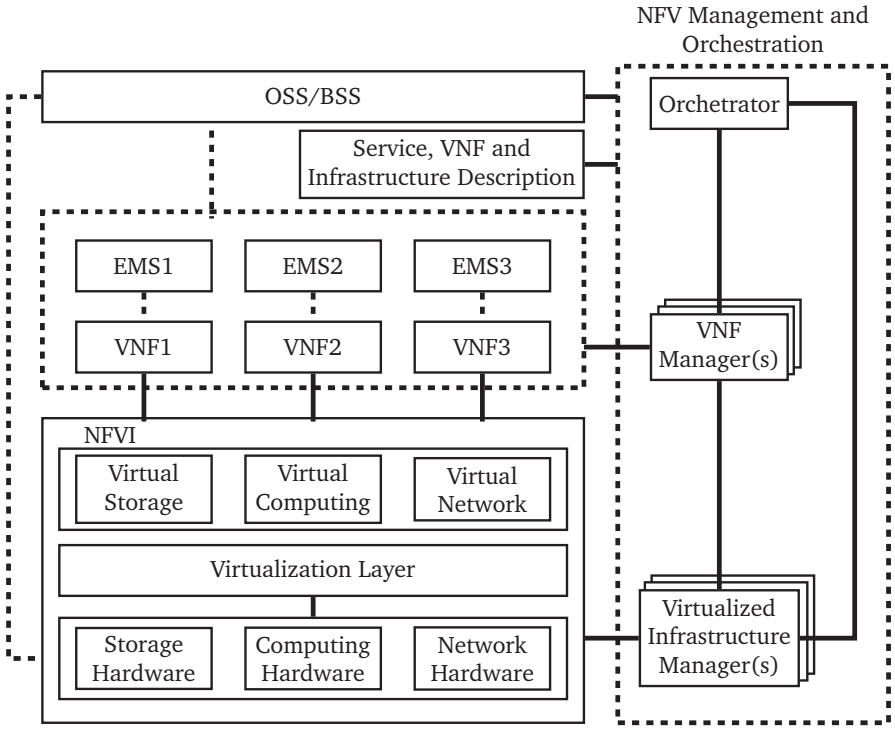


Figure 6.4: ETSI NFV reference architecture framework (adapted from [203]). P4NFV realizes the ETSI architecture, where, in addition, physical resources are also enhanced by P4 forwarding devices. Consequently, the NFVI programs those devices, and hence, VNFs' operation is still completely transparent.

6.3.4 Enable Runtime Reconfiguration

Next, we explore the capability of P4NFV to reconfigure the functionality at runtime. We face two challenges while implementing the Proof-of-Concept: *(i)* how to reconfigure the data plane during runtime, and *(ii)* how to reduce the impact of reconfigurations on packet processing. Accordingly, we propose two approaches called Pipeline Manipulation `(PM)` and Program Reload `(PR)`. Figure 6.5 demonstrates the two reconfiguration approaches.

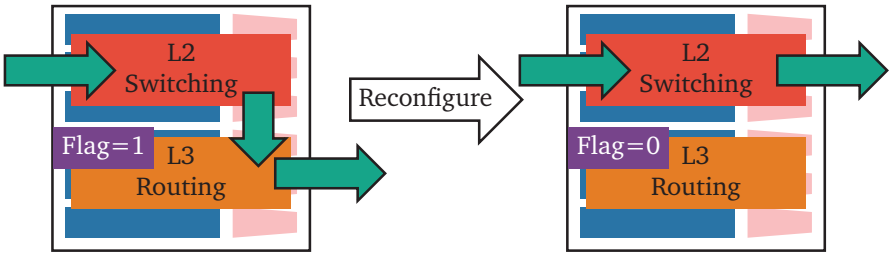
6.3.4.1 Pipeline Manipulation

With this approach, after loading the P4 program in the target, we use binary register values to manipulate the packet processing match-action pipeline at runtime. The pipeline is described by the CFG, which portrays the dependency between different tables and their associated actions (details are introduced in Section 2.2.2).

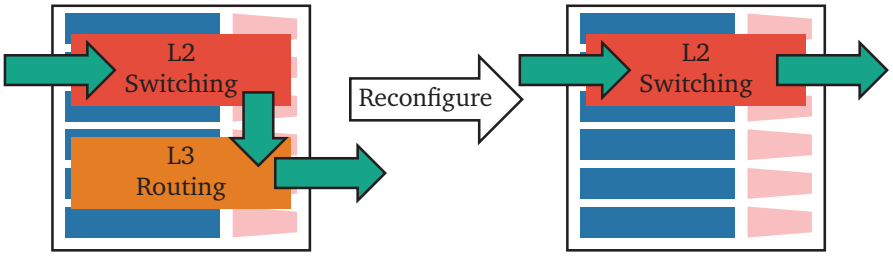
Figure 6.6 shows the pipeline of a switch that performs L2 and L3 forwarding. MAC/IP forwarding and ACL are considered as different NFs, each guarded with a binary register value to indicate its existence. Upon initialization, all functions are enabled. After we change the register value of MAC forwarding from 1 to 0 through the control plane, the packets will bypass it and jump to the IP forwarding NF. Similarly, we can also disable IP forwarding plus ACL and only keep MAC forwarding.

6.3.4.2 Program Reload (PR)

This approach provides more flexibility to reconfigure the functionality of the NFs, in comparison with `PM`. Every time a NF node needs to be reconfigured, the *target* will be loaded with a newly compiled P4



(a) The Pipeline Manipulation (PM) approach uses register value to indicate which functions are enabled.



(b) The Program Reload (PR) approach switches to a new pipeline completely.

Figure 6.5: Illustration of the two proposed data plane reconfiguration approaches. The green arrows represent the path of packet processing. After reconfiguration, only L2 switching NF remains in the NF node.

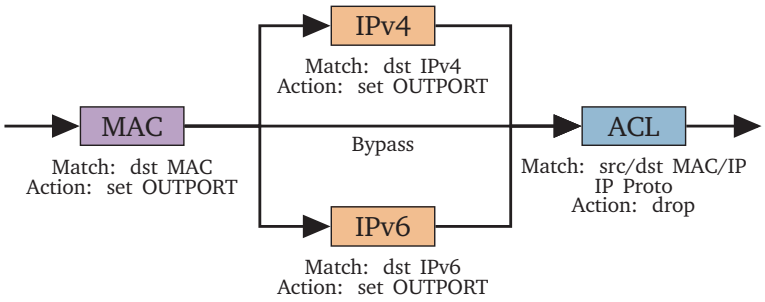


Figure 6.6: An example of packet processing pipeline of a P4 switch. Based on the IP version field, the egress port is decided by either looking-up the IPv4 table or the IPv6 table. The IP matching phase can also be bypassed for non-IP packets. The last ACL table drops malicious traffic.

program (including parser, processing actions, and deparser), followed by populating the new match table entries. Note that because of the implementation limits, some targets, especially hardware ones, may need to stop packet processing completely when a new configuration is being loaded, which causes service interruption.

6.3.4.3 Trade-offs between PM and PR

For PM, all NF nodes run the same P4 program, which includes all NF implementations. The *MOE* only needs to update the registers. However, PM demands additional match-action resources, which limits the total number of NFs that can be implemented at the same time. Furthermore, since registers can only be applied inside the pipeline, it is not capable of reconfiguring parser and deparser. When it comes to PR, different versions of P4 programs are pushed to different NF nodes, which could decrease reliability and increase management cost. Furthermore, PR may lead to service interruption for some targets. The advantages of PR are *(i)* that it demands less match-action resources, and *(ii)* that it can reduce

packet processing latency with proper pipeline compilation (i.e., merging different tables). Based on the NF requirement and the available infrastructure, operators can decide to use PM or PR to achieve a cost-effective NF implementation.

Regarding the state synchronization during an update of the stateful data plane, we discuss it in the following section.

6.4 State Management during Reconfiguration

This section copes with the state management during data plane reconfiguration and complements the architecture of P4NFV. We first demystify how to migrate the states between two P4 nodes. Afterward, we design a language analyzer called P4STATE, which derives the state variables that need to be maintained.

6.4.1 State Migration Approach

During data plane reconfiguration, the states (e.g., register values) that reside in the NFs need to be consistently updated; otherwise, the newly arrived packets will experience improper processing. Consider the migration of an IDS (Intrusion Detection System), where packet drops happen during the migration. The new IDS instance may have an incomplete fingerprint of a malicious flow as some packets are dropped and not recorded; hence, the IDS may fail to report the attack [174].

There are two options to preserve the states during reconfiguration. First, the states can be transferred directly in the data plane together with the live traffic [158]. In this case, NF nodes attach the state information directly to the respective flow. Second, a central entity collects the states from the data plane and redistributes them afterward. P4NFV applies the second option, because the first one can suffer from long overall migration time, when some of the flows do not show up during the migration. The *Resource Monitor* uses *CMI* to collect the states and stores them in the database. Notably, the introduction of *MOE* and *CME* induces latency overhead for state updates. In order to reduce such latency, P4NFV periodically fetches the states associated with the NF and only redistributes the states that are involved in the reconfiguration. Furthermore, advanced state management capability, such as state merge, split, and recompute [151], can be supported by a centralized manager. This is extremely helpful when scaling out a NF, e.g., stateful firewall, and the states are split across different firewall replicas [151].

The *MOE* coordinates the state migration process. We refer to Figure 6.7 to illustrate the process of PR for the BMV2 software switch [181], which involves four types of control plane messages. Note that for other P4 targets the message syntax might need to be adapted. The `load_new_config_file` indicates that a new data plane will be configured on the target. A series of `table_adds` populate the table entries. State migration consists of a series of `register_read/writes` which copy the register values directly from the source to the target NF node. Finally, the `swap_configs` signals the moment when the new data plane would take effect on the target. The procedure of PM is similar to that of PR, but a bit simpler: a series of `table_adds` configure the table entries, and then `write_register` messages turn on and off the NF at the respective NF nodes. Because of packet buffering in BMv2, we do not lose any packet when we switch to a new configuration or change register values in a pipeline.

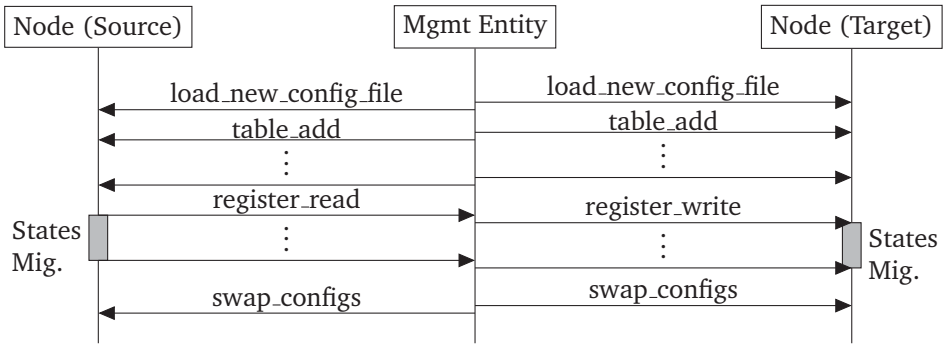


Figure 6.7: The message exchange and states of NF (on/off) during migration with PR from an initial to a target NF node. The management entity first loads new compiled P4 programs into the source and target nodes, followed by preparing respective table entries. Afterwards, a series of `register_read` and `register_write` migrate the necessary states from source to target node, before the new configuration is up running.

6.4.2 Consistent State Management with P4State Analyzer

After introducing the migration approaches, we address the next research question, i.e., how to decide which state to migrate. As mentioned in Section 6.2.3.3, the overall migration time scales with the total number of state variables that are migrated. Migrating all defined state variables would potentially incur longer latency and plague the network's flexibility. Accordingly, we propose P4STATE to analyze the register accesses of a P4 program and return the necessary state variables during data plane reconfiguration.

Instead of directly parsing the P4 code, we leverage the P4 compiler to produce a compiled `.json` file and feed it to the algorithm suite [171]. The basic idea is to perform control-flow analysis [160] and identify state accesses in tables and conditionals (Section 6.4.2.1 and 6.4.2.2), translate the P4 program into a CFG (Section 6.4.2.3), delete all nodes in the CFG without references to registers (Section 6.4.2.4), traverse all paths in the CFG to collect register accesses, and deduce the registers that need to be migrated (Section 6.4.2.5). The CFG represents all paths that might be traversed in a program during its execution.

6.4.2.1 Identifying States

As the first step, we collect all the declared registers (including their depths and widths) and classify them as *flow-based* or *device-based*. The classification criterion is the width of the register. As a common practice, a flow is identified by a hash value with width 16 or 32 [184], which corresponds to hashing algorithms defined in v1 model [205] (CRC16 or CRC32). Therefore, a register whose width is larger or equal than 16 is classified as *flow-based* register. In order to be comprehensive, we also classify the registers, whose indexes are calculated with hash functions, as *flow-based* registers. Those that are not classified as *flow-based* registers will be treated as *device-based* registers.

6.4.2.2 Table/Conditional Register-Binding

Algorithm 9 traverses all tables and if-conditionals defined in the `.json` file and associates the registers that are accessed by them. In order to do this, it first collects all defined registers, headers, actions, tables, and conditionals in the pipelines.

```
1   bit<16> scaler_flag;
2   ...
3   table my_table {
4   key = {}
5     actions = {my_action;}
6   }
7   action my_action() {
8     reg.read(scaler_flag, 0);
9   }
10  apply {
11    my_table.apply();
12  }
```

Figure 6.8: Statements within apply struct transformed into a table and an action.

Algorithm 9 Register Binding

Input: p4prog.json
Output: set of registers R, headers H, actions A, tables T and conditionals C

Fill R, H and A;
$T = \emptyset, C = \emptyset$;
Extract ingress/egress pipeline;
for t in pipeline **do**
  $R_t = \emptyset$;
  for a in A_t **do**
    for r in a_r **do**
      $R_t = R_t \cup \{r\}$;
    end for
  end for
  $T = T \cup \{t\}$;
end for
for c in pipeline **do**
  $R_c = \emptyset$
  for h in associated headers **do**
    Update set of registers R_c through h_r;
  end for
  $C = C \cup \{c\}$;
end for

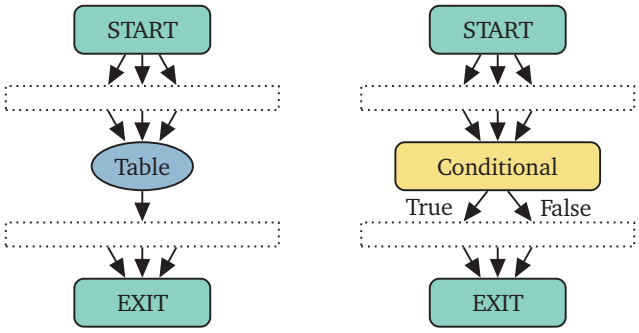
Algorithm 10 Explore Next Node (`ExpNext`)

Input: current node n, node list N
Output: Set of all paths

$N = N \cup n$;
if $n \in T$ **then**
  if next node of n is EXIT **then**
    $N = N \cup EXIT$;
    Call `DrawPath(`N`);`
  else
    for each node $\hat{n}$ after n **do**
      Call `ExpNext(`$\hat{n}$`,` N`);`
    end for
  end if
else
  for each next node $\hat{n}$ after n **do**
    if $\hat{n}$ is EXIT **then**
      $N = N \cup EXIT$;
      Call `DrawPath(`N`);`
    else
      Call `ExpNext(`$\hat{n}$`,` N`);`
    end if
  end for
end if

Since direct register access inside conditionals is not possible, it is non-trivial to associate registers to conditionals. For this purpose, we first collect all header fields (including the temporary variables), e.g., *flag* in Figure 6.1 and *reg_index* in Figure 6.2. Afterwards, we traverse all statements in actions and associate the header field with the register when there is a register access. Note that the statements in the *apply* struct are translated into an action associated with a table. For instance, line 5-6 in Figure 6.1 is transformed into Figure 6.8 by the compiler. The temporary variable *flag* is declared as a custom header field *scaler_flag*. The register read statement is placed inside an action *my_action*, which is the only associated action in a new declared table *my_table*.

Besides the set of all registers R, we also fill the set of accessed registers for each table t and conditional c. For a table t, we check all associated actions and place every accessed register in the set



(a) A table with one egress path.

(b) A conditional with two egress paths.

Figure 6.9: The basic elements of a CFG. The main difference between a table and a conditional is that a table has one egress edge, whereas a conditional has two.

R_t (see line 5-8 in Algorithm 9). Similarly, for a conditional c, we check all associated headers and place every accessed register in the set R_c.

6.4.2.3 Stateful CFG Construction

As mentioned before, the packet processing pipeline (i.e., the CFG) described by P4 can be decomposed as a bunch of basic entities (nodes) of tables and conditionals. The difference between a table and a conditional is demonstrated in Figure 6.9. A table has only one egress, whereas a conditional has two, each associated with a decision result (True/False).

We construct a CFG of the P4 program under analysis with Algorithm 12. Following the pattern of p4c-graphs [191], the processing path of each packet always starts from "START", traverses different sets of tables and conditionals, and terminates at "EXIT". The introduction of START and EXIT (as dummy tables) provides the two anchor points for all available processing paths. After adding the first edge between START and the initial node (table or conditional), it calls `ExpNext()` to further explore the path until EXIT, which is described in Algorithm 10.

Algorithm 10 works recursively: it stops calling itself only if *(i)* there is no node after the current table (line 3-5), or *(ii)* there are no entities on the true or false branch of the current conditional (line 11-13). In this case, it calls `DrawPath()` to draw a full path from START to EXIT.

6.4.2.4 CFG Pruning

The stateful CFG assists the following register accesses analysis. In our design, the analysis should be able to return all paths with state access. However, the original CFG and the paths inside can have an enormous size, which is hard for humans to consume. Inspired by the idea of *Thin Slicing* [175], we exclude all stateless nodes from the CFG, in order to produce a human-friendly (pruned) version. The pruned CFG provides an evident view of all stateful operations.

The pruning process consists of two steps (described in Algorithm 11 and 13). The first step detects all nodes that do not have access to registers, i.e., $n_R == \emptyset$ in line 5, and removes these nodes. The nodes before any node that need to be removed, i.e., $\overline{n}$, and after it, i.e., $\underline{n}$, should be reconnected to ensure complete path(s) from START to EXIT. The second step merges the following tables on a single

path if they access the same registers, i.e., $\underline{n}_R == n_R$. Only the first table stays, whereas the following tables are replaced with edges in the graph.

As function utilities, the method `UpdateNeighbourNodes()` updates the previous and subsequent nodes of each node, given the current status of the CFG. The method `RemoveNode()` removes one node and all edges connected to it.

Algorithm 11 Pruning – Stateless Node Elimination

Input: Original CFG
Output: Intermediate CFG

Call `UpdateNeighbourNodes()`;
for each node n in CFG **do**
  if n is START or EXIT **then**
    continue;
  end if
  if $n_R == \emptyset$ **then**
    Call `RemoveNode(CFG, n)`;
    for $\overline{n}$ in $\overline{N}_n$ **do**
      for $\underline{n}$ in $\underline{N}_n$ **do**
        Call `AddEdge(CFG, `$\overline{n}$`, `$\underline{n}$`)`;
      end for
    end for
  end if
  Call `UpdateNeighbourNodes()`;
end for
for each conditional c in C **do**
  if c in CFG **then**
    Get all paths P from START to c;
    for $p \in P$ **do**
      $\hat{R} = \emptyset$;
      for node n in $p[1:-1]$ **do**
        $\hat{R} = \hat{R} \cup n_R$;
      end for
      if $\hat{R} == \emptyset$ **then**
        Call `RemoveNode(CFG, c)`;
        for $\overline{n}$ of c **do**
          for $\underline{n}$ of c **do**
            Call `AddEdge(`$\overline{n}$`, `$\underline{n}$`)`;
          end for
        end for
      end if
    end for
  end if
end for

Algorithm 12 Graph Formulation (`GraphForm`)

Input: Set of R, H, A, T, C
Output: CFG

Initiate CFG as an empty directed graph;
Add node START and EXIT as two dummy tables;
Extract the first node n_1 from the pipeline;
Call `AddEdge(CFG, START, `n_1`)`;
$N = \{\text{START}\}$;
Call `ExpNext(`n_1, N`)`;

Algorithm 13 Pruning – Nodes Merging

Input: Intermediate CFG
Output: Pruned CFG

for each node n in CFG **do**
  for n is START or EXIT or not a table **do**
    continue;
  end for
  for next node $\underline{n}$ of n **do**
    if $\underline{n}$ is not a table **then**
      continue;
    end if
    if $\underline{n}_R == n_R$ **then**
      Call `RemoveNode(CFG, `$\underline{n}$`)`;
      for next node $\underline{\underline{n}}$ of $\underline{n}$ **do**
        Call `AddEdge(CFG, `n`, `$\underline{\underline{n}}$`)`;
      end for
    end if
    Call `UpdateNeighbourNodes()`;
  end for
end for

6.4.2.5 Path & Role Identification

Finally, the analyzer recognizes all paths with state access and generates the pruned CFG as well as a report listing all stateful paths and their respective associated state sets. When a P4 program consists of multiple functions, which are enabled/disabled upon startup (e.g., HULA [184]), the analyzer can also infer such information and report the enabled functionalities. For this, it leverages both the pruned CFG and the controller rules, such as register initialization (typically specified in a file). If the controller

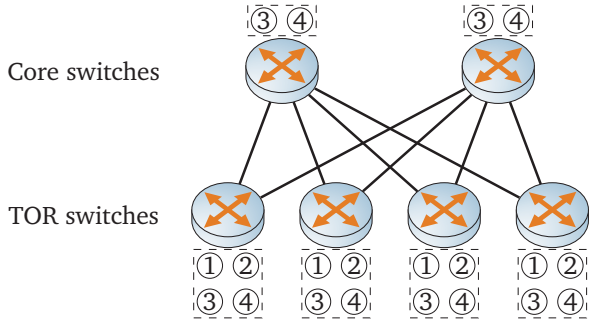


Figure 6.10: An exemplary topology of HULA with core and Top-of-Rack (ToR) switches. The circled numbers represent different types of registers.

maintains the state consistency, such information can also assist the decision of the order in which different types of states should be transferred.

6.4.3 Prototype and Evaluation

The prototype of P4STATE mainly includes a code analyzer and the utilities of the P4C compiler [191]. We implement the analyzer in Python. P4STATE takes a P4 program as input, analyzes its compiled `.json` format, and outputs the paths with state accesses.

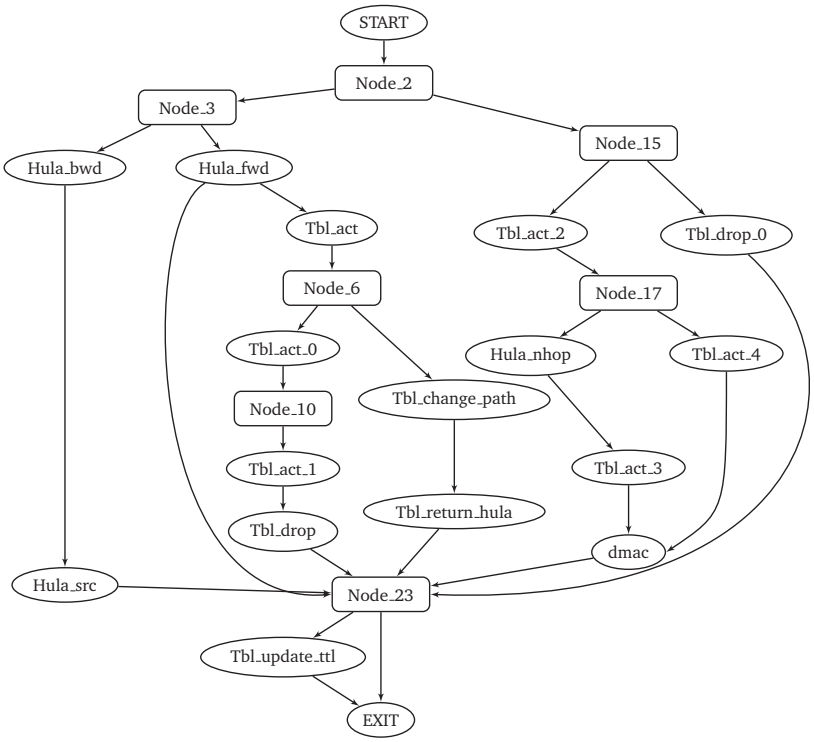
We provide an impression of P4STATE's practicability with a case study on two real P4 programs. Afterward, we evaluate the efficiency of our proposed algorithms with both real and synthetic programs.

6.4.3.1 Case Study: HULA

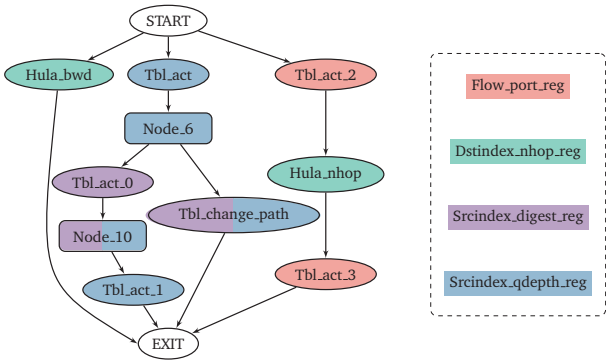
HULA addresses congestion in DC networks with two main functions, namely probing and forwarding. Figure 6.10 depicts an exemplary data center topology with HULA. Probing is deployed on the ToR switches for finding the best paths in the core. The probing function updates the forwarding function that forwards all data plane traffic afterward. The multi-rooted feature of data center networks enables a significant degree of multipathing between servers. As a data plane load balancing mechanism to utilize the full bisection bandwidth, instead of spreading traffic uniformly across multiple paths, HULA tracks congestion of all possible paths to a particular destination (through a neighboring switch), and forwards traffic via the best path; this path is dynamically updated with the help of probing packets.

`hula.p4` [184] is a simplified version of HULA with four types of registers: ① `srcindex_qdepth_reg`, ② `srcindex_digest_reg`, ③ `dstindex_nhop_reg` and ④ `flow_port`. ① and ② store the queue length and the digest of the best path from each ToR. ③ keeps the next hop to reach each ToR, and ④ keeps the next hop for each flow. In the case of HULA, there are two types of packets flowing in the network. The packets belonging to the first type help to maintain the best forwarding paths based on the queue build-up along each path. Their headers contain special fields to store the necessary information. The packets of normal forwarding do not have the special header fields. The packets of the first type are processed only at the ToR switches, whereas packets of the second type are processed at all switches.

`hula.p4` merges the pipeline of probing and the pipeline of forwarding in one program. To decide which pipeline should be referred to for a single packet, the program checks the *hula* header field of the received packet. For the ToR switches, all four types of registers are needed to enable HULA update and normal packets switching. For the non-ToR (i.e., core) switches, only type ③ and ④ are needed.



(a) Original CFG.



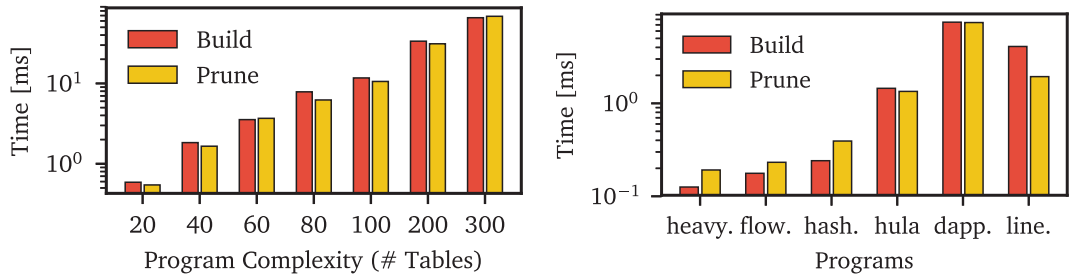
(b) Pruned CFG.

Figure 6.11: CFG of `hula.p4`: ellipses represent tables, squares represent conditionals, and arrows represent packet processing paths. Colors in (b) indicate the accesses of respective register types.

P4STATE successfully recognizes the above two functions, and outputs the pruned CFG with 12 nodes (original CFG 25 nodes), which is shown in Figure 6.11. Such knowledge can help to maintain state consistency during data plane reconfiguration. For example, when a core switch is about to overload, migrating states of type ③ and ④ to a backup switch would be sufficient to ensure that all current best paths in the core are preserved. The source code of HULA's ingress pipeline is listed in Appendix B.

6.4.3.2 Case Study: HashPipe

To perform line-rate measurements in the data plane, HashPipe [68] implements a pipeline of hash tables to record heavy flows, i.e., flows with a considerable number of packets. There are 8 types of



(a) Synthetic P4 programs.

(b) Real P4 programs.

Figure 6.12: Runtime of CFG construction and pruning.

registers that come within a total of 224 entries. The flows are tracked, and the tracking information is maintained within three types of registers (two stages, in total six). One type is used to track the flow identifiers (source IPv4 addresses), and another type is used to store the packet counts corresponding to the identifiers. The last type shows whether each table entry is valid, i.e., there are non-zero values for the previous two types of registers. P4STATE recognizes only one function, i.e., counter update, and all registers that needed to be migrated.

6.4.3.3 General Performance Measurement

To evaluate the efficiency of P4STATE, we measure the runtime of the CFG construction module and the CFG pruning model, which together account for the most algorithmic execution time. The measurement is performed on both synthetic and real P4 programs. We use Whippersnapper [67] to generate synthetic programs having 20 to 300 tables. The realistic programs are selected from Table 6.1.

Figure 6.12a presents the runtimes when analyzing synthetic programs; each bar denotes the average of 30 measurement runs. It shows comparable runtimes of CFG construction and pruning, which increase exponentially with the number of tables. Nevertheless, a program with 300 tables (which is more than all realistic P4 programs that we have collected in [21]) can be analyzed in around 100 ms. Figure 6.12b presents the results for real programs, which we sort according to the program's complexity (represented as LoC) along the x-axis. We observe that the code analysis takes up to 15 milliseconds for the CFG construction and pruning (the case of `dapper.p4`). Even though `linearroad.p4` has the highest LoC, its analysis time is not necessarily the highest, due to its simpler pipeline structure. Therefore, we show the efficiency of the P4STATE analyzer.

6.4.4 Discussions

Line-rate Processing and Verification. P4STATE outputs an overview of multiple register accesses in a P4 program. With that, we can identify when a read or a write operation is performed more than one time to the same register type, which can lead to longer processing times [68]. Moreover, the analyzer can automatically detect race conditions of register access and write-before-read error.

Group Transfer. Currently, the controller only accesses one register entry at a time. If multiple entries can be transferred simultaneously, the forwarding latency can be significantly reduced. In that case, P4STATE can be extended to detect the valid entries that will be transferred all at the once.

Counters & Meters. Since the data plane cannot read counters, the control plane reads and stores all values before reconfiguration, and if possible, updates the counted values afterward. For the meters, the control plane is always in charge of their configurations; therefore, the controller only needs to configure the previous meter settings upon the startup of a new data plane entity. P4STATE can be extended to recognize counters and meters and facilitate the maintenance of them during runtime.

6.5 Evaluation of Reconfiguration Performance

This section studies the performance overhead of reconfiguration based on measurements. It first evaluates a software prototype implementation of P4NFV and studies the overhead of service disruption and degradation. Afterwards, it evaluates a hardware P4 target and checks the reconfiguration latency, i.e., the time it takes until a control plane command takes effect.

6.5.1 Evaluation on Software Target

For the evaluation, we consider the following setup. Recently, strict latency requirements and high networking traffic overhead in the core drive the infrastructure providers to virtualize and locate NF nodes at the edge of the network infrastructure [143]. Such virtualization reduces the investment and expedites the infrastructure deployment process [69], [143]. Moreover, locating NFs in the proximity of the end-users also contributes to smaller end-to-end latency, which is critical for emerging applications like V2X [134] and tactile Internet [23].

However, the NF nodes deployed at the edge are still limited with resources and thus prone to overload in the face of highly dynamic traffic from end-users. P4NFV offers two opportunities of improvement: *(i)* P4 itself promises a better and more efficient hardware utilization [176], and *(ii)* P4NFV introduces data plane reconfiguration mechanisms that relocate the NF instances if possible.

6.5.1.1 Measurement Setup

We implement four types of NFs for the prototype demonstration: a *Packet Forwarder*, a *Network Address Translator (NAT)*, a *FireWall (FW)* and a *Load Balancer*. The *Packet Forwarder* forwards the packets based on destination MAC or IP addresses. It also modifies the source and destination MAC addresses for each packet. Its forwarding rules, stored as table entries, are populated with the *CMI* upon the startup of the network and remain static. The *NAT* translates IP addresses between private and public domains. The end-users and the core data centers typically have private network addresses, whereas the network entities in between use public network addresses [143]. The *FW* detects and blocks malicious flows that originate from the end-users. It works in a stateful manner: other than blocking flows based on static rules, it calculates fingerprints of flows and blocks the ones if the fingerprints violate the predefined policy. In our case, the fingerprint of flow is its packet inter-arrival rate. The FW drops packets of flows whose fingerprints are higher than a threshold.

We adopt a three-tier topology as presented in [177] and depict it in Figure 6.13. We differentiate three networking domains: access, aggregation, and core. The dark red color represents the core domain, the lighter color the aggregation domain, and the lightest red color the access domain. Circles labeled from S1-S5 denote the physical nodes hosting the P4-based network functions. Square nodes represent end-hosts and servers in the access and the core domain. The two nodes S1 and S2 in the access region are gateways that connect the end-user nodes H1-H3. Each gateway node is linked to the

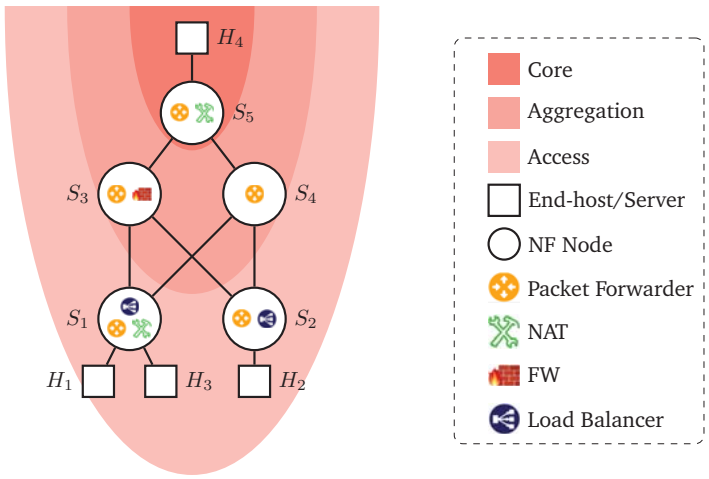


Figure 6.13: Topology setup of the use case study with five NF nodes and four types of NFs, namely Packet Forwarder, NAT, FW, and Load Balancer. Areas shaded with different color-intensity indicate cover nodes with different packet processing capacities.

two aggregation nodes S3 and S4. The aggregation node S5 accumulates traffic from end-users and forwards it to the core node, where H4 represents the server that the end-users want to connect to.

For our proof-of-concept implementation, we apply the BMv2 software switch [181] to host the NFs and build up the topology with Mininet. We use BMv2 CLI as the control plane and D-ITG traffic generator [178] to generate data plane flows. The evaluations are executed in an environment with Ubuntu 16.04, an Intel Xeon E3-1275v5 CPU of 3.6GHz, and 32 GB of RAM. For each evaluation scenario, we repeat 30 runs to gain statistical confidence.

6.5.1.2 Static Performance Analysis

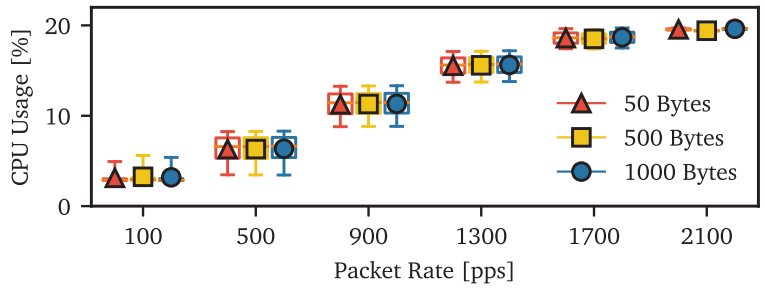
We first study the static performance by sending User Datagram Protocol (UDP) packets from H1 to H4 and analyze the impact of different pps and payload sizes (in Bytes). We instantiate one *Packer Forwarder* on node S1.

Packet Rate vs. CPU Usage

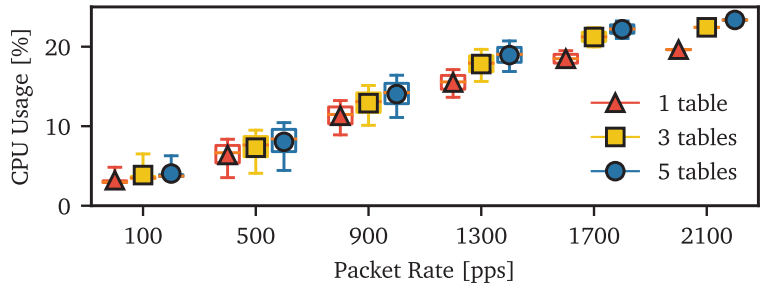
Figure 6.14a shows box-plots of the software switch's CPU utilization in percentage over increasing packet rates. The different shapes in the box-plots indicate the corresponding mean values. The CPU utilization increases with the packet rate, and the maximum mean value is around $19\,\%$. Whereas the CPU utilization increases with the packet rate, the different payload sizes (50, 500, and $1\,000$) only pose a marginal impact on CPU utilization. The observation of rate-dependent CPU utilization motivates the use of the reconfiguration mechanisms of P4NFV, i.e., migrating NF to a node with higher processing capacity.

Pipeline vs. CPU Usage

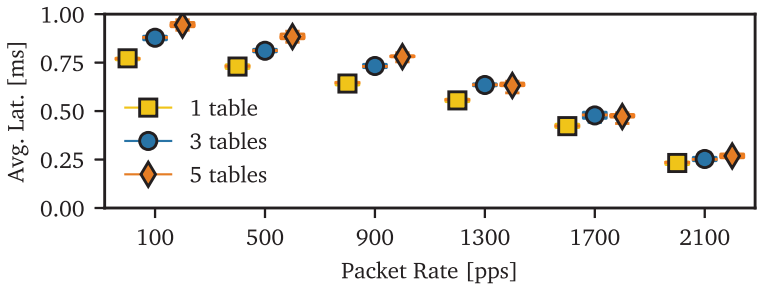
The implementation of `PM` can be realized with multiple tables. We now investigate whether the number of tables can impact the performance of NFs. We use the *Packet Forwarder* located on node S3. We have two alternatives for implementation. All actions can be either implemented in three tables (one table per step: decide output port, update source MAC, update destination MAC) or in one aggregated



(a) CPU Usage vs. packet rate.



(b) CPU Usage vs. the number of tables.



(c) Latency vs. the number of tables.

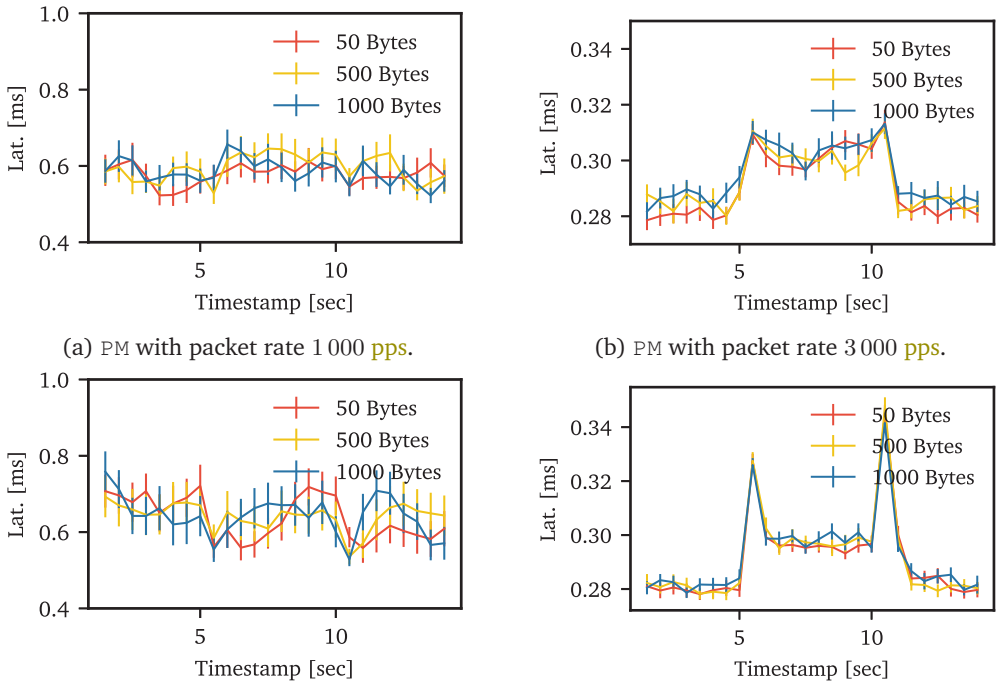
Figure 6.14: Figure 6.14a shows the relation between the CPU usage of different payload sizes and packet rates. Figure 6.14b shows the relation between the CPU usage of different packet rates and the number of table matches in the pipeline (payload size 50 Bytes). Figure 6.14c shows the relation between the average latency of different packet rates and the number of table matches in the pipeline (payload size 50 Bytes).

table. To elaborate on the overhead of additional tables (can be the case for the PM), we also analyze an implementation with two more dummy tables mimicking the behavior of an ACL filtering function. Figure 6.14b compares the CPU utilization of each scenario. For the same packet rate, more tables in the pipeline add a CPU utilization overhead, which is notable in the case of packet rates from 1 300 to 2 100.

Pipeline vs. Latency

As for the latency, Figure 6.14c reports nearly 20% higher average processing latency when we have five tables instead of one. The results confirm that an NF implementation with more tables induces higher resource utilization and processing latency. The latency and the previous CPU usage results demonstrate the potential overhead of PM, whereas with PR, multiple tables of different NFs can be compressed in favor of lower resource utilization and latency.

We also observe an interesting phenomenon that the latency decreases when the packet rate is higher. This happens because of the thread-based implementation of the software switch. In case of



(a) PM with packet rate 1 000 pps.

(b) PM with packet rate 3 000 pps.

(c) PR with packet rate 1 000 pps.

(d) PR with packet rate 3 000 pps.

Figure 6.15: Impact of NAT (stateless NF) migration on the packet forwarding latency, comparing two reconfiguration approaches and different packet rate. The NAT is instantiated at 5 second, and then migrated at 10 second.

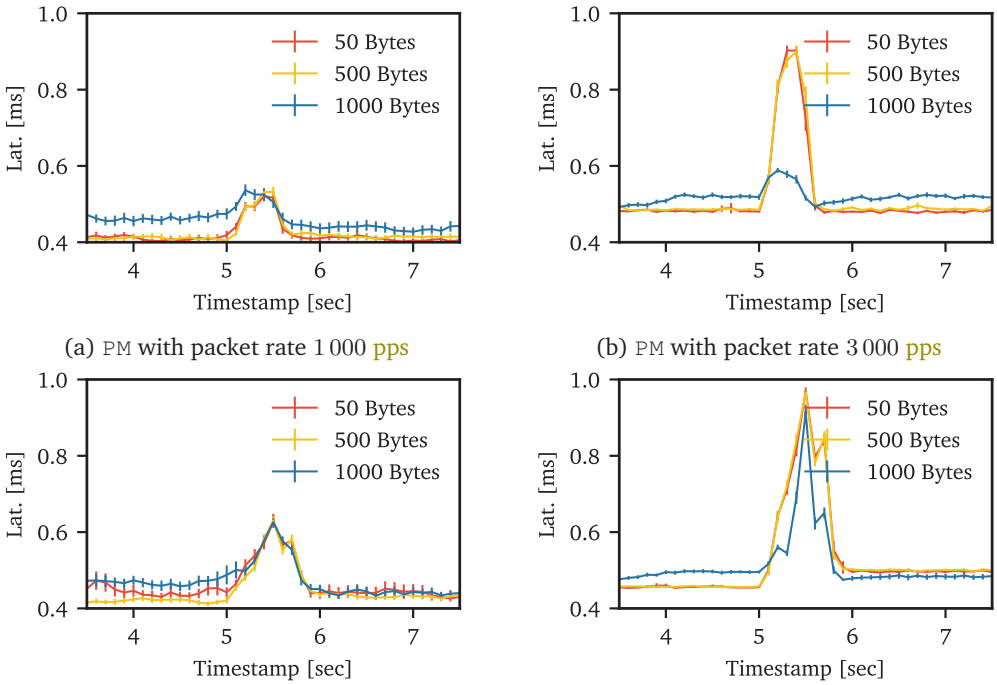
higher packet rates, the threads fall less asleep, and it takes less time until the threads wake up to process the packets, which contributes to shorter latency.

6.5.1.3 Stateless NF Migration

We choose the *NAT* as a representative for reconfiguration of a stateless NF. We send UDP traffic from H1 to H4, following the path S1-S3-S5. For each run, we first instantiate the NAT on S1 after 5 seconds and then migrate the NAT to S3 after another 5 seconds. For PM, we enable/disable the NAT tables through updating the binary value in the register, whereas for PR, we load different P4 programs with/without the NAT implementation, followed by populating the tables accordingly. We analyze two packet rates 1 000 pps and 3 000 pps for three payload sizes 50, 500, and 1 000 Bytes. The plots show the mean values and the 95% confidence intervals of 30 runs.

Impact on Functionality

We first examine the NF's functionality during migration and observe that all UDP packets successfully reach the destination. After dumping all the packets and checking their source IP addresses, we confirm that the IP addresses are modified correctly in all scenarios, i.e., no service disruption happens during NF migration. The BMv2 software switch can start working with the new intended packet processing pipeline immediately after the new configuration, e.g., the register values or P4 code is set via the *CMI*. However, this observation applies mainly to P4 software targets; for hardware targets, additional mechanisms such as buffering may be required to ensure minimal service disruption, i.e., latency increase and potential packet drops.



(a) PM with packet rate 1 000 pps

(b) PM with packet rate 3 000 pps

(c) PR with packet rate 1 000 pps

(d) PR with packet rate 3 000 pps

Figure 6.16: Impact of NF (stateful NF) migration on the concurrent traffic, comparing two reconfiguration approaches and different packet rate. The migration takes place at 5 second.

Impact on Latency

We measure the packet transmission latency from the source to the destination, which reflects the processing time of the NFs along the forwarding path. The results are reported in Figure 6.15. In general, the difference between different UDP payload sizes is not significant. For the packet rate at 1 000, the difference between PM and PR is marginal. When the packet rate increases to 3 000, peaks show up in the curves when reconfigurations happen. This overhead (more evident for PR) comes from longer queuing delays in the NF.

6.5.1.4 Stateful NF Migration

For the stateful NF, we evaluate a stateful FW that drops UDP packets that originate from a source IP with a sending rate higher than a threshold. The registers store the sending rates with indexes of hash values calculated from packet header 3-tuples (src. IP, dst. IP, and IP protocol number). They indicate whether a flow needs to be blocked. S1 and S2 host two stateless load balancers, which randomly forward packets to distribute the load on both links. When performing a migration, the *MOE* needs to configure the initial and target NF node in order to preserve the state consistency, e.g., keep dropping the packets of the blocked flows. *MOE* reads the register entries from source NF node S1 and then writes to the target S5.

In order to emulate both normal and malicious traffic, we create two UDP flows. The host H1 sends UDP traffic to H4 with a high packet rate — the traffic should be blocked by the FW. For the concurrent normal traffic, H2 sends 1 000 UDP packets per second to H4 with 50 Byte payload. For each run, we initiate the FW on S1 and then start the traffic generation. The UDP packets from H1 should be blocked, whereas the ones from H2 should always reach the destination. After that, the FW is migrated at time

Table 6.2: Comparison of the performance between P4NFV and a legacy NFV approach.

NF	Traffic	P4NFV	Legacy
NAT	500 pps, 7 500 packets	0 second	0.217 seconds (108.43 drops)
FW	500 pps, 7 500 packets	0 second	0.241 seconds (120.53 drops)

5 seconds from S1 to S5. We record the forwarding latencies of all packets that belong to the concurrent flows.

Impact on Functionality

We confirm that no FW service disruption happens during migration for any scenario, as no packets of the blocked flow reach the destination server. The state that indicates the blocking of H1's flow is copied from S1 to S5 before migrating the FW. Thus, the prototype can preserve the state information of the FW during its migration.

Impact on Latency

We do not observe any packet loss of the concurrent traffic. However, as illustrated in Figure 6.16, the migration indeed poses an impact on the forwarding latency. In general, `PR` introduces slightly higher latency (0.6 milliseconds) than `PM` (0.5 milliseconds), and `PR`'s performance degradation lasts 0.3 seconds longer than `PR`. Because of more packets buffering, the maximal delay during FW migration of 3 000 pps can be two times of 1 000 pps. Such delay can be alleviated by applying P4 targets that support parallel packet processing.

6.5.1.5 Comparison with a Legacy NFV Solution

Following the legacy NFV solution, we implement pure software-based NFs running inside VMs. We deploy the VMs in an OpenStack cloud and evaluate the performance during VM migrations. We instantiate three VMs, which act as the traffic source/sink and the packet processing entity, respectively. We implement the NAT and the stateful FW in Python with the Scapy library [206]. The NF migration makes use of the VM live-migration option of OpenStack [207].

Table 6.2 summarizes the induced data plane interruption of P4NFV with the legacy solution for the NAT and FW scenario. For the NAT scenario, a flow with 500 pps is generated for 15 s (7 500 packets to be transmitted in total). On average, 108.43 packets disappear during the migration of the NAT, which corresponds to service disruption of 0.217 seconds. For the FW scenario, two flows with 500 pps are generated for 15 s. Because the FW VM stores the state, there is no need to coordinate the state migration. Also, for the VM-based NF setting, no packets of the blocked flow reach the sink. However, the non-blocked flow experiences a similar packet loss as in the NAT scenario: on average, 120.53 packets are lost, which corresponds to service disruption of 0.241 s.

In contrast to the legacy solution, P4NFV can migrate both NFs without any service interruption. For the performance, we observe only a small latency increase during migration.

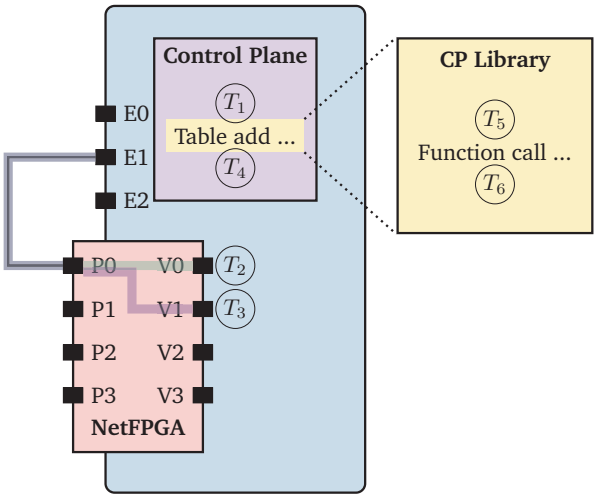


Figure 6.17: Measurement set-up of reconfiguration latency on the NetFPGA-SUME. The NetFPGA-SUME board is connected to the workstation via PCIe interface, and it offers four physical interfaces (P0 - P4) and four virtual interfaces (V0 - V3). The workstation has one 1 Gbps port and two 10 Gbps ports. Traffic is sent from E1 to P0 and then forwarded by the NetFPGA-SUME to V0, and to V1 after reconfiguration. Timing information are measured either by dumping the timestamps directly inside control plane programs, i.e., T_1, T_4, T_5, and T_6, or by checking the timestamps of the dumped packets, i.e., T_2 and T_3. The original traffic is illustrated in green from P0 to V0. After the reconfiguration, the traffic is redirected to V1, shown in purple.

6.5.2 Evaluation on Hardware Target

For the evaluation of hardware target, we decide to use NetFPGA-SUME (introduced in Section 2.2.4) because of its popularity in related research [15], [65]–[67]. We first evaluate the two reconfiguration approaches, i.e., `PM` and `PR`. Unfortunately, with the current setup and utility support, an old P4 configuration cannot be replaced without rebooting the host operating system which disrupts packet processing. Thus, the `PR` approach is not supported. The SimeSUMESwitch architecture (introduced in 2.2.4) of NetFPGA-SUME defines the register access as an extern function [208] instead of a primitive in the V1Model of BMv2 software switch; therefore, the P4 program under test needs to be adapted accordingly[2]. Nevertheless, we do not observe an increase in processing latency due to buffering during the reconfiguration. This is because NetFPGA-SUME processes packets with an internal pipeline which runs at the frequency of more than 100 MHz [209], while the maximum throughput is 40 Gbps that corresponds to 19.5 MHz for 64 Byte small packets and 0.8 MHz for 1518 Byte large packets. We indeed observe the difference in packet processing latency when the P4 pipeline changes, but only because the old and new pipelines have a different number of matching tables. More details of static performance measurement of P4 targets can be found in our work [15].

6.5.2.1 Measurement Setup

Similar to the previous works for OpenFlow switches [32], [179], we apply a measurement approach that evaluates NetFPGA-SUME's reconfiguration latency, which is a critical factor to consider while managing heterogeneous data plane devices. Figure 6.17 illustrates the measurement setup. The overall reconfiguration latency is defined based on the data plane effect delay [32]: it is the latency from sending the reconfiguration command until the effect is visible on the data plane. On the data plane,

[2]This slightly violates the "target-independence" feature of P4, which needs to be considered while migrating between different P4 targets.

Table 6.3: Reconfiguration scenarios.

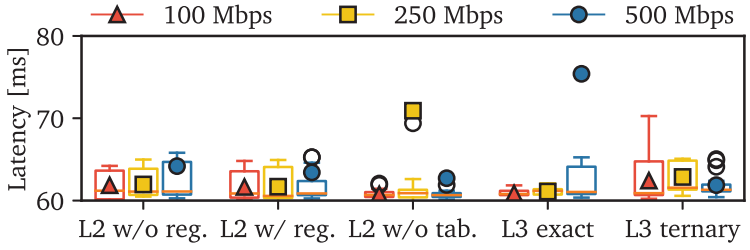
Scenario	Match On	Type of FIB
L2 w/o reg.	Ingress port	Exact table
L2 w/ reg.	Ingress port	Exact table & register
L2 w/o tab.	Ingress port	Register
L3 exact	Destination IP	Exact table
L3 ternary	Destination IP	Ternary table

traffic with constant packet size and data rate is sent initially from E1 to P0 and then forwarded to V0; after the reconfiguration, it is forwarded to V1. T_1 is the timestamp right before issuing the control plane command, and T_4 is the timestamp right after the command finishes. T_2 is the timestamp of the last data plane packet appearing on V0, while T_3 is the time of the first packet showing up on V1, which indicates the effect of reconfiguration. The control plane command is issued through the Python API [208], e.g., `Table add` in Figure 6.17. The command needs to experience a pre-processing procedure, e.g., with sanity check and address mapping, before calling the function defined in the control plane library and doing the reconfiguration. T_5 and T_6 are the timestamps before and after the function call in the library, respectively.

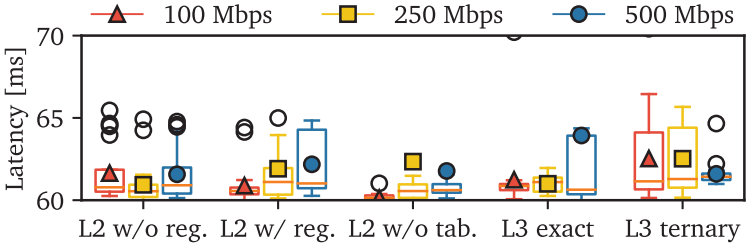
We mainly evaluate three timing characteristics: latency of control plane pre-processing (i.e., $T_5 - T_1$), latency of core reconfiguration (i.e., $T_3 - T_5$), and latency of overall reconfiguration (i.e., $T_4 - T_1$). The first one records the overhead at the control plane before actually issuing the command. The second one reveals the time it takes to execute the function call inside the control plane library, and the last one reflects the total time it needs before the next reconfiguration can take place sequentially.

There are five different methods to reconfigure the data plane, which we summarize in Table 6.3. It is worth mentioning that the second scenario represents the `PM` reconfiguration approach, only the two pipelines share the same functionality but with different table entries. The third scenario is useful when the forwarding information is kept in the register and can be updated by the forwarding device itself, e.g., in the case of path failover, the switch maintains port status with register [161]. By the time of this measurement, the control plane API for Longest Prefix Matching (LPM) table is still incomplete; therefore, we use the ternary table as the last scenario instead.

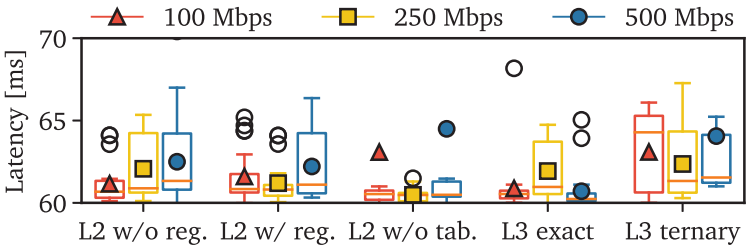
The measurements are performed in the same environment as in Section 6.5.1. We use three packet sizes, i.e., 256, 1 000, and 1 500 Byte. Regarding the throughput of the data plane traffic flow, even though the physical interface of NetFPGA-SUME supports up to 10 Gbps, its virtual interface has a capacity of 500 Mbps due to the bottleneck of the Direct Memory Access (DMA) RIFFA [210] driver implementation. Thus, we set the throughput at 100, 250, and 500 Mbps. Each scenario is measured for 20 times.



(a) Packet size: 256 Byte.



(b) Packet size: 1 000 Byte



(c) Packet size: 1 500 Byte

Figure 6.18: Control plane pre-processing latency of NetFPGA-SUME comparing different reconfiguration scenarios. Subplot (a), (b), and (c) shows the results of packet size 256 Byte, 1 000 Byte, and 1 500 Byte respectively.

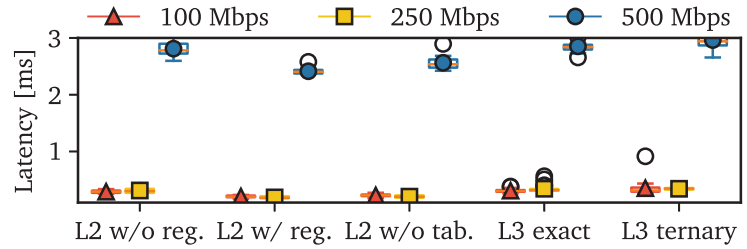
6.5.2.2 Measurement Results

Latency of Control Plane Pre-Processing

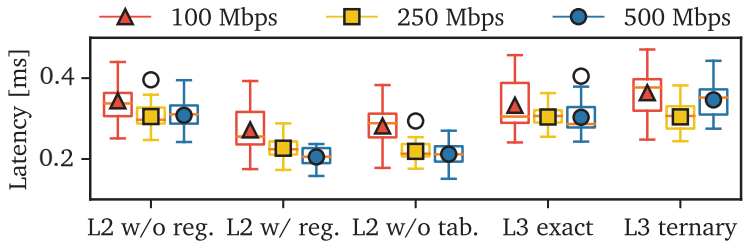
Figure 6.18 shows the measurement results for the latency control plane pre-processing. The latency values of each scenario are shown in a box-plot with the mean, median, 25 % and 75 % quartiles, and outliers. Note that the range of the latency data in 256 Byte (Figure 6.18a) is different from the other two cases (Figure 6.18b and Figure 6.18c). The median of most scenarios is around 61 milliseconds, and there is no significant difference between the median values of different throughput parameters. Some cases, e.g., L2 w/o tab. in 250 Mbps and L3 exact in 500 Mbps of 256 Byte packets, present huge outliers that push the mean values up. The mean latency of updating an entry in the L3 ternary table is higher than that of the other reconfiguration scenarios for packet size 1 000 and 1 500 Byte.

Latency of Core Reconfiguration

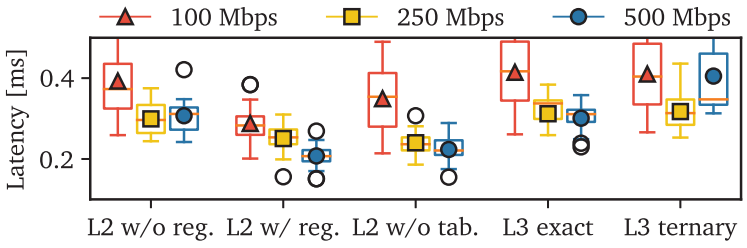
Compared with the pre-processing, the core reconfiguration takes much less time. The results are illustrated in Figure 6.19. Most scenarios have the mean and median values between 0.2 and 0.4 milliseconds. The only exception happens when the throughput is 500 Mbps, and the packet size is 256 Byte: the mean and median escalate to more than 2.5 milliseconds, which is around 10 times



(a) Packet size: 256 Byte.



(b) Packet size: 1 000 Byte



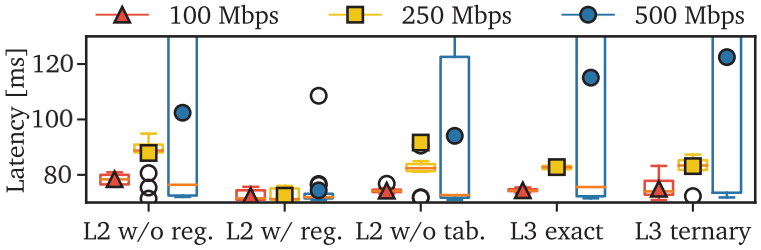
(c) Packet size: 1 500 Byte

Figure 6.19: Core reconfiguration latency of NetFPGA-SUME comparing different reconfiguration scenarios.

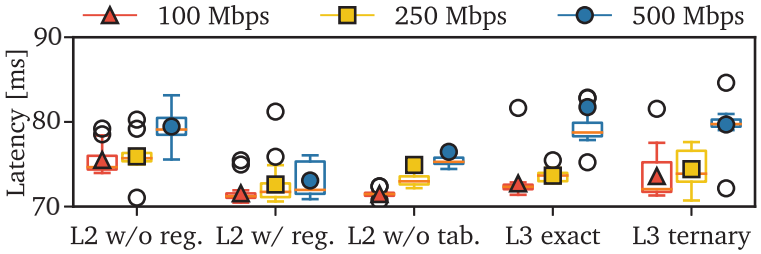
than the others and implies a bottleneck. The corresponding throughput and packet size combination indicates the highest packet rate, i.e., 244.1 Kpps. For lower packet rate, updating the register, i.e., L2 w/reg and L2 w/o tab., tends to take less time when the throughput increases. Furthermore, L3 ternary is more likely to consume longer time than its counterparts.

Latency of Whole Reconfiguration

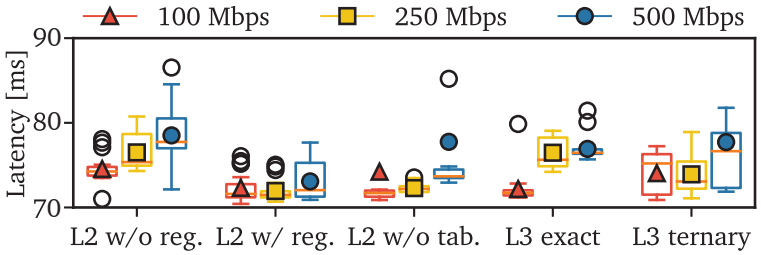
Figure 6.20 shows the measurement results for the overall control plane reconfiguration latency. In general, the overall latencies of most parameter combinations fall into the range between 70 and 80 milliseconds and is dominated by the pre-processing latency shown earlier. Again, the case of the highest packet rate shows many notable outliers. For large packet sizes, higher throughput tends to incur longer overall latency. To sum up, the overall reconfiguration latency of NetFPGA-SUME is comparable to that of many commercial SDN switches [32]. Because the core reconfiguration latency only takes a tiny part (less than 1 %) of the overall latency, it would be promising to rethink the design of control plane program, reduce the effort in pre-processing, and even parallelize multiple update commands, in order to achieve fast reconfigurations in the P4 data plane.



(a) Packet size: 256 Byte.



(b) Packet size: 1 000 Byte



(c) Packet size: 1 500 Byte

Figure 6.20: Single control plane reconfiguration latency of NetFPGA-SUME comparing different reconfiguration scenarios.

6.6 Summary and Discussion

P4 promises the advantages of protocol- and target-independence, which facilitates the design and deployment process of novel networking protocols and functions with high flexibility. Leveraging the data plane programmability of P4, we propose P4NFV, an architecture that is able to use hybrid infrastructure resources and reconfigure the network functionalities in the field. The structure of P4NFV complies with the recommendation of the standardization group, prompting its acceptance by network operators. To avoid potential data plane misbehaviors, the architecture preserves the consistency of stateful NFs during reconfigurations. Two approaches can achieve runtime reconfiguration with the consideration of network state management. The PM approach can include and exclude a particular part of a P4 program with the help of register values, whereas the PR approach swaps the old and new configurations completely.

As the next step, motivated by the need to adapt data plane functionality efficiently, we presented, implemented, and evaluated P4STATE, an automated mechanism to recognize states in P4, which is a critical prerequisite of the actual state migration process. P4STATE can quickly analyze programs and successfully recognizes the register types that need maintenance during data plane reconfiguration. The control-flow analysis on top of the CFG leverages the dependency of states in the definition of the

header, table, and actions, and can also be further extended to verify the state implementation of a P4 program. Use case study demonstrates that P4STATE can take a P4 program as input, collect state access along the packet processing pipelines, and visualize the analysis output for P4 programmers and operators. Besides, with synthetic and real programs, we show the efficiency of the proposed algorithms in terms of runtime.

Further, we demonstrate with our proof-of-concept implementation the live migration of NFs. We also measure the performance of a P4 data plane, especially during the reconfiguration. Static performance evaluations motivate the necessity of NF relocation in the face of dynamic traffic and the careful design of the pipeline structure. In comparison with a conventional VM solution, P4NFV ensures the liveness of functions without packet loss and acceptable performance degradation when migrating functions. We also evaluate the overhead of reconfiguration on the P4 data plane implemented in software and hardware. During the reconfiguration of a software P4 target, the packet processing latency increases for a short period due to the internal queue build-up. For the hardware P4 target, an increase of packet processing latency is not observed, but the overall reconfiguration latency – around 70 to 80 milliseconds – is still essential to consider.

Chapter 7

Conclusion and Outlook

Next-generation communication networks need to cope with frequent changes in user requirements, traffic distributions, service demands, and system anomalies. Accordingly, we have witnessed a boost of research efforts towards designing networks to accommodate those changes in a timely manner. The efforts are paid back with the emergence of networking techniques such as SDN and PDP, which flourish new research directions. Meanwhile, flexibility is often claimed to be one of the research proposals' advantage. The comparison between different proposals targeting the same networking problem in terms of flexibility is not accomplishable, because a common understanding is still missing in the networking research community. This gap also hinders the development of forthcoming novel techniques and concepts.

Network softwarization is the path adopted by both academia and industry, targeting the issues of legacy networking paradigm from both the control and data planes. SDN promises higher efficiency in terms of network deployment and management, because of the control and data plane separation and the global view of the network. Compared with the legacy networks, the control plane can make optimal decisions in the domain of networking traffic control, policy enforcement, and system reconfiguration and update. The control plane translates high-level policies into switch-level flow table entries and populated via open southbound interfaces, e.g., OpenFlow, leveraging the dynamic programmability of the forwarding devices. Therefore, the devices can focus on traffic processing, which decreases their design and manufacturing cost. SDN represents a fundamental paradigm shift towards the goal of softwarized networks from the control plane perspective.

PDP compensates one fundamental limitation of SDN, namely limited functionality that is exposed in protocols such as OpenFlow. Aided with domain-specific languages such as P4, PDP enables thorough programming of the data plane, fundamentally changing the way packets are processed on networking devices. Such programmability does not sacrifice the performance: recent chip designs have demonstrated the feasibility of Reconfigurable Match Table (RMT) structure to maintain Terabit throughput for P4 programs [56]. The alternative approach from the software world, in the meantime, also strives to leverage platforms such as Intel DPDK [192] to enable the implementation of sophisticated NFs described by P4 on commodity servers at line-rate up to tens of Gigabit throughput. Therefore, PDP, together with SDN, facilitate the goal of network softwarization and contribute to the design of communication networks that are powerful and flexible.

In light of the above open issues and opportunities, the previous chapters of this thesis make separate research contributions. This chapter concludes the thesis with the summary and discussion of the respective key research findings. We also outline several directions for future work.

7.1 Summary and Discussion

This thesis addresses the potential issues of managing control and data plane from both theoretical and practical aspects towards the target of flexible softwarized networks. The work is divided into four components and presented in Chapter 3, 4, 5, and 6. The first component proposes a formal definition of flexibility in the context of communication networks to enable a quantitative comparison between different network systems or design choices. The second component studies the deployment problem of the dynamic control plane for single and multi-period time-slots. The third component concentrates on the evaluation and optimization of the dynamic control plane. The last component explores the potentials of the data plane for runtime reconfiguration in terms of functionality.

Framework quantifying flexibility of communication networks. We interpret "flexibility" of a networking system as its timely support of changes in the network requirements with small cost. Topology, flows, node functions, and resources are recognized as the essential elements of a network that can reveal flexibility in various aspects. Formally, flexibility is defined as the relative ratio between the number of supported demand changes and the total number of changes under evaluation. The metric is mapped into a value in the range of $[0, 1]$. It reflects our intuition on flexibility and enables concise argumentation.

Deployment optimization of the dynamic control plane. By pushing the control plane remotely, SDN opens a new dimension of the resource optimization problem, i.e., making decisions of where to place the controllers and how to assign switches to the controllers. In the meantime, fast-changing data plane traffic flows introduce challenges in terms of control plane adaptation to maintain an acceptable flow setup performance. Accordingly, we propose single and multi-period DCPP: the former focuses on optimizing only for the flow setup performance, and the latter considers multiple time-slots – each with a different data plane flow profile – and maps the differences of the control plane states of two time-slots into cost, which is a first step to study the impact of adaptation. We design efficient heuristic algorithms and leverage ML techniques to accelerate the decision process. Numerical simulation results demonstrate the benefit of the dynamic control plane in terms of the average flow setup time, compared with the static control plane where the controller placement and switch assignment are fixed. Further, the trade-off between the flow setup performance and the adaptation cost is studied, which derives useful lessons-learnt for network operators.

Evaluating and optimization of the dynamic control plane. With the knowledge of how to evaluate the flexibility of a networking system, we can compare different control plane design choices from this new perspective. The adaptation time of the control plane is calculated as the time spent for controller migration and switch assignment. As the changing traffic triggers control plane adaptation, we define the demand changes as the data plane flow profiles. Numerical simulation results warn that the intuition of "more controllers lead to higher flexibility" is not always accurate under our definition. Indeed, with a strict adaptation time threshold, it can be difficult for multiple controllers to migrate. In contrast, a single controller may still be in the optimal location hence without the need to migrate. Moving one step further, we show the impact of DC locations on the flexibility of the dynamic control plane. If two

DCs locate far away from each other, it would be nearly impossible to migrate a controller in between. Accordingly, we propose a mathematical programming model that takes the flow profiles of different time-slot and optimize the number of supported profiles under predefined adaptation time and cost constraints. Evaluation based on simulation shows that it is possible to increase the flexibility value significant by planning the DC locations and the effectiveness of the proposed heuristic methods.

Programmable data plane targeting runtime reconfiguration. We first propose P4NFV, an NFV management architecture for the P4 data plane comprising of heterogeneous devices. The architecture applies two mechanisms to update the functionality, i.e., pipeline structure, of a particular P4 device at runtime, which can be useful for different reconfiguration scenarios. The need for adaptive networks implemented with a stateful programmable data plane also places the challenge of consistent and efficient reconfigurations. In this regard, we present P4STATE, a suite of algorithms that takes a P4 program as input, collects state access along the packet processing pipeline and visualizes the analysis output for P4 programmers and operators. The states are recognized by analyzing the CFG that reveals the logical structure of a P4 pipeline. With its analysis output, the following state maintenance mechanism can only keep track of the necessary state variables during reconfiguration. Data plane reconfiguration might show turbulence of its processing capability, hence brings in potential overhead. To study this overhead, we measure both software and hardware P4 targets in terms of performance metrics such as throughput and latency. The measurement result suggests that service disruption can be avoided entirely with a proper state synchronization mechanism. However, performance degradation, e.g., longer delay, is to be expected for software targets.

7.2 Future Work

The management of control and data plane in the context of network softwarization is still an active research area. We envision that the following research questions can be interesting for future work.

Application of data-automation and AI techniques for network resource management tasks. The tasks are naturally decision problems, no matter whether it is resource allocation for hybrid P4 data plane, or controller placement for the SDN control plane. Developing new solution techniques for decision problems should not only rely on classical algorithms. We shed some light on the applicability of ML algorithms on a simple use case of controller placement with a rather simple optimization objective. This methodology can be further promoted to cover other decision problems, as most of them face repetitive problem instances. Because of the frequent calling of algorithms, a tremendous amount of runtime data are generated, which makes big data analytics available. Besides, it would also be promising to move the training phase of ML algorithms from offline to online, which can better capture the input's dynamics of the decision problems, e.g., changing traffic distributions and update of topology.

Trade-off analysis between average flow setup time and other performance metrics. Although the average flow setup is one primary performance indicator of the SDN control plane, it can contradict other critical indicators such as resilience and reliability. Therefore, the controller placement problem can be extended to a multi-objective case to derive Pareto optimal placements concerning several objective functions.

Impact of graph features on flexibility. For the evaluation and optimization of the dynamic control plane, the numerical study is only performed on limited network topologies, which hinders conclusions

from a broader perspective. Since graph measures are one candidate to characterize network topologies, future work can dive deeper into the graph features of the respective topologies and develop knowledge to even predict the flexibility given a topology. The same methodology can be applied to other use cases that consider other flexibility aspects such as flow routing and topology adaptation.

Efficient state synchronization mechanism between heterogeneous P4 devices. The current P4NFV architecture proposal assumes a rather simple state synchronization that takes place after the explicit notification of functionality reconfiguration. While it works fine for NFs with a few state variables and low volume of inter-crossing traffic, seamless reconfiguration of state-abundant NFs without any temporary service malfunction is still hard to achieve. Efficient mechanisms need to be developed to leverage both control and data plane for state synchronization of different types.

Appendices

Appendix A

Proof of Proposition 1

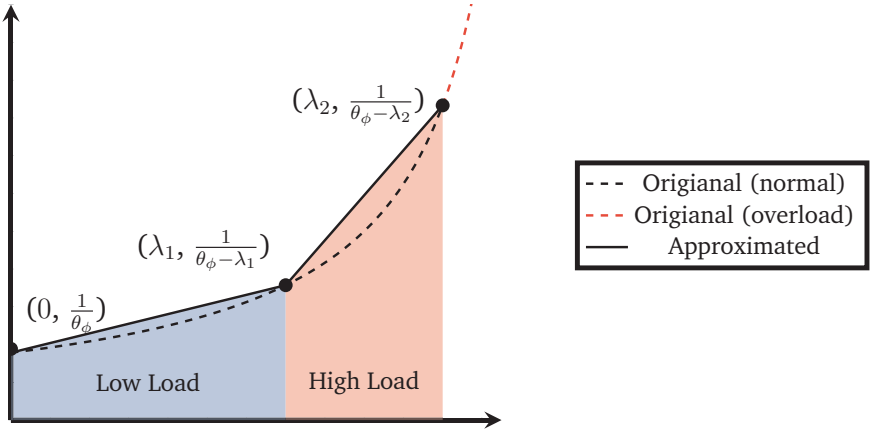


Figure A.1: Piece-wise linear approximation of the average controller sojourn time.

To approximate the curve, three anchor points are defined, namely $(0, \frac{1}{\theta_\phi})$, $(\lambda_1, \frac{1}{\theta_\phi-\lambda_1})$, and $(\lambda_2, \frac{1}{\theta_\phi-\lambda_2})$. The two line pieces connecting the three anchor points are expressed as follows.

$$f_1(\lambda) = \frac{1}{(\theta_\phi - \lambda_1)\theta_\phi}\lambda + \frac{1}{\theta_\phi}, \quad 0 \leq \lambda < \lambda_1 \tag{A.1}$$

$$f_2(\lambda) = \frac{1}{(\theta_\phi - \lambda_1)(\theta_\phi - \lambda_2)}\lambda - \frac{\lambda_1 + \lambda_2 - \theta_\phi}{(\theta_\phi - \lambda_1)(\theta_\phi - \lambda_2)}, \quad \lambda_1 \leq \lambda < \lambda_2 \tag{A.2}$$

In order to get the largest gap between the approximation and the real curves, we substrate $\frac{1}{\theta_\phi-\lambda}$ from (A.1) and (A.2), and calculate the first order derivative, e.g.,

$$\begin{aligned}(\Delta f_1(\lambda))' &= \Big[\frac{1}{(\theta_\phi - \lambda_1)\theta_\phi}\lambda + \frac{1}{\theta_\phi} - \frac{1}{\theta_\phi - \lambda}\Big]' \\ &= \frac{1}{\theta_\phi(\theta_\phi - \lambda_1)} - \frac{1}{(\theta_\phi - \lambda)^2}\end{aligned} \tag{A.3}$$

Let the derivative equal zero, we then have

$$\theta_\phi(\theta_\phi - \lambda_1) = (\theta_\phi - \lambda_l)^2$$
$$\lambda_l = \theta_\phi \pm \sqrt{\theta_\phi^2 - \theta_\phi \lambda_1} \tag{A.4}$$

We take the minus and do the same thing to (A.2)

$$\lambda_r = \theta_\phi \pm \sqrt{\theta_\phi^2 - \theta_\phi(\lambda_1 + \lambda_2) + \lambda_1\lambda_2} \tag{A.5}$$

$\max(\lambda_l, \lambda_r)$ returns the largest gap. In the worst-case, all flows need to issue flow setup requests in each control domain, and therefore, we have $\max(\lambda_l, \lambda_r) \cdot K$ for the approximation curve on top of the real one.

Appendix B

Source Code of Hula.p4

HULA [140] is a data plane load balancing mechanism for DC networks. A prototype has been developed with P4 [184], and we use it for our P4STATE analyzer in Section 6.4. The source code below only contains the logic of the ingress pipeline and is amended to save space.

```
1  control MyIngress(inout headers hdr,
2  inout metadata meta,
3  inout standard_metadata_t standard_metadata) {
4
5  /* At destination ToR, saves the queue depth of the best path from
6  * each source ToR
7  */
8  register<qdepth_t>(TOR_NUM) srcindex_qdepth_reg;
9
10 /* At destination ToR, saves the digest of the best path from
11 * each source ToR
12 */
13 register<digest_t>(TOR_NUM) srcindex_digest_reg;
14
15 /* At each hop, saves the next hop to reach each destination ToR */
16 register<bit<16>>(TOR_NUM) dstindex_nhop_reg;
17
18 /* At each hop saves the next hop for each flow */
19 register<bit<16>>(65536) flow_port_reg;
20
21 /* This action will drop packets */
22 action drop() {
23   mark_to_drop();
24 }
25
26 action nop() {
27 }
28
29 action update_ttl(){
30   hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
31 }
32
```

```
33 action set_dmac(macAddr_t dstAddr){
34   hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
35   hdr.ethernet.dstAddr = dstAddr;
36 }
37
38 /* This action just applies source routing */
39 action srcRoute_nhop() {
40   standard_metadata.egress_spec = (bit<9>)hdr.srcRoutes[0].port;
41   hdr.srcRoutes.pop_front(1);
42 }
43
44 /* Runs if it is the destination ToR.
45 * Control plane Gives the index of register for best path from a source ToR
46 */
47 action hula_dst(bit<32> index) {
48   meta.index = index;
49 }
50
51 /* On reverse path, update nexthop to a destination ToR to the ingress port
52 * where we receive hula packet
53 */
54 action hula_set_nhop(bit<32> index) {
55   dstindex_nhop_reg.write(index, (bit<16>)standard_metadata.ingress_port);
56 }
57
58 /* Read next hop that is saved in hula_set_nhop action for data packets */
59 action hula_get_nhop(bit<32> index){
60   bit<16> tmp;
61   dstindex_nhop_reg.read(tmp, index);
62   standard_metadata.egress_spec = (bit<9>)tmp;
63 }
64
65 /* Record best path at destination ToR */
66 action change_best_path_at_dst(){
67   srcindex_qdepth_reg.write(meta.index, hdr.hula.qdepth);
68   srcindex_digest_reg.write(meta.index, hdr.hula.digest);
69 }
70
71 /* At destination ToR, return packet to source by
72 * - changing its hula direction
73 * - send it to the port it came from
74 */
75 action return_hula_to_src(){
76   hdr.hula.dir = 1;
77   standard_metadata.egress_spec = standard_metadata.ingress_port;
78 }
79
80 /* On forward path:
81 * - if destination ToR: run hula_dst to set the index based on srcAddr
```

```
82 * - otherwise run srcRoute_nhop to perform source routing
83 */
84 table hula_fwd {
85   key = {
86     hdr.ipv4.dstAddr: exact;
87     hdr.ipv4.srcAddr: exact;
88   }
89   actions = {
90     hula_dst;
91     srcRoute_nhop;
92   }
93   default_action = srcRoute_nhop;
94   size = TOR_NUM_1; // TOR_NUM + 1
95 }
96
97 /* At each hop in reverse path
98 * update next hop to destination ToR in registers.
99 * index is set based on dstAddr
100 */
101 table hula_bwd {
102   key = {
103     hdr.ipv4.dstAddr: lpm;
104   }
105   actions = {
106     hula_set_nhop;
107   }
108   size = TOR_NUM;
109 }
110
111 /* On reverse path:
112 * - if source ToR (srcAddr = this switch) drop hula packet
113 * - otherwise, just forward in the reverse path based on source routing
114 */
115 table hula_src {
116   key = {
117     hdr.ipv4.srcAddr: exact;
118   }
119   actions = {
120     drop;
121     srcRoute_nhop;
122   }
123   default_action = srcRoute_nhop;
124   size = 2;
125 }
126
127 /* Get nexthop based on dstAddr using registers */
128 table hula_nhop {
129   key = {
130     hdr.ipv4.dstAddr: lpm;
```

```
131   }
132   actions = {
133     hula_get_nhop;
134     drop;
135   }
136   default_action = drop;
137   size = TOR_NUM;
138 }
139
140 /* Set right dmac for packets going to hosts */
141 table dmac {
142   key = {
143     standard_metadata.egress_spec : exact;
144   }
145   actions = {
146     set_dmac;
147     nop;
148   }
149   default_action = nop;
150   size = 16;
151 }
152
153 apply {
154   if (hdr.hula.isValid()){
155     if (hdr.hula.dir == 0){
156       switch(hula_fwd.apply().action_run){
157
158         /* if hula_dst action ran, this is the destination ToR */
159         hula_dst: {
160
161             /* if it is the destination ToR compare qdepth */
162             qdepth_t old_qdepth;
163             srcindex_qdepth_reg.read(old_qdepth, meta.index);
164
165             if (old_qdepth > hdr.hula.qdepth){
166               change_best_path_at_dst();
167
168               /* only return hula packets that update best path */
169               return_hula_to_src();
170             }else{
171
172               /* update the best path even if it has gone worse
173               * so that other paths can replace it later
174               */
175               digest_t old_digest;
176               srcindex_digest_reg.read(old_digest, meta.index);
177               if (old_digest == hdr.hula.digest){
178                 srcindex_qdepth_reg.write(meta.index, hdr.hula.qdepth);
179               }
```

```
180
181                 drop();
182               }
183             }
184           }
185         }else {
186           /* update routing table in reverse path */
187           hula_bwd.apply();
188
189           /* drop if source ToR */
190           hula_src.apply();
191         }
192
193       }else if (hdr.ipv4.isValid()){
194         bit<16> flow_hash;
195         hash(
196           flow_hash,
197           HashAlgorithm.crc16,
198           16w0,
199           { hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.udp.srcPort}, 32w65536);
200
201         drop();
202
203         /* set the right dmac so that ping and iperf work */
204         dmac.apply();
205       }else {
206         drop();
207       }
208
209       if (hdr.ipv4.isValid()){
210         update_ttl();
211       }
212     }
213 }
```

Bibliography

Publications by the author

Journal publications

[1] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Klügel, and A. M. Alba, "How to measure network flexibility? A proposal for evaluating softwarized networks," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 186–192, 2018. DOI: 10.1109/MCOM.2018.1700601.

[2] M. He, A. M. Alba, A. Basta, A. Blenk, and W. Kellerer, "Flexibility in softwarized networks: Classifications and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 3, pp. 2600–2636, 2019. DOI: 10.1109/COMST.2019.2892806.

[3] M. He, A. Varasteh, and W. Kellerer, "Towards a flexible design of SDN dynamic control plane: An online optimization approach," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1694–1708, 2019. DOI: 10.1109/TNSM.2019.2935160.

[4] A. Papa, T. De Cola, P. Vizarreta, M. He, C. M. Machuca, and W. Kellerer, "Design and evaluation of reconfigurable SDN LEO constellations," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1432–1445, 2020. DOI: 10.1109/TNSM.2020.2993400.

[5] P. Babarczi, M. Klügel, A. M. Alba, M. He, J. Zerwas, P. Kalmbach, A. Blenk, and W. Kellerer, "A mathematical framework for measuring network flexibility," *Computer Communications*, vol. 164, pp. 13–24, 2020. DOI: 10.1016/j.comcom.2020.09.014.

[6] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer, "P8: P4 with predictable packet processing performance," *IEEE Transactions on Network and Service Management*, 2020. DOI: 10.1109/TNSM.2020.3030102.

Conference publications

[7] M. He, A. Basta, A. Blenk, and W. Kellerer, "Modeling flow setup time for controller placement in sdn: Evaluation for dynamic flows," in *Proceedings of the 2017 IEEE International Conference on Communications (ICC)*, IEEE, 2017, pp. 1–7. DOI: 10.1109/ICC.2017.7996654.

[8] ——, "How flexible is dynamic SDN control plane?" In *Proceedings of the 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2017, pp. 689–694. DOI: 10.1109/INFCOMW.2017.8116460.

[9] M. He, P. Kalmbach, A. Blenk, W. Kellerer, and S. Schmid, "Algorithm-data driven optimization of adaptive communication networks," in *Proceedings of the IEEE 25th International Conference on Network Protocols (ICNP)*, IEEE, 2017, pp. 1–6. DOI: 10.1109/ICNP.2017.8117592.

[10] M. He, A. Basta, A. Blenk, N. Đerić, and W. Kellerer, "P4NFV: An NFV architecture with flexible data plane reconfiguration," in *Proceedings of the 2018 14th International Conference on Network and Service Management (CNSM)*, IEEE, 2018, pp. 90–98. [Online]. Available: https://ieeexplore.ieee.org/document/8584950.

[11] M. He, A. Blenk, S. Schmid, and W. Kellerer, "Toward consistent state management of adaptive programmable networks based on P4," in *Proceedings of the ACM Special Interest Group on Data Communication Workshop (SIGCOMM WKSP)*, ACM, 2019, pp. 1–7. DOI: 10.1145/3341558.3342202.

[12] M. Klügel, M. He, W. Kellerer, and B. Péter, "A mathematical measure for flexibility in communication networks," in *Proceedings of the IFIP Networking*, 2019. DOI: 10.23919/IFIPNetworking46909.2019.8999465.

[13] M. He, M.-Y. Huang, and W. Kellerer, "Optimizing the flexibility of SDN control plane," in *Proceedings of the 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, IEEE, 2020, pp. 1–9. DOI: 10.1109/NOMS47738.2020.9110474.

[14] M. He, A. M. Alba, M. Ehab, and K. Wolfgang, "Evaluating the control and management traffic in OpenStack cloud with SDN," in *Proceedings of the 2019 IEEE International Conference on High Performance Switching and Routing (HPSR)*, 2019. DOI: 10.1109/HPSR.2019.8807989.

[15] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer, "Towards understanding the performance of P4 programmable hardware," in *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, IEEE, 2019, pp. 1–6. DOI: 10.1109/ANCS.2019.8901881.

[16] A. Varasteh, S. Hofmann, N. Đerić, M. He, D. Schupke, W. Kellerer, and C. Mas Machuca, "Mobility-aware joint service placement and routing in space-air-ground integrated networks," in *Proceedings of the IEEE International Conference on Communications (ICC)*, IEEE, 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761265.

[17] A. Papa, T. De Cola, P. Vizarreta, M. He, C. M. Machuca, and W. Kellerer, "Dynamic SDN controller placement in a LEO constellation satellite network," in *Proceedings of the 2018 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2018, pp. 206–212. DOI: 10.1109/GLOCOM.2018.8647843.

[18] T. Li, H. Salah, M. He, T. Strufe, and S. Santini, "REMO: Resource efficient distributed network monitoring," in *Proceedings of the 2018 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, IEEE, 2018, pp. 1–9. DOI: 10.1109/NOMS.2018.8406239.

[19] A. M. Alba, P. Babarczi, A. Blenk, M. He, P. Kalmbach, J. Zerwas, and W. Kellerer, "Modeling the cost of flexibility in communication networks," in *Proceedings of the 2021 IEEE Conference on Computer Communications (INFOCOM)*, IEEE, 2021, pp. 1–10.

Others

[20] M. He, A. Blenk, A. Basta, and W. Kellerer, "Exploring runtime reconfigurability of P4 data plane," in *ACM CoNEXT 2018-Student Workshop*, 2018. [Online]. Available: https://mediatum.ub.tum.de/doc/1462426/229043.pdf.

[21] *Survey of Public Available P4 Programs*, https://github.com/muhe1991/p4-programs-survey.

General publications

[22] L. Tan and N. Wang, "Future Internet: The Internet of things," in *Proceedings of the 2010 3rd international conference on advanced computer theory and engineering (ICACTE)*, IEEE, vol. 5, 2010, pp. V5–376. DOI: 10.1109/ICACTE.2010.5579543.

[23] M. Simsek, A. Aijaz, M. Dohler, J. Sachs, and G. Fettweis, "5G-enabled tactile Internet," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 460–473, 2016. DOI: 10.1109/JSAC.2016.2525398.

[24] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5G wireless networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016. DOI: 10.1109/COMST.2016.2532458.

[25] M. Moradi, Y. Lin, Z. M. Mao, S. Sen, and O. Spatscheck, "Softbox: A customizable, low-latency, and scalable 5G core network architecture," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 438–456, 2018. DOI: 10.1109/JSAC.2018.2815429.

[26] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms," in *Proceedings of the 3rd Workshop on Hot topics in Software Defined Networking (HotSDN)*, 2014, pp. 121–126. DOI: 10.1145/2620728.2620746.

[27] G. Antichi and G. Rétvári, "Full-stack sdn: The next big challenge?" In *Proceedings of the 6th ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2020, pp. 48–54. DOI: 10.1145/3373360.3380834.

[28] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: a software-defined framework for developing, deploying, and managing network functions," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2016, pp. 511–524. DOI: 10.1145/2934872.2934875.

[29] M. A. Khan, S. Peters, D. Sahinel, F. D. Pozo-Pardo, and X.-T. Dang, "Understanding autonomic network management: A look into the past, a solution for the future," *Computer Communications*, vol. 122, pp. 93–117, 2018. DOI: 10.1016/j.comcom.2018.01.014.

[30] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet inter-domain traffic," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 75–86, 2010. DOI: 10.1145/1851275.1851194.

[31] A. Basta, A. Blenk, M. Hoffmann, H. J. Morper, K. Hoffmann, and W. Kellerer, "SDN and NFV dynamic operation of LTE EPC gateways for time-varying traffic patterns," in *Proceedings of the International Conference on Mobile Networks and Management*, Springer, 2014, pp. 63–76. DOI: 10.1007/978-3-319-16292-8_5.

[32] C. Sieber, R. Durner, and W. Kellerer, "How fast can you reconfigure your partially deployed SDN network?" In *Proceedings of the 2017 IFIP Networking Conference (IFIP Networking) and Workshops*, IEEE, 2017, pp. 1–9. DOI: 10.23919/IFIPNetworking.2017.8264845.

[33] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014. DOI: 10.1109/JPROC.2014.2371999.

[34] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *Proceedings of the 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, IEEE, 2018, pp. 1–7. DOI: 10.1109/HPSR.2018.8850761.

[35] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the 3rd Workshop on Hot topics in Software Defined Networking (HotSDN)*, ACM, 2014, pp. 1–6. DOI: 10.1145/2620728.2620744.

[36] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014. DOI: 10.1109/MM.2014.61.

[37] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, 2019. DOI: 10.1109/JSAC.2019.2894308.

[38] N. Yaseen, J. Sonchack, and V. Liu, "Synchronized network snapshots," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 402–416. DOI: 10.1145/3230543.3230552.

[39] T. Feng, J. Bi, and K. Wang, "Joint allocation and scheduling of network resource for multiple control applications in sdn," in *Proceedings of the 2014 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, IEEE, 2014, pp. 1–7. DOI: 10.1109/NOMS.2014.6838242.

[40] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *Proceedings of the 21st IEEE International Workshop on Local and Metropolitan Area Networks*, IEEE, 2015, pp. 1–6. DOI: 10.1109/LANMAN.2015.7114738.

[41] V. Kotronis, X. Dimitropoulos, and B. Ager, "Outsourcing the routing control logic: Better internet routing based on sdn principles," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets)*, 2012, pp. 55–60. DOI: 10.1145/2390231.2390241.

[42] G. Meyer, G. Adomavicius, P. E. Johnson, M. Elidrisi, W. A. Rush, J. M. Sperl-Hillen, and P. J. O'Connor, "A machine learning approach to improving dynamic decision making," *Information Systems Research*, vol. 25, no. 2, pp. 239–263, 2014. DOI: 10.1287/isre.2014.0513.

[43] Y.-K. Lin, H. Chen, R. A. Brown, S.-H. Li, and H.-J. Yang, "Healthcare predictive analytics for risk profiling in chronic care: A Bayesian multitask learning approach," *MIS Quarterly*, vol. 41, no. 2, 2017. DOI: 10.25300/MISQ/2017/41.2.07.

[44] S. J. Deodhar, M. Subramani, and A. Zaheer, "Geography of online network ties: A predictive modelling approach," *Decision Support Systems*, vol. 99, pp. 9–17, 2017. DOI: 10.1016/j.dss.2017.05.010.

[45] M. Bohanec, M. K. Borštnar, and M. Robnik-Šikonja, "Explaining machine learning models in sales predictions," *Expert Systems with Applications*, vol. 71, pp. 416–428, 2017. DOI: 10.1016/j.eswa.2016.11.010.

[46] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the First Workshop on Hot topics in Software Defined Networking (HotSDN)*, ACM, 2012, pp. 7–12. DOI: 10.1145/2377677.2377767.

[47] T. Y. Cheng, M. Wang, and X. Jia, "QoS-Guaranteed Controller Placement in SDN," in *Proceedings of 2015 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2015.7416960.

[48] E. Sakic, M. Avdic, A. Van Bemten, and W. Kellerer, "Automated bootstrapping of a fault-resilient in-band control plane," in *Proceedings of the 6th ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2020, pp. 1–13. DOI: 10.1145/3373360.3380829.

[49] D. Erickson, "The Beacon Openflow controller," in *Proceedings of the 2nd Workshop on Hot topics in Software Defined Networking (HotSDN)*, 2013, pp. 13–18. DOI: 10.1145/2491185.2491189.

[50] L. Yao, P. Hong, and W. Zhou, "Evaluating the controller capacity in Software Defined Networking," in *Proceedings of the 2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2014, pp. 1–6. DOI: 10.1109/ICCCN.2014.6911857.

[51] J. Li, X. Chang, Y. Ren, Z. Zhang, and G. Wang, "An effective path load balancing mechanism based on SDN," in *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, 2014, pp. 527–533. DOI: 10.1109/TrustCom.2014.67.

[52] L. Fawcett, S. Scott-Hayward, M. Broadbent, A. Wright, and N. Race, "Tennison: a distributed SDN framework for scalable network security," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2805–2818, 2018. DOI: 10.1109/JSAC.2018.2871313.

[53] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015. DOI: 10.1109/TNSM.2015.2401568.

[54] Z. Niu, H. Xu, L. Liu, Y. Tian, P. Wang, and Z. Li, "Unveiling performance of NFV software dataplanes," in *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, ACM, 2017, pp. 13–18. DOI: 10.1145/3155921.3158430.

[55] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proceedings of the 2nd Workshop on Hot topics in Software Defined Networking (HotSDN)*, 2013, pp. 127–132. DOI: 10.1145/2491185.2491190.

[56] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013. DOI: 10.1145/2534169.2486011.

[57] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014. DOI: 10.1145/2656877.2656890.

[58] T. K. Dangeti, R. Upadrasta, *et al.*, "P4LLVM: An LLVM based P4 compiler," in *Proceedings of the 2018 IEEE 26th International Conference on Network Protocols (ICNP)*, IEEE, 2018, pp. 424–429. DOI: 10.1109/ICNP.2018.00059.

[59] J. S. da Silva, F.-R. Boyer, L.-O. Chiquette, and J. P. Langlois, "Extern objects in P4: an ROHC header compression scheme case study," in *Proceedings of the 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, IEEE, 2018, pp. 517–522. DOI: 10.1109/NETSOFT.2018.8460108.

[60] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A programmable, protocol-independent software switch," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2016, pp. 525–538. DOI: 10.1145/2934872.2934886.

[61] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, and P. Shelar, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, vol. 15, 2015, pp. 117–130. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf.

[62] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A target-independent compiler for protocol-independent packet processors," in *Proceedings of the 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, IEEE, 2018, pp. 1–8. DOI: 10.1109/HPSR.2018.8850752.

[63] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4-¿ NetFPGA workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2019, pp. 1–9. DOI: 10.1145/3289602.3293924.

[64] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2018, pp. 576–590. DOI: 10.1145/3230543.3230559.

[65] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A rapid prototyping framework for P4," in *Proceedings of the 3rd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2017, pp. 122–135. DOI: 10.1145/3050220.3050234.

[66] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep: Towards in-network complex event processing," in *Proceedings of the ACM Special Interest Group on Data Communication Workshop (SIGCOMM WKSP)*, ACM, 2018, pp. 33–38. DOI: 10.1145/3229591.3229593.

[67] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A P4 language benchmark suite," in *Proceedings of the 3rd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2017, pp. 95–101. DOI: 10.1145/3050220.3050231.

[68] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the 3rd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2017, pp. 164–176. DOI: 10.1145/3050220.3063772.

[69] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford, "CLOVE: How I learned to stop worrying about the core and love the edge," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets)*, ACM, 2016, pp. 155–161. DOI: 10.1145/3005745.3005751.

[70] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2017, pp. 15–28. DOI: 10.1145/3098822.3098824.

[71] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2016, pp. 44–57. DOI: 10.1145/2934872.2934899.

[72] S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla, "Beyond fat-trees without antennae, mirrors, and disco-balls," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2017, pp. 281–294. DOI: 10.1145/3098822.3098836.

[73] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2019, pp. 367–379. DOI: 10.1145/3341302.3342090.

[74] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling programmable transport protocols in high-speed NICs," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, USENIX, 2020. DOI: https://www.usenix.org/system/files/nsdi20-paper-arashloo.pdf.

[75] T. Tao, *An introduction to measure theory*. American Mathematical Society, 2011, vol. 126, ISBN: 978-1-4704-1187-9. [Online]. Available: https://bookstore.ams.org/gsm-126.

[76] A. Baumgartner, V. S. Reddy, and T. Bauschert, "Combined virtual mobile core network function placement and topology optimization with latency bounds," in *Proceedings of the 2015 Fourth European Workshop on Software Defined Networks*, IEEE, 2015, pp. 97–102. DOI: 10.1109/EWSDN.2015.68.

[77] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance SDN switches," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2017, pp. 283–295. DOI: 10.1145/3143361.3143391.

[78] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Measuring control plane latency in SDN-enabled switches," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2015, pp. 1–6. DOI: 10.1145/2774993.2775069.

[79] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 323–334, 2012. DOI: 10.1145/2342356.2342427.

[80] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *Proceedings of the 2009 International Conference on Field Programmable Logic and Applications*, IEEE, 2009, pp. 99–104. DOI: 10.1109/FPL.2009.5272543.

[81] Y. Kanizo, O. Rottenstreich, I. Segall, and J. Yallouz, "Optimizing virtual backup allocation for middleboxes," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2759–2772, 2017. DOI: 10.1109/TNET.2017.2703080.

[82] A. Basta, A. Blenk, H. B. Hassine, and W. Kellerer, "Towards a dynamic SDN virtualization layer: Control path migration protocol," in *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, IEEE, 2015, pp. 354–359. DOI: 10.1109/CNSM.2015.7367382.

[83] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed Software Defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013. DOI: 10.1145/2486001.2486019.

[84] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "ElastiCon: an elastic distributed SDN controller," in *Proceedings of the ACM Symposium on Architectures for Networking and Communications Systems (ANCS)*, IEEE, 2014, pp. 17–27. [Online]. Available: https://ieeexplore.ieee.org/document/7856398.

[85] R. Ahmed and R. Boutaba, "Design considerations for managing wide area Software Defined Networks," *IEEE Communications Magazine*, vol. 52, no. 7, pp. 116–123, 2014. DOI: 10.1109/MCOM.2014.6852092.

[86] Z. Drezner and H. W. Hamacher, *Facility location: applications and theory*. Springer Science & Business Media, 2001, ISBN: 978-3-540-21345-1.

[87] T. Das, V. Sridharan, and M. Gurusamy, "A survey on controller placement in SDN," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, 2019. DOI: 10.1109/COMST.2019.2935453.

[88] A. K. Singh and S. Srivastava, "A survey and classification of controller placement problem in SDN," *Wiley International Journal of Network Management*, vol. 28, no. 3, 2018. DOI: 10.1002/nem.2018.

[89] M. T. I. ul Huque, W. Si, G. Jourjon, and V. Gramoli, "Large-scale dynamic controller placement," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 63–76, 2017. DOI: 10.1109/TNSM.2017.2651107.

[90] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann, "Heuristic approaches to the controller placement problem in large scale SDN networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 4–17, 2015. DOI: 10.1109/TNSM.2015.2402432.

[91] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic controller provisioning in Software Defined Networks," in *Proceedings of the 2013 9th International Conference on Network and Service Management (CNSM)*, IEEE, 2013, pp. 18–25. DOI: 10.1109/CNSM.2013.6727805.

[92] G. Wang, Y. Zhao, J. Huang, and Y. Wu, "An effective approach to controller placement in Software Defined Wide Area Networks," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 344–355, 2018. DOI: 10.1109/TNSM.2017.2785660.

[93] B. P. R. Killi and S. V. Rao, "Capacitated next controller placement in Software Defined Networks," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 514–527, 2017. DOI: 10.1109/TNSM.2017.2720699.

[94] T. Wang, F. Liu, and H. Xu, "An efficient online algorithm for dynamic SDN controller assignment in data center networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2788–2801, 2017. DOI: 10.1109/TNET.2017.2711641.

[95] M. Tanha, D. Sajjadi, R. Ruby, and J. Pan, "Capacity-aware and delay-guaranteed resilient controller placement for Software-Defined WANs," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, 2018. DOI: 10.1109/TNSM.2018.2829661.

[96] G. Yao, J. Bi, Y. Li, and L. Guo, "On the capacitated controller placement problem in Software Defined Networks," *IEEE Communications Letters*, vol. 18, no. 8, pp. 1339–1342, 2014. DOI: 10.1109/LCOMM.2014.2332341.

[97] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an OpenFlow architecture," in *Proceedings of the 23rd International Teletraffic Congress (ITC)*, IEEE, 2011, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/6038457.

[98] Y. Wang, Q. Zhong, X. Qiu, and W. Li, "Resource allocation for reliable communication between controllers and switches in SDN," *Springer Journal of Network and Systems Management*, pp. 1–27, 2018. DOI: 10.1007/s10922-018-9450-7.

[99] L. Yao, P. Hong, W. Zhang, J. Li, and D. Ni, "Controller placement and flow based dynamic management problem towards SDN," in *Proceedings of the 2015 IEEE International Conference on Communication (ICC)*, IEEE, 2015. DOI: 10.1109/ICCW.2015.7247206.

[100] J. Liao, H. Sun, J. Wang, Q. Qi, K. Li, and T. Li, "Density cluster based approach for controller placement problem in large-scale Software Defined Networkings," *Computer Networks*, vol. 112, pp. 24–35, 2017. DOI: 10.1016/j.comnet.2016.10.014.

[101] A. Sallahi and M. St-Hilaire, "Optimal model for the controller placement problem in Software Defined Networks," *IEEE Communications Letters*, vol. 19, no. 1, pp. 30–33, 2015. DOI: 10.1109/LCOMM.2014.2371014.

[102] M. L. Brandeau and S. S. Chiu, "An overview of representative problems in location research," *Management science*, vol. 35, no. 6, pp. 645–674, 1989. DOI: 10.1287/mnsc.35.6.645.

[103] P. Xiao, Z.-y. Li, S. Guo, H. Qi, W.-y. Qu, and H.-s. Yu, "A K self-adaptive SDN controller placement for wide area networks," *Frontiers of Information Technology & Electronic Engineering*, vol. 17, no. 7, pp. 620–633, 2016. DOI: 10.1631/FITEE.1500350.

[104] S. A. Curtis, "The classification of greedy algorithms," *Science of Computer Programming*, vol. 49, no. 1-3, pp. 125–157, 2003. DOI: 10.1016/j.scico.2003.09.001.

[105] F. W. Glover and G. A. Kochenberger, *Handbook of metaheuristics*. Springer Science & Business Media, 2006, vol. 57. DOI: 10.1007/978-1-4419-1665-5.

[106] F. Dunke, "Online optimization with lookahead," Ph.D. dissertation, Karlsruhe Institute of Technology, Jul. 2014. [Online]. Available: https://publikationen.bibliothek.kit.edu/1000042132.

[107] W. Kwon and A. Pearson, "A modified quadratic cost problem and feedback stabilization of a linear system," *IEEE Transactions on Automatic Control*, vol. 22, no. 5, pp. 838–842, 1977. DOI: 10.1109/TAC.1977.1101619.

[108] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Springer Cluster computing*, vol. 12, no. 1, pp. 1–15, 2009. DOI: 10.1007/s10586-008-0070-y.

[109] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006, ISBN: 978-1-4939-3843-8. [Online]. Available: https://www.springer.com/gp/book/9780387310732.

[110] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining*, vol. 3, no. 3, 2006. DOI: 10.4018/978-1-60566-058-5.ch021.

[111] M. A. Nielsen, *Neural networks and deep learning*. 2015, vol. 2018. [Online]. Available: http://neuralnetworksanddeeplearning.com/.

[112] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in Software-Defined Networks," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2012. [Online]. Available: https://www.usenix.org/system/files/conference/hot-ice12/hotice12-final33_0.pdf.

[113] P. Xiao, W. Qu, H. Qi, Z. Li, and Y. Xu, "The SDN controller placement problem for WAN," in *Proceedings of the 2014 IEEE/CIC International Conference on Communications in China (ICCC)*, IEEE, 2014, pp. 220–224. DOI: 10.1109/ICCChina.2014.7008275.

[114] Y. Xu, M. Cello, I.-C. Wang, A. Walid, G. Wilfong, C. H.-P. Wen, M. Marchese, and H. J. Chao, "Dynamic switch migration in distributed software-defined networks to achieve controller load balance," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 515–529, 2019. DOI: 10.1109/JSAC.2019.2894237.

[115] Y. Zhou, K. Zheng, W. Ni, and R. P. Liu, "Elastic switch migration for control plane load balancing in SDN," *IEEE Access*, vol. 6, pp. 3909–3919, 2018. DOI: 10.1109/ACCESS.2018.2795576.

[116] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011. DOI: 10.1109/JSAC.2011.111002.

[117] A. Nucci, A. Sridharan, and N. Taft, "The problem of synthetically generating IP traffic matrices: initial recommendations," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 3, pp. 19–32, 2005. DOI: 10.1145/1070873.1070876.

[118] A. Blenk, A. Basta, J. Zerwas, M. Reisslein, and W. Kellerer, "Control plane latency with SDN network hypervisors: The cost of virtualization," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 366–380, 2016. DOI: 10.1109/TNSM.2016.2587900.

[119] A. Blenk, P. Kalmbach, W. Kellerer, and S. Schmid, "o'zapft is: Tap your network algorithm's big data!" In *Proceedings of the ACM Special Interest Group on Data Communication Workshop (SIGCOMM WKSP)*, ACM, 2017, pp. 19–24. DOI: 10.1145/3098593.3098597.

[120] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit, "Local search heuristics for k-median and facility location problems," *SIAM Journal on computing*, vol. 33, no. 3, pp. 544–562, 2004. DOI: 10.1145/380752.380755.

[121] Z.-J. Zha, X.-S. Hua, T. Mei, J. Wang, G.-J. Qi, and Z. Wang, "Joint multi-label multi-instance learning for image classification," in *Proceedings of the 2008 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2008, pp. 1–8. DOI: 10.1109/CVPR.2008.4587384.

[122] M.-L. Zhang and Z.-H. Zhou, "A review on multi-label learning algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 8, pp. 1819–1837, 2014. DOI: 10.1109/TKDE.2013.39.

[123] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015. [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf.

[124] S. Nickel and F. S. da Gama, *Multi-period facility location*. Springer, 2015, pp. 289–310, ISBN: 978-3-319-13111-5. DOI: 10.1007/978-3-319-13111-5.

[125] K. Mahmood, A. Chilwan, O. Østerbø, and M. Jarschel, "Modelling of OpenFlow-based software-defined networks: The multiple node case," *IET Networks*, vol. 4, no. 5, pp. 278–284, 2015. DOI: 10.1049/iet-net.2014.0091.

[126] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *Proceedings of the IEEE International Conference on Cloud Networking (CloudNet)*, IEEE, 2015, pp. 255–260. DOI: 10.1109/CloudNet.2015.7335318.

[127] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of network and computer applications*, vol. 52, pp. 11–25, 2015. DOI: 10.1016/j.jnca.2015.02.002.

[128] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ACM, 2007, pp. 169–179. DOI: 10.1145/1254810.1254834.

[129] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13. DOI: 10.1145/3132211.3134460.

[130] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven SDN controller architecture," in *Proceedings of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, IEEE, 2014, pp. 1–6. DOI: 10.1109/WoWMoM.2014.6918985.

[131] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010. DOI: 10.1145/1773912.1773922.

[132] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski, "SNDlib 1.0: Survivable network design library," *Wiely Networks: An International Journal*, vol. 55, no. 3, pp. 276–286, 2010. DOI: 10.1002/net.20371.

[133] T. Braud, F. H. Bijarbooneh, D. Chatzopoulos, and P. Hui, "Future networking challenges: The case of mobile augmented reality," in *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017, pp. 1796–1807. DOI: 10.1109/ICDCS.2017.48.

[134] Z. Amjad, A. Sikora, B. Hilt, and J.-P. Lauffenburger, "Low latency V2X applications and network requirements: Performance evaluation," in *Proceedings of the 2018 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2018, pp. 220–225. DOI: 10.1109/IVS.2018.8500531.

[135] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy, "Flow processing and the rise of commodity network hardware," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 2, pp. 20–26, 2009. DOI: 10.1145/1517480.1517484.

[136] Z. Bronstein, E. Roch, J. Xia, and A. Molkho, "Uniform handling and abstraction of NFV hardware accelerators," *IEEE Network*, vol. 29, no. 3, pp. 22–29, 2015. DOI: 10.1109/MNET.2015.7113221.

[137] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and Software-Defined Networking," *Proceedings of the IEEE*, vol. 103, no. 7, pp. 1102–1124, 2015. DOI: 10.1109/JPROC.2015.2435732.

[138] H. Moens and F. De Turck, "Customizable function chains: Managing service chain variability in hybrid NFV networks," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 711–724, 2016. DOI: 10.1109/TNSM.2016.2580668.

[139] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 357–371. DOI: 10.1145/3230543.3230555.

[140] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable load balancing using programmable data planes," in *Proceedings of the 2nd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2016, p. 10. DOI: 10.1145/2890955.2890968.

[141] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 2, pp. 18–24, 2016. DOI: 10.1145/2935634.2935638.

[142] W. Shen, M. Yoshida, K. Minato, and W. Imajuku, "vConductor: An enabler for achieving virtual network integration as a service," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 116–124, 2015. DOI: 10.1109/MCOM.2015.7045399.

[143] A. Lombardo, A. Manzalini, G. Schembra, G. Faraci, C. Rametta, and V. Riccobene, "An open framework to enable NetFATE (Network Functions at the edge)," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, IEEE, 2015, pp. 1–6. DOI: 10.1109/NETSOFT.2015.7116179.

[144] S. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "NFVnice: Dynamic backpressure and scheduling for NFV service chains," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2017, pp. 71–84. DOI: 10.1145/3098822.3098828.

[145] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea, "Dynamic chaining of virtual network functions in cloud-based edge networks," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, IEEE, 2015, pp. 1–5. DOI: 10.1109/NETSOFT.2015.7116127.

[146] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[147] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, USENIX, 2014, pp. 459–473. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-martins.pdf.

[148] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112. [Online]. Available: https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf.

[149] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2014. DOI: 10.1145/2740070.2626317.

[150] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013. DOI: 10.1145/2534169.2486026.

[151] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, USENIX, vol. 13, 2013, pp. 227–240. DOI: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final205.pdf.

[152] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," vol. 44, no. 4, pp. 163–174, 2014. DOI: 10.1145/2619239.2626313.

[153] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic service chaining with Dysco," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2017, pp. 57–70. DOI: 10.1145/3098822.3098827.

[154] D. Hancock and J. Van Der Merwe, "Hyper4: Using P4 to virtualize the programmable data plane," in *Proceedings of the 12th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, ACM, 2016, pp. 35–49. DOI: 10.1145/2999572.2999607.

[155] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "HyperV: A high performance hypervisor for virtualization of the programmable data plane," in *Proceedings of the 2017 26th International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2017, pp. 1–9. DOI: 10.1109/ICCCN.2017.8038396.

[156] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018, pp. 98–111. DOI: 10.1145/3281411.3281436.

[157] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang, "SFC-Checker: Checking the correct forwarding behavior of Service Function chaining," in *Proceedings of the 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, IEEE, 2016, pp. 134–140. DOI: 10.1109/NFV-SDN.2016.7919488.

[158] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent updates for stateful and programmable data planes," in *Proceedings of the 3rd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2017, pp. 115–121. DOI: 10.1145/3050220.3050233.

[159] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2016, pp. 29–43. DOI: 10.1145/2934872.2934892.

[160] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, USENIX, 2016, pp. 239–253. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-khalid.pdf.

[161] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, and S. Schmid, "Supporting emerging applications with low-latency failover in P4," in *Proceedings of the ACM Special Interest Group on Data Communication Workshop (SIGCOMM WKSP)*, ACM, 2018, pp. 52–57. DOI: 10.1145/3229574.3229580.

[162] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2015, p. 5. DOI: 10.1145/2774993.2774999.

[163] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2017, pp. 29–42. DOI: 10.1145/3098822.3098825.

[164] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of TCP," in *Proceedings of the 3rd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2017, pp. 61–74. DOI: 10.1145/3050220.3050228.

[165] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé, "Life in the fast lane: A line-rate linear road," in *Proceedings of the 4th ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2018, p. 10. DOI: 10.1145/3185467.3185494.

[166] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, ACM, 2017, pp. 121–136. DOI: 10.1145/3132747.3132764.

[167] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, USENIX, 2017, pp. 97–112. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-kablan.pdf.

[168] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, "Automated synthesis of adversarial workloads for network functions," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, ACM, 2018, pp. 372–385. DOI: 10.1145/3230543.3230573.

[169] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance contracts for software network functions," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 517–530. DOI: https://www.usenix.org/system/files/nsdi19-iyer.pdf.

[170] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for P4 programs," in *Proceedings of the 4th ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, ACM, 2018, p. 5. DOI: 10.1145/3185467.3185497.

[171] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of P4 programs in feasible time using assertions," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018, pp. 73–85. DOI: 10.1145/3281411.3281421.

[172] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with Vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 518–532. DOI: 10.1145/3230543.3230548.

[173] K. Birnfeld, D. da Silva, W. Cordeiro, and B. de Franca, "P4 switch code data flow analysis: Towards stronger verification of forwarding plane software," in *Proceedings of the 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, IEEE, 2020, pp. 1–9.

[174] W. Wang, Y. Liu, Y. Li, H. Song, Y. Wang, and J. Yuan, "Consistent state updates for virtualized network function migration," *IEEE Transactions on Services Computing*, 2017. DOI: 10.1109/TSC.2017.2765636.

[175] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *ACM SIGPLAN Notices*, ACM, vol. 42, 2007, pp. 112–122. DOI: 10.1145/1250734.1250748.

[176] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 103–115. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-jose.pdf.

[177] M. Gao, B. Addis, M. Bouet, and S. Secci, "Optimal orchestration of virtual network functions," *Computer Networks*, vol. 142, 2018. DOI: 10.1016/j.comnet.2018.06.006.

[178] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre, "D-ITG distributed Internet traffic generator," in *Proceedings of International Conference on the Quantitative Evaluation of Systemss*, IEEE, 2004, pp. 316–317. DOI: 10.1109/QEST.2004.1348045.

[179] A. Van Bemten, N. Đerić, J. Zerwas, A. Blenk, S. Schmid, and W. Kellerer, "Loko: Predictable latency in small networks," in *Proceedings of the 15th International Conference on Emerging Networking EXperiments And Technologies (CoNEXT)*, 2019, pp. 355–369. DOI: 10.1145/3359989.3365424.

Miscellaneous

[180] *OpenFlow Switch Specification - Version 1.5.1*, https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, Accessed: 2020-03-21.

[181] *P4 behavioral-model*, https://github.com/p4lang/behavioral-model/, Accessed: 2018-02-20.

[182] *Netronome SmartNIC*, https://www.netronome.com/blog/p4-programmability-for-the-netronome-agilio-smartnic/, Accessed: 2018-02-20.

[183] *TOFINO: World's fastest P4-programmable Ethernet switch ASICs*, https://www.barefootnetworks.com/products/brief-tofino/, Accessed: 2020-03-21.

[184] *HULA*, https://github.com/p4lang/tutorials/tree/sigcomm_17/SIGCOMM_2017/exercises/hula, 2019.

[185] *Floodlight OpenFlow Controller*, http://www.projectfloodlight.org/floodlight/, Accessed: 2020-03-21.

[186] *ONOS - A New Carrier-Grade Network Operating System*, https://onosproject.org/, Accessed: 2020-03-16.

[187] *Home - OpenDaylight*, https://www.opendaylight.org/, Accessed: 2020-03-16.

[188] *OpenStack*, https://www.openstack.org/, Accessed: 2020-03-21.

[189] *OpenFlow Switch Specification - Version 1.0.0*, https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf, Accessed: 2020-03-21.

[190] *P4 16 Language Specification*, https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, Accessed: 2020-03-21.

[191] *P4C*, https://github.com/p4lang/p4c, 2019.

[192] *DPDK (Data Plane Development Kit)*, https://www.dpdk.org/, Accessed: 2020-03-21.

[193] *Mapping P4 to SmartNICs*, https://p4.org/assets/p4_d2_2017_nfp_architecture.pdf, Accessed: 2020-03-21.

[194] *Protocol Buffers*, https://developers.google.com/protocol-buffers, Accessed: 2020-03-21.

[195] *gRPC: A high-performance, open source universal RPC framework*, https://grpc.io/, Accessed: 2020-03-21.

[196] *P4Runtime Specification*, https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.html, Accessed: 2020-03-21.

[197] *Gurobi - The fastest solver*, https://www.gurobi.com/, Accessed: 2020-03-21.

[198] *CPLEX Optimizer — IBM*, https://www.ibm.com/analytics/cplex-optimizer, Accessed: 2020-03-21.

[199] *National population totals and components of change: 2010-2019*, https://www.census.gov/data/tables/time-series/demo/popest/2010s-national-total.html, Accessed: 2020-03-16.

[200] *Docker: Empowering App Development for Developers*, https://www.docker.com/, Accessed: 2020-04-06.

[201] *Linux KVM*, https://www.linux-kvm.org/page/Main_Page, Accessed: 2020-04-06.

[202] *Xen Project*, https://xenproject.org/, Accessed: 2020-04-06.

[203] *Network Functions Virtualisation (NFV); Architectural Framework v1.1.1 ETSI GS NFV 002*, http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/002/01.01.01_60/gs_NFV-002v010101p.pdf, Accessed: 2020-03-16, 2013.

[204] *P4-SDNet User Guide (UG1252) - Xilinx*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1252-p4-sdnet.pdf, Accessed: 2020-04-06.

[205] *V1 Model*, https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4, 2019.

[206] *Scapy Library*, https://github.com/secdev/scapy, Accessed: 2018-06-10.

[207] *OpenStack Instance Live Migration*, https://docs.openstack.org/nova/pike/admin/configuring-migrations.html, Accessed: 2018-06-10.

[208] *P4-NetFPGA Workflow*, https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview, Accessed: 2020-04-06.

[209] *SDNet Packet Processor*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf, Accessed: 2020-04-06.

[210] *RIFFA 1.0*, https://sites.google.com/a/eng.ucsd.edu/matt-jacobsen/riffa/riffa_1_0, Accessed: 2020-05-28.

Acronyms

ACL Admission Control List 18, 96, 97, 108

AI Artificial Intelligence 35, 121

ALU Arithmetic Logic Unit 14

API Application Programming Interface 10, 11, 13, 15, 16, 18, 19, 88, 92, 113, 157

AR Augmented Reality 85

ARP Address Resolution Protocol 11

ASIC Application-Specific Integrated Circuit 2, 17

BMv2 Behavior Model version 2 16

CFG Control Flow Graph 7, 91, 96, 99, 101, 102, 104, 105, 116, 121, 154

CLI Command-Line Interface 95, 107

CPP Controller Placement Problem 4, 32, 33, 78, 84

CPU Central Processing Unit 14, 17, 23, 64, 85, 87, 91, 94, 107, 108

DC Data Center 3, 4, 5, 6, 20, 21, 22, 27, 33, 63, 64, 72, 73, 77, 78, 80, 87, 103, 120, 121, 127

DCPP Dynamic Controller Placement Problem 4, 5, 6, 32, 33, 35, 37, 38, 40, 41, 43, 46, 50, 52, 60, 61, 63, 64, 66, 69, 72, 73, 76, 120, 153, 157

DMA Direct Memory Access 113

DPDK Data Plane Development Kit 14, 17, 87, 119

DPI Deep Packet Inspection 13, 85

DT Decision Tree 36, 48, 49

ETSI European Telecommunications Standards Institute 92, 95, 96, 154

FHC Fixed Horizon Control 34, 35, 55, 56, 57

FIB Forwarding Information Base 11, 113

FPGA Field-Programmable Gate Array 2, 17, 27

FSM Finite State Machine 15

FW FireWall 106, 107, 110, 111, 154

GDY Greedy 48, 50

GUI Graphical User Interface 95

I/O Input/output 87, 94

ILP Integer Linear Programming 2, 33

INT Inband Network Telemetry 85

IoT Internet of Things 1

JSON JavaScript Object Notation 15

LoC Lines of Code 17, 90, 105

LPM Longest Prefix Matching 113

LR Logistic Regression 36, 48

LS Local Search 34, 48, 49, 50

MAC Media Access Control 18, 96, 106, 107

MIP Mixed Integer Programming 5, 7, 33

ML Machine Learning 2, 6, 32, 35, 36, 37, 46, 48, 49, 50, 61, 120, 121, 153, 157

MPLS Multiprotocol Label Switching 31

NAT Network Address Translator 106, 107, 109, 111, 154

NF Network Function 6, 13, 20, 22, 23, 27, 28, 85, 87, 88, 89, 92, 93, 94, 95, 96, 97, 98, 99, 106, 107, 108, 109, 110, 111, 116, 117, 119, 122, 154

NFP Network Flow Processor 17

NFV Network Function Virtualization 7, 13, 21, 22, 23, 24, 29, 85, 86, 87, 92, 95, 96, 111, 121, 154, 157

NFVI Network Functions Virtualization Infrastructure 95

NN Neural Network 36, 48, 49, 50

NOS Network Operating System 2

NV Network Virtualization 21, 22, 23, 24, 29, 157

ODL OpenDaylight 11, 65

ONOS Open Networking Operating System 2, 11, 65

OSPF Open Shortest Path First 31

OvS Open vSwitch 17

P4 Programming Protocol-Independent Packet Processors 7, 14, 17, 18, 19, 27

PDP Programmable Data Plane 1, 2, 3, 5, 6, 7, 9, 18, 19, 20, 84, 85, 86, 87, 88, 119, 153

pps packets per second 87, 107, 109, 110, 111, 115

QoS Quality of Service 1, 3, 4, 21, 33, 86, 92

QP Quadratic Programming 33

RAM Random-Access Memory 17, 87

RHC Receding Horizon Control 34, 35, 55, 56, 57

RMT Reconfigurable Match Table 119

RTL Register-Transfer Level 17

RTT Round-Trip Time 5, 68, 69

S-D Source-Desitination 43

SA Simulated Annealing 33, 34, 56, 58, 61

SD-WAN Software Defined Wide Area Network 31, 37, 45, 50, 57, 67

SDN Software-Defined Networking 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 31, 37, 40, 61, 64, 85, 87, 115, 119, 120, 121, 153, 157

SLA Service Level Agreement 40, 92

TCAM Ternary Content-Addressable Memory 17

TCP Transmission Control Protocol 17, 88

TDG Table Dependency Graph 15

ToR Top-of-Rack 103, 154

UDP User Datagram Protocol 107, 109, 110

V2X Vehicle-to-everything 85, 106

VM Virtual Machine 53, 64, 65, 75, 87, 111, 117

VN Virtual Network 21

VNF Virtual Network Function 21, 87, 88, 95, 96

WAN Wide Area Network 11, 31, 58, 153

List of Figures

List of Tables