Software switch deployment in SDN-enabled network virtualization environment

Yimeng Zhao

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2016-ENST-0029

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité - Informatique et Réseaux

présentée et soutenue publiquement par

Yimeng ZHAO

le 24 05 2016

Déploiement du Switch Logiciel Dans SDN-enabled Réseau Environnement de Virtualisation

Directeur de thèse : **Michel Riguidel**
Co-encadrement de thèse : **Luigi Iannone**

Jury

M. Ken Chen, Professeur, University of Paris 13	Rapporteur
M. Vania Conan, Directeur de recherche, Thales	Rapporteur
M. Claude Chaudet, Ingénieur de recherche, Axon Square	Examinateur
M. Walid Dabbous, Directeur de recherche, INRIA	Examinateur
M. Damien Saucez, Researcher, INRIA	Invité
M. Luigi Iannone, Maître de conférences, Telecom Paristech	Directeur de thèse
M. Michel Riguidel, Professeur, Telecom Paristech	Directeur de thèse

T H È S E

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Software Switch Deployment in SDN-enabled Network Virtualization Environment

Yimeng ZHAO

Department of Computer and Networking
Telecom Paristech

Acknowledgements

I would like to show great appreciation and gratitude to my doctoral advisors, Michel Riguidel and Luigi Iannone. Michel's wisdom and experience always helped me in numerous ways of my work. His deep insight across literature, history, painting and music also greatly extends my vision during the past years. Luigi has set an example of an excellent researcher to me. He kindly provides opportunities for me in all related academical and industrial activities. I have been fortunate enough to work with them, and thank them for their enthusiasms, patience, encouragement and trust to my work.

I am grateful to Professor Ken Chen and Director Vania Conan as the reviewers of my thesis. Thanks for their constructive suggestions in the details. I also owe my thanks to Director Walid Dabbous, Researcher Claude Chaudet and Damien Saucez for their interest and participation in my thesis defense as the examinator.

It was a great pleasure to work at 23 avenue d'Italie where I could share plenty of moments, chats and stories from such a group of kind persons: Adel Sohbi, Danilo Cicalese, Mario Alvarado Ruiz, Karel De Vogeleer, Felipe Díaz Sánchez, Han Qiu, Yue Li, Wenqin Shao and Xiaoxing Yu. I also owe a debt of gratitude to my friends, Hao Cai, Pengwenlong Gu, You Wang, Mengying Ren and Kaikai Liu, who work at 46 Rue Barrault. I show my great appreciation to all these friends for their kindness and various support throughput my past years in Telecom ParisTech.

In fact, I started my doctoral studies in Shanghai JiaoTong University before I came to Paris. I had a good memory and met many nice friends there throughout the years. So special thanks go to my former supervisors, Professor Yue WU and Xinping GUAN.

Last, but not least, I appreciate all the support from my family during PhD studies. Especially from my mother who has her own doctorate, her understanding and encouragement are indispensable for me to accomplish my thesis work.

Yimeng ZHAO
Paris, France
May, 2016

Abstract

Due to the growing trend of "Softwarization", virtualization is becoming the dominating technology in data center and cloud environment. Software Defined Network (SDN) and Network Function Virtualization (NFV) are different expressions of "Network Softwarization". Software switch is exactly the suitable and powerful tool to support network softwarization, which is also indispensable to the success of network virtualization. Regarding the challenges and opportunities in network softwarization, this thesis aims to investigate the deployment of software switch in a SDN-enabled network virtualization environment.

In this thesis, we first focus on the evaluation of software switch as data plane. Two typical implementations of OpenFlow-enabled software switch, namely OpenvSwitch and OFsoftswitch, are selected for measurements. We carry out a systematic measurement that exploits various performance factors from both hardware and software comprehensively. In addition, we also provide insights on the implementation of in-band control and compare it with out-of-band control. Second, the controller is evaluated due to its critical role in the architecture of SDN. The evaluation is fair and fully reproducible. Moreover, our results are compared to previous works to have an insight on what has changed and why. Beyond simple use of benchmark tools, the impacts of system wide settings (e.g., the interpreter, Hyper-Threading technology) are also investigated. Besides centralized controller, a preliminary study on distributed controller is also carried out to investigate the synchronization traffic among multiple controller nodes. Third, we aim at fine-grained resource control on software switch. Instead of focusing on individual switch, the orchestration of multiple software switches from a global view is investigated. For this purpose, we implement a prototype of Service Function Chaining (SFC) architecture where multiple switches and service functions are required to coordinate with each other. A runtime is then proposed to support automated fine-grained resource allocation in SFC scenario. This runtime is not only effective in resource allocation among multiple software switches to improve the overall performance, but also capable to provide stable real-time bandwidth dynamically.

Résumé

Avec la prévalence de logicielisation, virtualisation est devenue une technologie dominante dans des data-centres et clouds. Deux aspects principaux de la logicielisation de réseaux sont Software Defined Network (SDN) et Network Function Virtualization (NFV), dont un des outils essentiel sont les switches logiciels, à l'opposition des switches matériaux. Les switches logiciels sont également indispensables pour le succès de NFV. Cette thèse vise à relever des défis principaux dans la logicielisation de réseaux. Spécifiquement, elle porte sur le déploiement des switches logiciels dans un réseau virtuel avec SDN.

Premièrement, nous avons évalué la performance de switch logiciel en tant que plan de transfert. Deux implémentations de switch logiciel, Openvswitch et OFsoftswitch, font l'objet de notre étude. Dans nos expérimentations, nous avons exploré des divers métriques de performance, ceux concernant les aspects logiciels aussi bien que ceux matériaux, d'une façon systématique. En plus, nous avons étudié le contrôle en bande de switch logiciel et l'avons comparé avec l'approche hors bande. Deuxièmement, le contrôleurs, en raison de son rôle stratégique dans un réseaux SDN, est étudié. Nous avons évalué ses réalisations différents d'une façon reproductible et impartiale. En outre, nous avons également comparé nos résultat avec ceux des travaux précédents, en donnant un aperçu sur les différences et des raison sous-jacentes. Au delà d'une simple exploitation des outils de benchmark, nous avons étudié les impacts de configurations globales de réseau, e.x. interprète et hyper-threading. Plus loin, nous avons étudié le scénario avec des contrôleurs distribués, à l'opposé de contrôleur centralisé qui est plus facile à implémenter mais aussi plus fragile. Le problème de synchronisation entre les multiples instances de contrôleur sont exploré est illustré. Troisièmement, nous visons à attribuer des ressources avec fine granularité à l'aide de switch logiciel. Au lieu de se limiter sur un seul switch, nous avons élaboré l'orchestration de plusieurs switches d'un perspective global. A cette fin, nous avons mis en œuvre un prototype de Service Function Chaining (SFC), où multiples switches et fonctionnalités réseaux s'échangent entre eux. Puis, nous avons également fournis une plateforme qui permet l'allocation des ressources automatisée avec fine granularité. Non seulement cette plateforme améliore la performance globale, mais elle est aussi capable de garantir une bande passante stable requise par des applications d'une manière dynamique.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

A growing trend of "Softwarization" is happening in almost every field of Information and Communication Technology (ICT). The virtualization of servers is now prominent in the majority of data centers. Similarly in network area, both Software Defined Network (SDN) and Network Function Virtualization (NFV) are different expressions of Software Defined Infrastructures (SDI) in an overall transformation trend of "Network Softwarization". Software switch is exactly the suitable and powerful tool to support SDN and NFV. Regarding the challenges and opportunities in network softwarization, this thesis aims to investigate the deployment of software switch in a SDN-enabled network virtualization environment. In our study, we first focus on the performance evaluation of mainstream OpenFlow-enabled software switches. Then a reality check on controller performance is carried out due to its importance in SDN framework. Finally, we propose an automated runtime framework to provide dynamic and adaptive resource allocation for software switches. In this section, a brief introduction of software switch and its promising deployment scenarios are provided.

1.1 Software switch

Software switches are emerging as one of the most promising solutions for data center and virtualized network infrastructure [125][66]. It is reported in [49] that over 70% x86 workloads in data center are virtualized and the adoption rate of server virtualization will rise to over 80% in the next 2 years. In the meanwhile, from a perspective of networking, the growth rate of virtual server access ports is faster than physical ports as shown in Fig. 1.1. As a result of this growing trend, over 2/3 of all server access ports are already virtualized in 2015. Over 40% network administrators adopt software forwarding mechanism (i.e., virtual switch) to manage network virtualization environment.

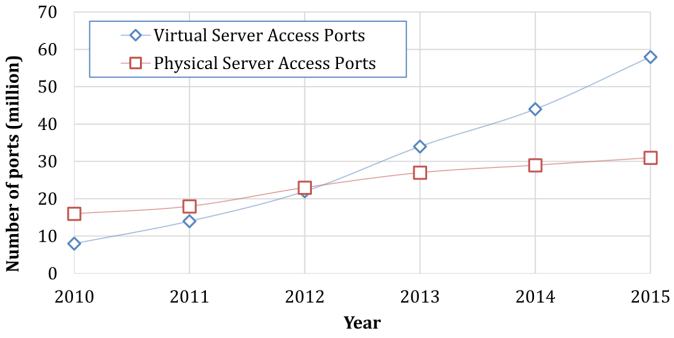


Fig. 1.1 The trend of virtualization for server access ports

1.1.1 Definition of software switch

Differently from the traditional definition in telecommunication [42] that refers to the central device or software used to connect telephone calls with different other phone lines, we define "software switch" as the virtual switch built on general purpose computer system to implement packet forwarding as well as other network functionality. More specifically, the general computer system in this thesis is limited to x86 architecture, and other platforms like ARM or FPGA are not considered. The typical implementations of software switch include Linux Bridge [4], Click Router [89], virtual switches in VMware ESX [123] or Microsoft Hyper-V [99], etc.

1.1.2 Software forwarding

Software forwarding approach has been developed for decades of years. Every server virtualization environment contains virtual switches that are used to connect multiple virtual machines. Linux bridge is this kind of virtual switch which has already been widely deployed with KVM [15]/QEMU [39] hypervisors. Click [89] is another well-known open-source software architecture for building flexible and configurable routers. Meanwhile, there are also good closed-source implementations from industry. VMware proposes vSphere Distributed Switch (VDS) for its ESX [123] environment in order to help facilitate the move to virtualization in data center. Cisco also develops an alternative solution for ESX called Nexus 1000V [7]. However, all these implementations either lack dynamic programmability and rich context for customized traffic control, or bind to specific platform without open-source support and general interface. In order to address these problems, SDN-enabled software switch is proposed to provide flexibility in customizing packet processing as well as standard control and management interfaces. OpenvSwitch [27] is a production quality,

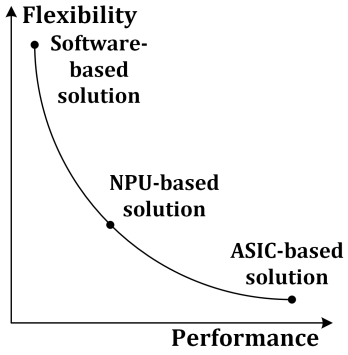


Fig. 1.2 Comparison among various architectures of packet forwarding

multilayer, SDN-enabled virtual switch under the open source Apache 2.0 license. It not only supports various protocols, including OpenFlow, NetFlow, sFlow, IPFIX, ets, but is also designed to support distribution across multiple physical servers similar to VMware's VDS or Cisco's Nexus 1000V. OpenvSwitch has been successfully combined in considerable products and deployed in large production environments (e.g., OpenStack [34]).

1.1.3 Comparison with other technologies

In order to build specialized packet processing platforms, a variety of technologies based on different kinds of hardware and software have been developed. These solutions can be mainly divided into three categories: Software, Application-Specific Integrated Circuit (ASIC) and Network Processor Unit (NPU). Their comparisons from various aspects are listed below.

- **Flexibility & Performance:** In ASIC-based solutions, the data plane is implemented with dedicated hardware which leads to high performance. However, ASICs lack flexibility to adopt new features and result in high cost for upgrading system. NPUs are originally expected to provide both high performance as ASIC-based solution and flexibility as software-based solution. They are specifically conceived for network function purposes and supposed to be used in parallel to allow workload balancing. However, NPU processing is only based on proprietary microcode which is a development hurdle in terms of available expertise and tool chain. Software-based solutions are built upon general-purpose CPU. Both control functionality and packet forwarding are performed by software. This provides great flexibility in introducing new features and value-added services with growing complexity. The deployment and upgrade of the system are also convenient due to software. But in the meanwhile, software forwarding introduces additional overhead in packet processing, which leads to lower

performance. Fig. 1.2 summarizes the comparison between three mentioned solutions on the basis of flexibility and performance.

- **Price:** When comparing price performance, we mainly focus on the switching at the edge. In edge scenarios, the switch should not only provides 10Gbps or higher bandwidth, but also supports a huge number of tunnels (N^2 in the number of N servers is quite common). In order for software switch to support 10Gbps, at least one CPU core is needed. Given a fairly modern CPU, one core weighs at around $100, and the motherboard and packaging support require additional $50. However, an ASIC hardware (e.g., a standard server NIC) around $150 is never recommended for high end scenarios. Similarly, NPU-based NICs are even more costly as much as 2-5 times of software solutions [108][45], which is largely due to supply chain issue and the immaturity of its market. Although dedicated hardware consumes less power than software forwarding in x86 [108], CPU resource assigned to software switch can be dynamically recycled according to workloads in order to lower total OPerational EXpenditure (OPEX).

- **Driver & Tool chain:** ASIC-based specific NICs have no or poor support of drivers for virtualization environment. NPU-based NICs are also rarely supported by mainstream virtualization platforms such as VMWare, Citrix, or common Linux distributions. Only software solutions rely on the underlying operating system and provide the appropriate drivers as well as configuration tools. Although the development tool for embedded NPU has come for a long way, it is still not comparable with the support of standard x86/Linux environment. Embedded development often leads to relatively specialized developers, expensive tools and complex debug environments. Furthermore, the openness of x86 environment stimulates the evolution of new development tools like DPDK [14], Netmap [21] and OpenOnload [33], which can significantly improve overall performance and reduce the complexity of development.

As summarized above, dedicated hardware provides higher performance as well as lower OPEX. While software forwarding achieves great flexibility in deploying and extending the system. It also motivates a series of open source projects for high performance packet processing based on commodity server. Although NPU is expected to be capable of both high performance and flexible processing, the market share of NPU is still not comparable with ASIC-based devices or software solutions even after a decade of development. This is mainly due to the lack of general support for hypervisor and middleware layers. Each solution has its own advantages and should be applied in suitable scenarios according to deployment requirements. For instance, dedicated hardware solution is more compelling in hosting the

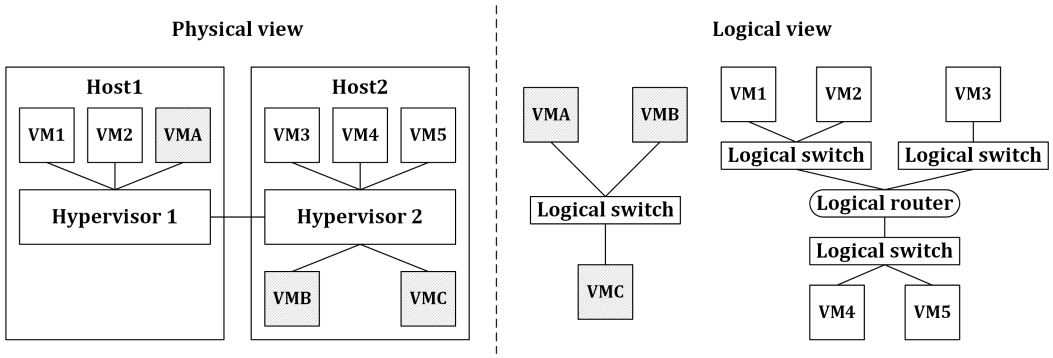


Fig. 1.3 Network virtualization

applications with small-sized and latency-sensitive traffic (e.g., voice and video). NPUs are usually adopted in middlebox to provide specific L4-L7 network functions with stateful processing such as Deep Packet Inspection (DPI). Software-based solution is dominant in data center and cloud environment which requires complex functionality at edge in order to support multi-tenant virtualization.

1.2 Software switch in network virtualization

Network virtualization allows the coexistence of multiple virtual networks that are sharing the same underlying physical infrastructures [70]. It separates traditional functionality of the network into two roles: *Infrastructure Provider* that builds physical network infrastructures and *Service Provider* that creates and manages virtual network as well as provides end-to-end network as a service. On the one hand, network virtualization is capable to combine multiple physical networks (or subnets) into one virtual network, e.g., VLAN. This simplifies the complexity of network management without modifying underlying infrastructures, which is treated as external network virtualization. On the other hand, internal network virtualization can break one physical network into several isolated virtual networks by using virtual nodes (virtual machines or containers) and virtual interfaces, which improves the utilization of physical resources. Both external and internal network virtualizations are integrated in a distributed system (e.g., data center or cloud) to provide an efficient, controlled, and secure sharing of the networking resources. Fig. 1.3 shows a typical network virtualization scenario. *VMA*, *VMB* and *VMC* are in the same subnet, while *VM1* to *VM5* belong to the other subnet. Although the VMs in the same subnet are running on different physical hosts, by using network virtualization, they can be managed in isolated logical networks through customized virtual topologies as shown in logical view.

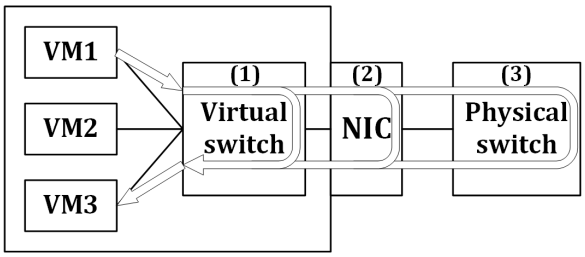


Fig. 1.4 VM traffic forwarding places

In order to implement network virtualization, it is crucial to forward VM traffic correctly and efficiently. There has long been an argument on the right place for inter-VM traffic forwarding. As shown in Fig. 1.4, in order to forward the traffic from VM1 to VM3, three potential places are marked: (1) virtual switch inside host, (2) host physical NIC and (3) first-hop physical switch. HP ProCurve [11] is a typical physical switch solution, requiring that each packet sent from VM should be tagged in order to be differentiated by physical switch. The tagging operation can be done either by the virtual switch in the hypervisor or by the NIC. The rationality for this approach is that dedicated switching hardware performs much faster than software. The downsides are also obvious: first, the bandwidth for inter-VM traffic is limited by the first-hop link. Second, it also consumes additional bandwidth of first-hop link and interferes with the traffic between VMs and the outside hosts because of hair-pinning method. Another proposal is to bypass the hypervisor and implement all inter-VM networking in NIC. However, switching in NIC has never been accepted by the market, because the performance of switching chipsets on NICs are not comparable to those used in standard switching gears. Furthermore, bypassing hypervisor obviates many advantages of virtualization. Software switching inside hypervisor seems to be the best choice, since it saves network resources (e.g., links or NIC) without offloading the decision to dedicated hardware (e.g., first-hop switch). Software switch essential fits in virtualization environment and can be easily integrated into any hypervisor or operating system.

As demonstrated in [67], there are more and more network deployment scenarios that are limiting the intelligence to network edge and keeping the core network simple. For instance in WANs, the inter-domain policies are usually implemented at the provider edge by applying MPLS. Similarly in network virtualization, a tunnel-based overlay is usually adopted in distributed data centers to provide connectivity across WAN. This overlay approach is compatible with existing IP-based Internet. All the encapsulation/decapsulation operations are implemented only at edge, i.e., the software switch. Various tunneling protocols have been proposed by Internet Engineering Task Force (IETF), including GRE, VXLAN, STT, LISP, GENEVE, etc. These protocols choose different layers for encapsulation and aim at specific

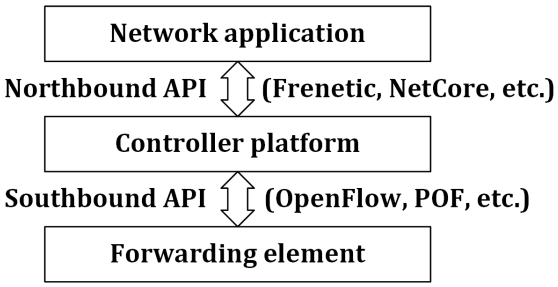


Fig. 1.5 Simplified SDN architecture

scenarios. Most of tunneling protocols are originally supported in Linux kernel, which can be easily integrated into software switch. Moreover, OpenvSwitch supports user-space tunneling in a platform independent way.

1.3 Software switch in Software Defined Network

The main idea of Software Defined Network (SDN) is to break the vertical integration by separating the networks control plane from the underlying data plane. SDN adopts a relatively centralized control plane and provides a global view of the whole network. A simplified view of SDN architecture is shown in Fig. 1.5. The forwarding element make forwarding decisions based on flow information (i.e., 12-tuple of header fields in OpenFlow 1.0) instead of traditional destination information. Any unmatched packet in forwarding element will be submitted to the controller for further instructions. The controller platform achieves direct control over the state of underlying forwarding elements by a well-defined programmable southbound interface. OpenFlow [31] is the most notable implementation of southbound API, and there are also other southbound protocols such as POF [118] and ForCES [124]. Various network applications are built upon controller platform through a northbound API. This API abstracts the low-level instruction sets from southbound and provides high-level programming functions for application developers. The typical northbound APIs include Frenetic [76], NetCore [100][115], etc.

As shown in Fig. 1.6, an OpenFlow-enabled switch consists of one or more flow tables used for packet classification and one or more OpenFlow channels for connecting external controllers. The switch and the controller communicates with each other via OpenFlow protocol. Each flow table contains a set of flow entries. Each flow entry consists of match fields, counters, and a set of instructions to apply to matching packets. Matching procedure starts at the first flow table and may continue to additional flow tables of the pipeline. Flow entries match packets in priority order, with the first matching entry in each table being used.

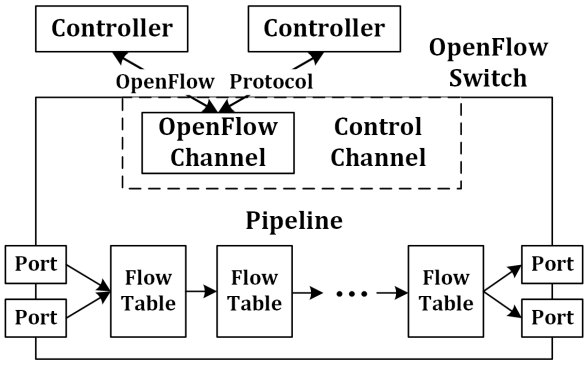


Fig. 1.6 Main components of an OpenFlow-enabled switch

If a matching entry is found, the instructions associated with this flow entry are executed. If no match is found in a flow table, the packet may be forwarded to the controllers, dropped, or to the next flow table. The controller can add, modify, or delete flow entries in flow tables.

Network virtualization has gained a new traction with the advent of SDN. Since network virtualization aims at decoupling network resources from underlying hardware, SDN offers a standard interface between controller applications and underling forwarding devices, which is a natural platform for network virtualization. For instance, Network Virtualization Platform (NVP) [91] is designed for multi-tenant data centers on the basis of SDN. NVP uses OpenvSwitch in all transport nodes (hypervisors, service nodes, and gateway) to forward packets. OpenvSwitch is remotely configurable by NVP controller to manage flow tables as well as overlay tunnels. Network virtualization has been one of the drivers behind the emergence of software switch. The SDN-enabled software switches are more promising to provide great flexibility in enabling network virtualization:

- The combination of SDN and software switch creates a pure software environment regardless underlying hardware infrastructures, which is convenient for the deployment, upgrading and migration of virtualization system.
- SDN-enabled software switch provides a unified and standard way to manage and configure the network virtualization dynamically.
- Software switch can follow the high-speed evolution of SDN. For instance, the OpenFlow specification is updated every year (from v1.0 in 2009 to v1.5 in 2014). Nowadays, nearly all hardware switches still only support specification v1.0, while software switches have already been running on v1.3 or v1.4.
- The overlay tunnels usually terminate inside virtual switches within hypervisors. SDN-enabled software switch can easily support hundreds of thousands of tunnels as well as provide customized network functionalities to enhance edge intelligence.

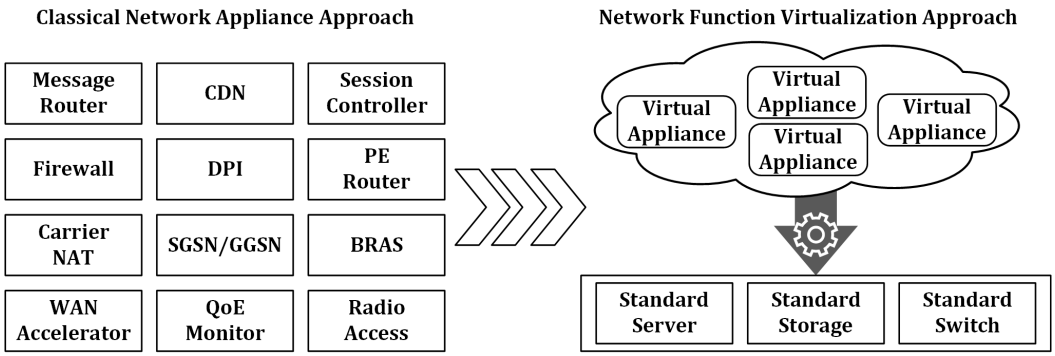


Fig. 1.7 Vision of Network Function Virtualization

1.4 Other promising scenarios

Besides SDN and network virtualization, software switch also finds its place in other emerging networking technologies such as Network Function Virtualization (NFV) and Network-as-a-Service (NaaS).

1.4.1 Network Function Virtualization

Network Function Virtualization (NFV) aims to transform the traditional way in deploying network functions by evolving standard virtualization technology to consolidate various types of network equipment or middlebox (e.g., Firewall, Load Balancer, WAN accelerator, etc.) onto standard high volume commodity servers [48][79]. As shown in Fig. 1.7, NFV implements network equipment or middlebox in the form of Virtual Network Functions (VNF) [50] that can be deployed and migrated to any location according to dynamic practical requirements. The typical use case of NFV is Service Function Chaining (SFC) [110][88] that aims to steer service-specific traffic to traverse multiple network functions in a given order. NFV and SDN are mutually beneficial and highly complementary to each other, but not dependent on each other. SDN can simplify NFV deployments, operations and migrations as well as enhance its performance. Meanwhile, NFV can provide underlying infrastructures for running SDN.

Software switch can be treated as the simplified VNF which only provides basic packet forwarding. Furthermore, a SDN-enabled software switch that supports fine-grained flow table control is capable to act as stateless Firewall or simple Load Balancer. Similarly to software switch, VNF based on standard commodity server also needs to address performance degradation problem, especially on heterogeneous hardware platforms. Any data plane acceleration technology (e.g., NAPI or DPDK) originally designed for software switch can be also applied in VNF to minimize the processing overhead and boost the performance. When

multiple VNFs are consolidated on the same server, software switches are used to forward the traffic among VNFs. Hence, the coordination between VNFs and software switches impact the overall performance, which should be considered and designed as a whole.

1.4.2 Network-as-a-Service (NaaS)

Network-as-a-Service (NaaS) [71] is proposed as a framework to integrate cloud environment with direct and secure access to the network infrastructure. NaaS allows tenants to customize forwarding decisions and network management based on application needs. Differently from traditional application-agnostic network services, by applying NaaS, the underlying network services are exposed and manageable by upper layer application, which improves the efficiency in implementing advanced network services, including in-network data aggregation, redundancy elimination, smart caching, etc.

Software switch built on commodity server has advantages in reducing costs, shortening update cycle and allowing rapid innovation. The flexibility of software switch in customization should be further extend for tenants to implement part of the application logic in cloud network. NetAgg [98][114] is a typical software forwarding platform that provides application-specific on-path aggregation in data centers.

1.5 Summary

Network softwarization is gradually and inevitably breaking the traditional vertical integration of vendor-specific network architecture, which allows multitude of services that could be created and provided through highly dynamic and borderless platforms of logical resources, fully decoupled from the underlying physical infrastructures. Software forwarding gains popularity as the basis for network softwarization, since it leads to costs savings, increased flexibility and programmability. Specifically, software switch, which has been widely deployed and tested in host virtualization environment, is a good start point to investigate the impact of software-based network virtualization combined with cutting-edge technologies such as SDN and NFV.

1.5.1 Main contributions

This thesis focuses on the deployment of software switch in SDN-enabled network virtualization environment. Our main contributions are listed as follows.

1. We carry out a systematic measurement on software switch performance in virtualization environment in order to overcome the shortcomings in previous partial benchmark tests. Our result highlights various performance factors from both hardware and software aspects. Numerical results are shown to evaluate their impacts, which is helpful to understand the limitation of software switch in deployment and point out the potential for improvement.

2. A comprehensive performance evaluation of five major open-source SDN controllers is provided. The measurements have been set up to be fair and easily reproducible. In order to provide a reality check beyond a simple benchmark, we not only examine general system wide settings, but also design specifically crafted scenarios for various measurement metrics. Our evaluation can be used as an indication of which controller is suitable in which scenario.

3. Aiming at the aspects ignored in previous studies, the design principle of SDN in-band control is explained in detail for the first time. The synchronization traffic among distributed controllers is also investigated for the first time. As a starting point, some preliminary results are provided to indicate future research directions.

4. We investigate the orchestration of multiple software switches on shared physical resources instead of focusing on the performance of single switch. In multiple-switch scenario, an automated runtime is proposed to support fine-grained resource allocation, which improves the overall performance. The runtime is also capable to provide stable real-time bandwidth in a Service Function Chaining scenario. Moreover, this runtime is a general framework that can be extended to support various resources and service-level objectives.

1.5.2 Thesis structure

The rest of this thesis document is structured as follows:

Chapter 2 first introduces mainstream open-source OpenFlow-enabled software switches. Then the related work on software switch deployment are classified and summarized based on SDN architecture. By analyzing the limitation of previous studies, we indicate our research directions and how they are supposed to overcome the existing shortcomings.

In Chapter 3, we focus on the evaluation of software switch as data plane. Two typical implementations of OpenFlow-enabled software switch, namely OpenvSwitch and OF-softswitch, are selected for measurements. Since former benchmark tests only rely on partial parameters or configurations, we carry out a systematic measurement that exploits various

performance factors from both hardware and software comprehensively. Quantitative results are used to evaluate the impact of factor. Combined with implementations, we discuss the cause of performance issues and reveal some clues in further improvement. For instance, our measurements and analysis prove that software switch performance is sensitive to traffic pattern, which can be further used to optimize the overall performance by properly customizing the topology and traffic. In addition, we also provide insights on the implementation of in-band control and compare it with out-of-band control.

We evaluate the controller in Chapter 4 due to its critical role in the architecture of SDN. In order to provide fair and reproducible results on controller performance, only open-source controllers and benchmark tools are chosen. Moreover, our results are compared to previous works to have an insight on what has changed and why. The evaluation is also beyond simple use of benchmark tools, the impacts of system wide settings (e.g., the interpreter, Hyper-Threading technology) are also investigated. Based on the outcome of the evaluation, it is helpful to derive some useful recommendations on which SDN controller is most suitable for which scenario. By combining the result of data plane in Chapter 3, the real bottleneck of the whole system in different scenarios can be identified. Besides centralized controller, a preliminary study on distributed controller is also carried out to investigate the synchronization traffic among multiple controller nodes.

Chapter 5 investigates fine-grained resource control on software switch. Different from previous works, we not only aim to guarantee the performance of individual switch, but also focus on the orchestration of multiple software switches within limited resources from a global view. For this purpose, we implement a prototype of Service Function Chaining architecture where multiple switches and service functions are required to coordinate with each other. Our case study demonstrates the effectiveness of suitable resource allocation among multiple software switches in improving the overall performance. Hence, we propose a runtime to support automated fine-grained resource allocation. The runtime is also capable to provide stable real-time bandwidth for Service Function Chaining scenario dynamically.

Last but not least, we draw the conclusion of current work in Chapter 6. Meanwhile, we also underline how the presented work can be deepened with further works.

Chapter 2

Background & Related Work

Since traditional software switches (e.g., CLICK, Linux Bridge, etc.) have been widely deployed in various industrial environments over a decade, their deployment issues are already well studied. The promising technologies including SDN and network virtualization further stimulate the development of software switch. A growing number of SDN-enabled software switches are expected in the near future. However, due to the immaturity, SDN raises questions regarding its performance and scalability. The switch that represents data plane in SDN plays an indispensable role for answering these questions. Hence understanding the performance and limitation of SDN-enable software switch is crucial to its success in deployment.

2.1 OpenFlow-enabled software switch

Several OpenFlow-enabled software switches are already available, while their usability differs from conceptual prototype to production quality. A selection of open source implementations are introduced as follows.

- **OpenvSwitch** [27]: OpenvSwitch is a open source, production quality, multilayer virtual switch. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols (e.g., NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag). In addition, it is designed to support distribution across multiple physical servers.

- **OpenFlow Reference Switch** [30]: This is the first prototype of OpenFlow-enabled software switch that provides OpenFlow switching capability to a Linux PC with multiple NICs. It defines the basic and necessary components to implement a OpenFlow switch. Many follow-up OpenFlow switch implementations are based on it.

- **OFsoftswitch** [29]: This is an OpenFlow v1.3 compatible user-space software switch that is intended for fast experimentation purposes. The code is based on the OpenFlow Reference Switch and the Ericsson TrafficLab v1.1 softswitch implementation, with modification to support OpenFlow v1.3.

- **Indigo** [13]: Indigo Virtual Switch (IVS) is an open source virtual switch for Linux. It is compatible with KVM hypervisor and leveraging OpenvSwitch kernel module for packet forwarding. IVS is built on the Indigo Framework and leverages LoxiGen generated code (loci) to handle OpenFlow messages.

- **LINC** [16]: LINC is a pure OpenFlow software switch written in Erlang, which is also implemented in user-space as an Erlang node. It provides great flexibility and allows quick deployment of new OpenFlow features by sacrificing the performance. It supports OpenFlow specification from v1.2 to v1.4 as well as OF-Config v1.1.1 management protocol.

- **OpenFlowClick** [32]: OpenFlowClick encapsulates OpenFlow Reference Switch as a Click element, which can be connected to other elements to reuse their functionality. This design combines OpenFlow forwarding control mechanisms with per-packet processing capability of Click.

- **Pantou** [37]: Pantou turns a commercial wireless router/Access Point to an OpenFlow-enabled switch. OpenFlow is implemented as an application on top of OpenWrt. Pantou is based on the BackFire OpenWrt release (Linux 2.6.32). The OpenFlow module is based on OpenFlow Reference Switch.

According to above introductions, all software switches are either based on OpenvSwitch or OpenFlow Reference Switch except LINC, which indicates that the original designs of OpenFlow-enabled software switches are few. Table 2.1 further summaries the main characteristic of selected software switches. Nearly all implementations choose C as programming language to guarantee high performance. Erlang is used only by LINC to provide high flexibility but sacrifice the performance. For maintenance, OpenvSwitch and Indigo are developed by commercial companies, which can provide periodical updates and new releases. Especially for OpenvSwitch, due to its popularity and wide deployment, its development is so intensive to meet performance requirements in various real scenarios. OFsoftswitch and LINC that are from academic communities can also keep updating timely and adding new features gradually. The other implementations have not been updated for a quite long time. For supported version of OpenFlow specification, v1.0 and v1.3 are two important milestones. Since v1.0 is the first published formal version of OpenFlow, and v1.3 represents

Table 2.1 OpenFlow-enabled Software Switches

Product	Developer	Code Base	Language	OF version	Update
OpenvSwitch	Nicira	Original	C	v1.0 & v1.3	08/2015
OF ref switch	Stanford	Original	C	v1.0	06/2011
OFsoftswitch	CPqD	OF ref switch	C&C++	v1.3	07/2015
Indigo	Big Switch	OpenvSwitch	C	v1.0	07/2015
LINC	FlowForwarding	Original	Erlang	v1.2 $\sim$ v1.4	08/2015
OpenFlowClick	Stanford	OF ref switch	C	v0.9	08/2009
Pantou	Stanford	OF ref switch	C	v1.0	08/2010

another big step forward by introducing multiple flow tables and other useful concepts. However, only OpenvSwitch, OFsoftswitch and LINC support up to v1.3, while the others only for v1.0. Based on our comprehensive comparison on performance, maintenance and supported OpenFlow version, OpenvSwitch and OFsoftswitch are two most representative implementations. More details of their implementations can be found in Chapter 3.

2.2 Related work

This thesis is supposed to carry out a comprehensive study on SDN-enabled software switch. It integrates several subjects that are only investigated separately before. Fig. 2.1 shows a simplified SDN framework to present current mainstream research areas in related works. The switch is composed of two compact elements, forwarding engine and OpenFlow module. Forwarding engine is in charge of data traffic forwarding, while OpenFlow module communicates with controller through control network and manipulates forwarding engine according to the instructions from controller. Similarly, in controller, the OpenFlow module is used to interact with switch. The application built on OpenFlow module implements the real control logic. The related works can be mainly divided into four categories according to the areas depicted in Fig. 2.1 (marked by numerated blocks in dash line).

(1) In switch (data plane), the performance of forwarding engine is paramount to the success of the whole framework, especially in software switch which has performance concerns compared with dedicated hardware. Besides performance evaluation of forwarding engine, there are a number of works aiming at performance improvement by applying new I/O framework. They exploit the potential advancements from both hardware and software perspectives.

(2) The interaction between OpenFlow module and forwarding engine is crucial for the efficiency of data plane. Since it matters not only the latency of the flow installation

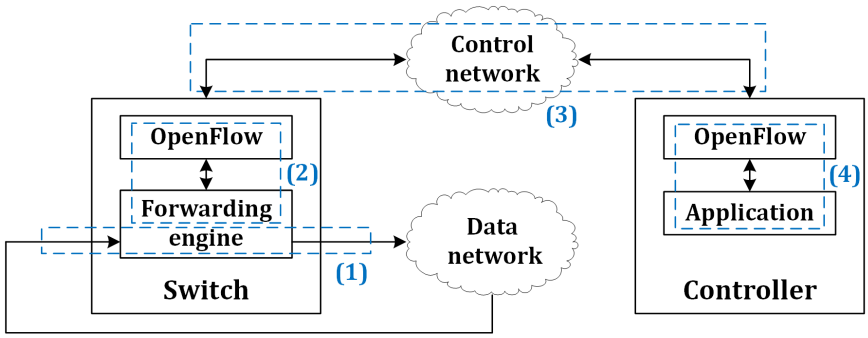


Fig. 2.1 Overview on previous related works

but also the capacity in handling new flow requests simultaneously. In general cases, OpenvSwitch performs more efficiently and more predictably than most of vendor-specific hardware switches on switch control path [83].

(3) In SDN framework, each switch needs to establish and maintain a TCP connection with its controller. There are two categories on how this connection traverses the network, namely "in-band" control and "out-of-band" control. Out-of-band uses a dedicated control network that is completely different from data network controlled by the switches, while in-band control shares the same data switching network. As a consequence, in-band control introduces complexity and additional latency in the transmission of control messages. The impact of in-band control still needs further investigation.

(4) Differently from traditional network, SDN adopts a centralized control plane called controller. The controller is a critical cornerstone in SDN paradigm, since it provides key support of all networking control logic, in accordance with the polices defined by network operators. Hence the controller directly determines the scalability and availability of the whole system. In such a context, it is necessary to understand the implementations of controllers and identify their bottlenecks, which in return provides advice on selecting suitable controller for a given scenario.

As concluded above, the deployment of SDN software switch covers several aspects from both switch and controller sides. Furthermore, due to the decoupling of data plane and control plane, SDN introduces new challenges besides the problems in traditional network. We further discuss the related works in details.

2.2.1 Switch performance

There are very few formal performance evaluations of software switch in SDN architecture in contrast with many uncompleted and partial evaluations published as blogs on Internet. OpenvSwitch now is the most widely used SDN software switch. It is a multi-layer, open source virtual switch for network virtualization. Nearly all formal publications on software switch performance evaluation target on OpenvSwitch. The developers of OpenvSwitch detail the implementation in [105] and highlight the design of caching technique and flow classification system. A two layer caching system is adopted in datapath: a `microflow cache` and a secondary layer `megaflow cache`. The microflow cache is dealing with forwarding decisions for per transport connection, while the megaflow cache is handling forwarding decisions for traffic aggregation beyond individual connection. Several micro benchmarks on caching effectiveness and efficiency are given in order to better understand its design criteria. When comparing to Linux bridge, OpenvSwitch achieves identical throughput but higher CPU usage in the simplest configuration. However, with more complex configurations, the CPU usage in Linux bridge increases over 26-fold due to per-packet overhead. While OpenvSwitch can remain constant throughput and CPU usage as before, which proves the advantage of megaflow cache.

Bianco *et al.* re-evaluate OpenvSwitch, Linux bridge and Linux IP routing in [60]. The testing scenarios cover different traffic patterns and switch configurations. The result shows that OpenvSwitch performs equivalently or better than the other two in most cases, but only slightly worse when processing 64-byte packets. OpenvSwitch also shows good fairness capability in handling multiple flows simultaneously. A more comprehensive analysis work on OpenvSwitch performance characteristics is carried out in [73]. Under various scenarios involving physical and virtual network interfaces, it shows that OpenvSwitch always turns out to be a good general purpose software switch for most scenarios. Furthermore, an investigation on CPU load of context switching and vNIC queuing leads to a few guidelines for cloud system operators: Virtual machines and NIC interrupts should be explicitly pinned to disjoint sets of CPU cores. A similar conclusion can be found in [101] that proposes a software packet processing unit to adaptively distribute $softirq$ on multiple cores.

All above evaluations measure OpenvSwitch directly and explicitly. Meanwhile, many projects targeting various research goals choose OpenvSwitch as the underlying software basis and build their prototype upon it. This again proves that OpenvSwitch is flexible and capable of implementing various functions and meeting various requirements. Moreover, the benchmark of these systems can be used as an indirect indicator of the performance of OpenvSwitch. For example, Chin Jr. *et al.* modify OpenvSwitch as a network monitor to achieve high-performance detection of DoS attack. Cloud Rack [109] has emerged to provide

a cost-efficient solution for virtual topology migration. Xing *et al.* propose SDN-based Intrusion Prevention System (IPS) based on Snort and OpenvSwitch, which can successfully alert all the threats that are sent at 15000 packets per second. There are also considerable amount of papers only investigating hardware OpenFlow switch (TCAM-based). However, due to the huge implementation differences between hardware and software switches, the conclusions in these papers can not be introduced to software switch.

As we can see, all the studies mentioned above are only based on OpenvSwitch, which lacks universality to obtain a general conclusion for SDN-enabled software switch. Even only aiming at OpenvSwitch, the performance evaluation is just the first step, the guidance on how to use OpenvSwitch in given scenarios is supposed to be more valuable. However, few suggestions for future deployment are explicitly provided, remaining for further investigations.

2.2.2 I/O framework design

Nowadays, 10Gbps interfaces are commonly used in servers. However, a standard Linux network stack is incapable of achieving line rate of its interface. This is mainly due to the overhead introduced by kernel architecture in operating system. CPU quickly becomes the bottleneck even only for normal forwarding without complex operations on packets. SDN framework requires more flexible and complicate packet processing, which further imposes great pressure on CPU. In order to achieve high-speed packet processing, several common guidelines are followed.

- **Polling**: The traditional scheme in handling incoming traffic is to generate an interrupt per packet. In scenario involving high packet arrival rate, it easily leads to livelock due to frequent interrupt requests and significantly limits the processing efficiency. Instead, polling scheme adopts a proactive manner to periodically check for the arrival of incoming packets without being interrupted, which eliminates the overhead of interrupt processing. Hence polling is more suitable for heavy traffic load. However, in an idle scenario, too frequent polling also introduces significant CPU overhead by repeatedly checking incoming packets that have not yet arrived. An alternative solution called NAPI [17] is invented by combining the advantages of both polling and interrupt schemes. NAPI works in interrupt mode by default and has the ability to switch to polling mode during the period of high traffic load for the sake of reducing total number of interrupts.

- **Zero copy**: In operating system, the standard scheme in processing a packet is shown in Fig. 2.2. The packet received by Network Interface Card (NIC) is first moved to

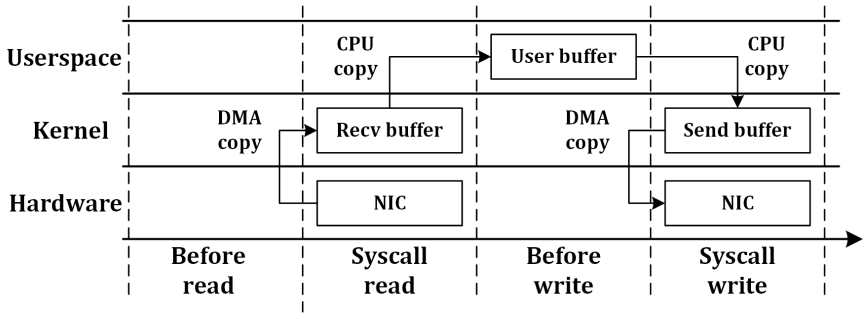


Fig. 2.2 Packet processing in operating system

kernel receive buffer through Direct Memory Access (DMA) transfer. The packet is then copied to user buffer as a system call by CPU for further user-space operations. When sending a packet, it is first copied to kernel space by CPU and then to NIC by DMA. As we can see, user-space packet forwarding requires two CPU data copies and two context switches. Zero copy aims at eliminating CPU copy as well as reducing context switches by arranging for a buffer pool to reside in a shared region of memory accessible to both user application and kernel. A true zero copy also requires a dynamic large buffer pool in case of memory overflow.

- **Bypass**: The default network stack in operating system is designed to provide a wide range of networking functionality. As a consequence, this generality leads to performance degradation which prevents high-speed packet processing. Hence a workaround is to bypass the default network stack and deliver the packet directly to the forwarding application in user-space. For basic routing and switching, a network stack is not even needed. However, in SDN scenario, a network stack is a must to establish and maintain TCP connection with the controller. The implementation of bypass usually requires the support of NICs.

- **Batching**: I/O batching technology is universally used in all high-performance solutions. By batching together multiple packets and processing them as one system call, the original RX/TX overhead and CPU load can be greatly reduced. The main drawback of batching is that per packet latency is enlarged due to the queuing time for forming a batch.

- **Pre-allocating**: The operating system always spends a significant amount of time allocating and de-allocating buffers dynamically. A pre-allocated large enough packet buffer is helpful for forwarding application to reduce this overhead without requiring further buffer operations. The huge pages [12] are usually used as pre-allocated buffers to present large regions of memory that allow user applications to access directly.

Other technologies based on specific hardware support are also proposed. For instance, multi-queue allows NIC to receive packets in multiple hardware queues and so improve load balancing on multi-core system. Since this thesis focuses on software implementation built on general purpose hardware, the dedicated hardware support is not in our scope. The techniques summarized above can be applied separately or combined together to take effect. We briefly review several representative high-performance packet processing frameworks and examine the techniques they choose.

- **mmap** [18]: mmap is a feature added to standard UNIX sockets in Linux kernel. It provides a size-configurable circular buffer mapped between kernel and user-space that can be used to either send or receive packets. mmap implements packets zero copy between two domains, which in return saves CPU resource. However, mmap uses two separate buffers for receiving and sending packets, and thus it can not realize true zero copy in a forwarding scenario. Because packets are still processed by kernel network stack and copied between two buffers.

- **PF_RING ZC** [23]: PF_RING ZC (Zero Copy) is a flexible packet processing framework that achieves 10 Gbps line rate packet processing (both RX and TX) at any packet size. It is developed by ntop [22] as a commercial library that only provides 5 minutes free use for testing purpose. It implements zero copy operations as well as programming patterns for inter-process and inter-VM (KVM) communications. PF_RING ZC also supports huge pages as well as per-NUMA node buffer.

- **Netmap** [21]: Netmap is a framework for high speed packet I/O, which is implemented as a single kernel module and available for FreeBSD, Linux and now also Windows. The network driver of Netmap is based on regular Linux driver and works transparently for operating system and traditional applications [111]. Similarly, it provides zero copy, kernel bypass, batch processing and multi-queue support to boost performance. Although it supports pre-allocated TX/RX buffers, their sizes are fixed once created. This prevents true zero copy in certain scenarios where a large amount of packets overflow the memory.

- **DPDK** [14]: The Intel Data Plane Development Kit (DPDK) is a collection of libraries and drivers for fast packet processing. It provides not only basic forwarding functions for sending and receiving packets, but also additional user-level functionality such as a longest prefix matching algorithm or a multi-core framework with enhanced NUMA-awareness. DPDK implements a run-to-completion model for packet processing over multiple logical cores as well as a pipeline model that passes packets or messages

Table 2.2 Packet forwarding framework techniques

Framework	Zero-copy	Bypass	Batching	Polling	Pre-allocating	Pcap
mmap	P	×	√	√	√	√
DPDK	√	√	√	√	√	√
Netmap	√	√	√	√	√	√
PF_RING ZC	√	√	√	√	√	√
PacketShader	×	√	√	√	√	×

among cores via the rings. The driver of DPDK is not transparent to Linux kernel, so once started, NIC is unavailable to the kernel. In order to achieve extreme performance, DPDK polls the devices in continuous loop, therefore, the CPU usage is always 100% regardless of the offered load.

- **PacketShader** [80]: PacketShader is a high-performance PC-based software router platform that accelerates packet processing with GPUs. It offloads computation and memory-intensive router applications to GPUs while optimizing the packet reception and transmission on Linux. Its I/O engine is designed for user-level application. The original packet I/O path in Linux is modified by adopting huge page buffer, batch processing, NUMA-aware data placement and multi-core scalability, and the default kernel stack is also bypassed.

As summarized in Table 2.2, mmap only implements an incomplete version of zero copy without memory sharing between sending and receiving buffer, so it is marked as P (Partial). Since mmap is only an added feature to Linux kernel, it can not bypass the default kernel stack. Due to the use of GPU, PackerShader has no support for zero copy and provides no functionality for packet capture (Pcap). DPDK, Netmap and PF_RING ZC adopt all techniques to boost performance. Since all these three frameworks are evolving in similar way, their performance gains also turn out to be close to each other based on their official benchmarks.

SDN-enabled software switch can also be accelerated by above frameworks. Table 2.3 lists the related works that boost OpenvSwitch performance by different frameworks. Due to the differences of testbed and scenario in each reference, their results can not be compared directly and fairly, but we can still draw some general conclusions. DPDK gains 6× improvement on OpenvSwitch [73] [74]. The original switch [107] based on DPDK performs slightly worse than OpenvSwitch. While Netmap achieves 3.85× improvement. Since this result was carried out in 2012, both Netmap and OpenvSwitch have been improved a lot in recent years, and we believe that a better result is expected by using the latest version. The 8× improvement in PacketShader [80] is based on a NVIDIA GTX480 graphic card. It

Table 2.3 Software switch performance improved by framework

Framework	Switch	Year	Baseline	Improved	Ratio
PacketShader [80]	OF ref switch	2010	4Gbps	32Gbps	8
Netmap [21]	OpenvSwitch	2012	0.78Mpps	3.0Mpps	3.85
DPDK [107]	Original	2013	-	10Mpps	-
DPDK [73] [74]	OpenvSwitch	2014	1.88Mpps	11.31Mpps	6.02

has 15 Streaming Multiprocessors (SMs), each of which consists of 32 Stream Processors (SPs), resulting in 480 cores in total.

By applying high-speed packet processing framework, the performance can be significantly improved by several times. For a single CPU core running over 3GHz, the maximum throughput is slightly above 10Mpps, as indicated in [73] [74] [107]. And more performance improvement can be obtained on a multi-core system. 6Wind [1] has announced in 2014 that OpenvSwitch can be accelerated to deliver 200Gbps for 1280-byte packet and 50Gbps for 64-byte packet with 10 cores. However, this improved performance still can not be comparable with dedicated hardware switch. For a 4-port 40Gbps ethernet switch which is common in data center, in order to achieve line rate on each port with 64-byte packet (full duplex mode), the backplane bandwidth should be at least 625Mpps. This is at least 6 times of 6Wind's demo, not mentioning hardware switches with more ports.

Furthermore, these performance improvements can not be fully introduced to SDN, because above results are only conducted in a simple Layer 2 forwarding scenario. While SDN represents much more than a learning switch, more complex operations on packets are required by SDN. It is also not practical to treat SDN-enabled switch as a pure data plane device, since it needs frequent communications with the controller, which in return impacts the overall performance of data plane. In brief, it is still a considerable long way for software switch to catch up the performance of dedicated hardware. At present, it is more reasonable to accept the fact that software switch has relatively lower performance compared to hardware switch. We focus on how to deploy software switch more intelligently based on their characteristics. In SDN framework, besides data plane performance, the performance of control plane and its interaction with data plane should be given equal attention.

2.2.3 Switch control path

As previously explained in Fig. 2.1, forwarding engine works as a fast path that is controlled by OpenFlow module. In hardware switch, forwarding engine is implemented by TCAM, while in OpenvSwitch, it is a kernel module associated with hashtable for fast lookup. For both TCAM and hashtable, their sizes can not be infinite to hold all matching rules. Thus,

new rules should be added and old ones should be evicted. Besides reactively install rules when receiving unmatched packets, the controller can also proactively push rules to switch in advance. All these operations are executed by OpenFlow module and take effect on forwarding module. Hence the interaction between OpenFlow module and forwarding engine (or called as "switch control path") determines not only the efficiency of flow operation but also the capacity of the whole framework [96]. As a potential bottleneck, several studies are carried out to exploit the impact of control path inside the switch.

OFLOPS [113] is an open and generic OpenFlow-enabled switch benchmark tool that mainly focuses on measuring the capabilities and bottlenecks between forwarding engine and OpenFlow module. Hardware instrumentation can be combined in OFLOPS with an extensible software framework. For instance, OFLOPS can utilize specialized NetFPGA platform in order to guarantee accuracy at sub-millisecond level [53]. In the measurements, OpenvSwitch is compared with three selected hardware switches based on PowerPC. First, through functionality testing, OpenvSwitch is proved to have fully implemented all actions defined in OpenFlow specification v1.0, while the hardware switches only support partial actions. Second, the average delay in installing/modifying a flow entry is measured. The delay in OpenvSwitch is 0.3$\sim$0.4ms which is at least one order of magnitude smaller than the result in hardware switches. Third, the average delays in querying traffic statistics in OpenvSwitch and hardware switches are around 10ms and 100ms respectively. OpenvSwitch is still 10 times faster than hardware switches. Finally, an artificial scenario is used to evaluate the impact where flow installation and frequent traffic statistic queries coexist. OpenvSwitch exhibits a marginal decrease in the median insertion delay as well as an increase in its variance, which is caused by default scheduling mechanism used in OpenvSwitch. This problem has been solved in the latest version of OpenvSwitch by introducing a more efficient scheduling framework with multi-thread support. This result implies that the user-space daemon of OpenvSwitch could be the bottleneck when dealing with massive interrupts caused by polling events. OFLOPS-Turbo [112] is one fork from original OFLOPS project and aims to be capable of testing next generation OpenFlow-enabled switches with support for 10GbE traffic. OFLOPS-Turbo is built on a series of NetFPGA-10G cards running a open source traffic generation and capturing system called Open Source Network Tester (OSNT) [52].

The diversity of SDN switch implementation including various hardware capabilities and software behaviors make it difficult to understand and accurately control the switches in SDN framework [81]. Tango [97] is proposed to explore the issues of SDN control in the presence of switch implementation diversity. OpenvSwitch and three hardware switches from different vendors are selected for measurements. The conclusion is in agreement with OFLOPS [113]. OpenvSwitch outperforms hardware switches over 10 times in installing and modifying the

flow entries. On hardware switches, the order of executing different rule operations (add, modify or delete) has a significant impact on overall performance, which can be neglected in OpenvSwitch. Moreover, OpenvSwitch provides a constant performance base on various rule-type patterns. All these results indicate that the performance of OpenvSwitch is more stable and superior to hardware switches.

Both OFLOPS and Tango are originally designed for hardware switch benchmarking. And OpenvSwitch is just used as a reference in contrast with hardware switches. It is also proved in [83] that OpenvSwitch is a poor approximation of hardware switch. However, few works aim to provide a specific and in-depth evaluation on software switch control path, although the performance of control path in software switch is higher and more predictable than hardware switch. Moreover, switch control path is only one of the components in the whole control path, the other components are control network and controller. All these three components take effect on overall performance comprehensively. Thus, the control network and the controller should be also taken into consideration for completeness.

2.2.4 Controller performance

In SDN paradigm the controller is a critical cornerstone, since it provides key support for networking control logic, in accordance with the policies defined by network operators. SDN is mainly based on a logically centralized control plane in emerging deployment scenarios (e.g., data centers, cloud, network virtualization). In SDN scheme, the first packet of a new flow is sent to the controller by the first forwarding device, and then this device waits for the instructions from the controller. This reactive scheme introduces additional latency to the forwarding of flows. Meanwhile, a large amount of control traffic may make control plane as a potential bottleneck. Because of such approach, the performance of the controller directly determines SDN scalability in such scenarios, becoming crucial to the success of SDN ecosystem.

The previous efforts addressing scalability issue can be divided into three main categories: high-performance framework, control offloading and distributed controller. Beacon [75], Floodlight [10] and Maestro [103] are representative centralized controllers that achieve high performance to handle over 5M requests per second. They all adopt well-known high-performance techniques, such as batch processing, buffering and multi-thread, to improve the total throughput. In order to alleviate the aggregated high load on control plane, DIFANE [127] offloads certain control functions from controller onto selected authoritative switches. These switches are responsible for installing rules on remaining switches. Aiming at data center, DevoFlow [72] can identify elephant flow and mice flow. Only elephant flows are handled by controller in order to reduce control overhead. Mice flows are simply

forwarded based on several pre-installed wildcard rules, requiring no more instructions from controller. Contrary to centralized controller, distributed controller can be scaled up to meet the requirement of large scale scenarios. Distributed controller relies on data distribution mechanisms and requires data consistency among multiple geographically distributed nodes. Botelho *et al.* have proved in [63] that high-performance, distributed and fault-tolerant data stores can be applied in SDN scenarios. Onix [92], OpenDayLight [28] and ONOS [26] are examples of distributed controller driven by industry. They are all hosted by a large community involving tens of vendors and collaborators, such as Google, Cisco, IBM, VMWare, HP, Juniper and Huawei. Their prototypes have been already successfully deployed in several production scenarios combined with other new technologies like OpenStack [34] and Docker [8].

The current implementations of controller are so diverse including various programming languages, different run-time technologies, and feature sets. Hence a performance evaluation is not only essential to understand these implementations and identify their bottlenecks, but also helpful to select suitable controller for a given scenario. Especially when the usability of such controllers differs from conceptual prototype to production quality, the performance evaluation must be carried out fairly and comprehensively to check what is the reality. In the past few years, number of works have carried out a partial performance evaluation. These works usually propose a new controller and use selected performance benchmarks to verify their advantages over others. Previous works on SDN controller performance benchmark is strongly related to the history of the release of controllers. Cai *et al.* [103] presented Maestro as a scalable controller in 2011. It had outperformed old versions of Nox and Beacon. In 2012, Tootoonchian *et al.* [120] first introduced multi-thread support to Nox, and consequently it defeated others at that time. Finally, the new version of Beacon [75] arrived in 2013 and brought significant improvement to outperform other controllers. Shalimov *et al.* explored the reliability and security of controller alongside performance in [116]. It proved that Ryu is more robust in handling malformed OpenFlow message, avoiding crash. As we can see, the advantages of controllers were usually temporary, because, in the meanwhile, the other controllers were as well evolving and improving. Thus, the performance evaluation of controllers had a hard time in converging on a conclusion. Since the majority of the controllers are mature enough now in their development, it is needed to re-investigate their performances in a fair manner.

To foster the evaluation of controllers, several frameworks such as Cbench [5], OFCBenchmark [84], OFCProbe [85] have been proposed. Cbench is a benchmark tool specifically designed for OpenFlow controllers. Cbench can emulate a configurable number of switches which connect to the controller by sending request messages and watching for reply messages.

Similarly, OFCBenchmark and OFCProbe also use emulated virtual switches to flood controller, but with more flexibility on switch configurations. The standard method to evaluate controllers is also discussed by IETF [59], defining a number of tests used to measure the performance characteristics of SDN controllers. In [122], Turull *et al.* further evaluates the performance of network virtualization applications build on various controllers. But the results are highly related to the implementation details of applications, which can not lead to a general conclusion. The controller is essentially a software program running on the basis of related hardware or software, however, these system-wide factors are usually overlooked in previous evaluations and should be taken into consideration in our studies. The evaluation of distributed controller is more complex, and the original benchmark tools for centralized controller are no longer suitable, which needs further investigation.

2.2.5 Analytical model

Besides above evaluation works based on practical deployment and experimentation platform, analytical modeling is also used to obtain a quick estimation on the performance, without requiring extensive simulation studies and expensive testbeds. In [86], Jarschel *et al.* model the relationship between forwarding speed and blocking probability of OpenFlow architecture as a first step. It indicates the importance of controller performance, especially in high-speed networks with 10Gbps or higher links. Since the model is based on a $M/M/1-S$ feedback queue, it can only analyze the scenario where one switch is connected to the controller. In [126], Yao *et al.* consider a multi-switch scenario and model the switch-controller communication as a batch arrival process $M^k/M/1$. In addition, it also provides a preliminary analysis on multi-controller scenario. Azodolmolky *et al.* introduce network calculus theory [56] [55] to model the behaviors of SDN controller and switch. This model can calculate the upper bound estimation of packet delay and required buffer size of switch based on the specification of various arrival processes. The packet processing delay of OpenvSwitch and buffer requirement of Beacon are demonstrated in the paper.

Beifus *et al.* carry out an in-depth investigation on packet processing latency in Linux network stack [58]. By reviewing the implementation code of PC-based packet processing in Linux, the NIC driver, NAPI mechanism and IRQ handler in operating system are modeled and simulated in ns-3. Taking OpenvSwitch as an example, the results show that this model can predict the packet latency with sub-microsecond accuracy except for some corner cases.

Although the analytical model is convenient to get a quick idea about the performance of design, most of them are still too rough to obtain an accurate and meaningful result for practical use. Even [58] is a good attempt for modeling realistic packet latency, it still can not handle several corner cases. An accurate model needs to take all variables and factors

into account, which requires lots of efforts to fully understand the whole framework in detail. Such a deep understanding is always acquired by extensive simulation studies.

2.3 What is missing?

The previous works mentioned above have covered different subjects related to software switch and SDN, including performance evaluation, framework design and analytical modeling. However, there are still many aspects that are overlooked.

2.3.1 In-band control

As explained in Fig. 2.1, there are two schemes to connect switches with the controller in SDN, namely "in-band" control and "out-of-band" control. Out-of-band control requires an independent control network that is separated from normal data network. Out-of-band control can simplify the switch implementation and keep the control traffic away from the interference of data traffic. While in-band control is easier to deploy without requiring additional control network. Both in-band and out-of-band control have their own advantages and are suitable for different scenarios.

In-band control is rarely mentioned previously and its impact is still unknown to network operations. In in-band scenario, it is challenging to handle network state update like device failover and rerouting, since any misconfiguration or wrong operation may result in the failure of the whole network. Regarding network update problem, [128] generates a correct ordering of the switches in which an update can be successfully implemented. Peregrine [121] is proposed as an Ethernet-based SDN (not OpenFlow) which adopts an in-band control network architecture. It mainly addresses two deployment issues: how to evolve the network from its initial bootstrapping mode to the explicit routing mode at run time and how to support fast failover for physical failures that break both control plane and data plane. Furthermore, [106] demonstrates that the bandwidth and the latency of control network have a negative impact on the overall performances of SDN framework. Especially for in-band control, the bandwidth and the latency are affected by the interference of data traffic, which makes the prediction of performance even more difficult.

Besides controller-switch traffic, distributed controllers require additional controller-controller traffic for information synchronization to act as a logically centralized controller. The synchronization traffic further increases the complexity of deployment. This is especially true in in-band scenario where control network and data network are the same. The interference between control traffic and data traffic can not be avoid and may impact the

overall performance. Without the dedicated out-of-band control network, the control traffic need to occupy a certain amount of bandwidth of data traffic. Overloaded data traffic may disrupt the transmission of control messages. The synchronization traffic contributes more than normal controller-switch traffic to the overall control traffic. Obadia *et al.* tackle the problem of minimizing the total control traffic generated by distributed SDN controllers in [104]. This problem is first modeled based on Mixed Integer Linear Program (MILP). A greedy algorithm is then proposed to find a near optimal placement and a spanning tree topology for controller nodes.

Although a limited number of works start to study the impact of in-band control, it is still far away from well investigated. Even the fundamental design principles of in-band control are not clearly clarified, not to mention other general conclusions for in-band control. In order to better understand in-band control, it is reasonable to start from a specific software switch like OpenvSwitch which supports both in-band and out-of-band control. The direct comparison between two schemes is helpful to observe the differences and identify the potential problems.

2.3.2 Fine-grained resource control

Compared with dedicated hardware, software switch brings more flexibility in deployment and upgrading. But software switch also leads to unstable performance in the meanwhile. For instance, the performance of software switch depends on CPU resource, and modern CPUs have features to change their running frequency dynamically according to workload. Hence the performance of software switch can not be kept consistent due to frequency variation. Even if the CPU frequency can be locked, the operating system background services that are running periodically or sporadically may also impact software switch performance. Fine-grained control on CPU resource is needed in software switch to provide predictable and stable performance.

Rather than focusing on individual switch performance, more efforts should be made in orchestration and coordination among multiple and various switches/network functions. Blaiech *et al.* propose a simple load balancing mechanism between software switches and NPU-based switches in [61]. This mechanism manages all software switches as a whole and ignore their internal conflicts. In [69], Chang *et al.* build a spring-based resource management model for end-to-end QoS of flows with given CPU and bandwidth resources. However, the basis of this model is not convincible, since it treats processing delay and available bandwidth as two independent resources. While in fact, these two resources are highly correlated in software switch. Moreover, in NFV and NaaS, multiple software switches and virtual network functions are usually required to cooperate with each other to implement a series of services

or tasks. A typical example is Service Function Chaining (SFC) which steers service-specific traffic to traverse network service functions (or middleboxes) in the given order. Since all software switches and virtual functions are sharing the same underlying physical resources, the competition and interference on resource can not be avoided. Fine-grained resource allocation is also helpful in these scenarios to minimize the interference as well as maximize the overall performance.

2.4 Summary

The combination of SDN and software switch is so promising that it has gained much attention from both academic and industry. But SDN and software switch are not panaceas and have their own limitations for practical use. It is argued in [90] that existing framework of SDN can not meet the need of real-life ISP traffic. Besides the issue of controller scalability, greater performance bottleneck may be located in the current OpenFlow switches. The performance issue of software switch is well-known and remains a hot topic in research. The combination of SDN and software switch also introduces new challenges such as the installation delay of flow rules. However, the deployment and performance problems of software switch are not yet systematically investigated in previous works. Several existing limitations are summarized as follows.

- Although there are different implementations of SDN-enabled software switches, previous performance evaluations all concentrated only on OpenvSwitch. Due to the lack of in-depth analysis and comprehensive comparison between different implementations, their conclusions can not provide the general characteristics of software switch.
- Most of previous studies investigate software switch as a standalone part and overlook the impact of other components in SDN framework such as control plane and control network, which can not lead to convincible results for practical use.
- Comparing with improving the performance of individual software switch, the orchestration and coordination among multiple software switches are even more important, especially in network virtualization environment where dozens of software switches are sharing the same underlying physical resource.
- Even if the performance of software switch can be greatly improved by new I/O frameworks, it still can not be comparable with dedicated hardware. Hence it is more reasonable to deploy software switch in suitable scenarios based on its characteristics.

In addition to the limitations mentioned above, it is challenging to integrate software switch into emerging technologies as SDN and NFV, since these technologies are immature

now and still evolving at a fast pace. The OpenFlow specification is updated and renewed every few months. More and more functions are added with the releases of new versions, e.g., multiple flow tables in v1.1, tunnel ID support in v1.3, etc. Besides OpenFlow, there are also other implementations of northbound API like POF [118] and P4 [62]. So it is hard to find out general and long term conclusions on software switch among various implementations and different versions. NFV refers to the softwarization of traditional network functions (e.g., Firewall, Load Balancer, DPI, etc.) other than basic forwarding. The work on the definition and specification of NFV is still ongoing. European Telecommunications Standards Institute (ETSI) [9] is selected to be the home of Industry Specification Group for NFV and working on the standardization of NFV. Meanwhile, Open Platform of NFV (OPNFV) [36] is launched by Linux Foundation to help ETSI bring NFV from specifications to reality using open source methodologies. By combining our research with the related work from ETSI and OPNFV, it is helpful to extend some of our conclusions on software switch to NFV.

Chapter 3

Software Switch Performance Evaluation

Dedicate hardware switches built on specific ASIC chips are capable of providing decent and reliable performance. While the performance of software switch on x86 server is not comparable to legacy hardware switch due to various factors from both hardware and software perspectives. The x86 CPU is originally designed for high-performance computation rather than high-performance packet forwarding, hence this general purpose framework introduces large overhead in processing packets compared to dedicated switching chips. Various software implementations also result in significant difference in performance. Moreover, software switch is essentially a service running above the operating system. Thus the contention on CPU resource among software switches and other services in operating system can not be avoided, which further increases the uncertainty of its performance.

Although software switch has performance concerns compared to dedicated hardware, it is gradually adopted in SDN and network virtualization based on the following reasons. First, software switches and hardware switches focus on different domains and responsibilities. Hardware switches are usually deployed in core network to guarantee high-performance traffic forwarding, while software switches reside in edge servers to connect virtual machines to external networks as well as provide customized network functions. Second, software switch provides high programmability and customization to satisfy various requirements and frequent modifications. Third, software switch essentially fits in virtualization environment. It is the basis of upper layer virtual services such as NFV and cloud computing.

In order to address the contradiction between the popularity of software switch and its limited performance for practical SDN deployment, a performance evaluation is necessary to identify the main performance factors and potential bottlenecks. As the data plane in SDN framework, software switch not only implements packet forwarding based on flow rules, but also interacts with the control plane. Hence, a comprehensive performance evaluation of software switch should also take SDN-related features into consideration.

Table 3.1 OpenvSwitch VS. OFsoftswitch

Softswitch	Implementation	Specification	Thread	Flow Table Type
OpenvSwitch	Kernel space	1.0 & 1.3	Multiple	Hashtable
OFsoftswitch	User space	1.3	Single	Lineartable

3.1 Selected OpenFlow-enabled switches

OpenFlow has become the de facto standard of SDN. Because the concept of SDN derives from OpenFlow, and OpenFlow has been widely deployed in various scenarios such as data centers and campus networks. Other southbound APIs all follow the similar design as OpenFlow and are immature for a performance evaluation aiming at providing long term results. As summarized in Section 2.1, OpenvSwitch and OpenFlow reference switch are two representative OpenFlow-enabled software switches, since other implementations are all based on them. However, the source code of OpenFlow reference switch has not been maintained and updated since 2011. OFsoftswitch is inherited and modified from OpenFlow reference switch to support OpenFlow v1.3, and its maintenance is still quite active. Hence OpenvSwitch and OFsoftswitch are selected for performance evaluation. According to the brief comparison in Table.3.1, we can see that OpenvSwitch and OFsoftswitch are quite different in various aspects, which is helpful to lead to a more general conclusion.

Fig. 3.1 shows a simplified structure of OpenvSwitch and OFsoftswitch. Each software switch has at least two fundamental components: one OpenFlow module to communicate with the controller ("ovs-vswitchd" in OpenvSwitch, "ofprotocol" in OFsoftswitch) and one datapath for packet forwarding ("kernel datapath" in OpenvSwitch, "ofdatapath" in OFsoftswitch). There are several main differences between OpenvSwitch and OFsoftswitch: 1) OpenvSwitch uses a built-in database ("ovsdb-server") to store various types of data besides OpenFlow related information, that is why OpenvSwitch can support various management interfaces and protocols (e.g., NetFlow, sFlow, IPFIX, RSPAN, CLI, etc.). 2) The datapath in OpenvSwitch is a kernel module that allows fast forwarding. Since the packets are received/sent by kernel, a kernel datapath can avoid large overhead in transition between user and kernel spaces. 3) OpenvSwitch supports multi-thread, while OFsoftswitch only runs in single thread.

Although the datapath in kernel is more efficient, the datapath in user-space has its own advantages. First, all components in user-space can be monitored, isolated, managed as processes. But the kernel module is controlled automatically and uniquely by operating system and most of execution details are invisible, which can not guarantee optimized resource allocation and security boundary in complex scenarios. Second, kernel programming

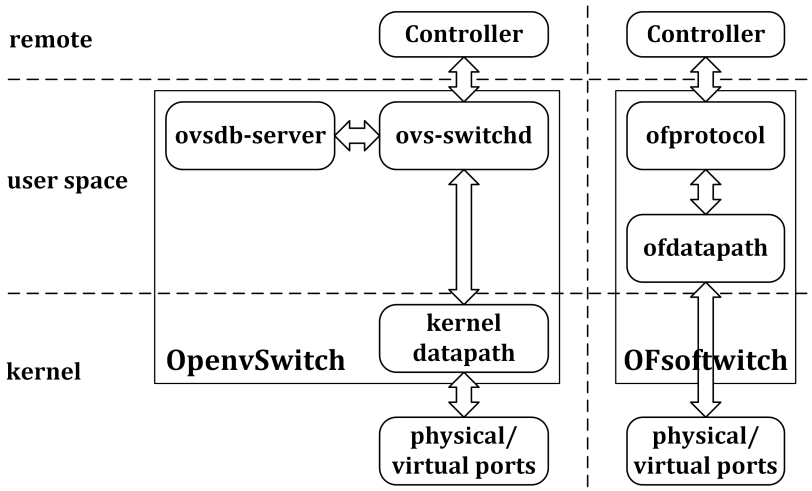


Fig. 3.1 Simplified structure of OpenvSwitch and OFsoftswitch

is much more complex and time-consuming. It is also impractical to put all OpenFlow processing into a kernel module because of the relative difficulty of developing in the kernel and updating kernel modules. Third, the user-space datapath is portable and can be easily deployed in a number of different operating systems or other platforms besides x86. Fourth, more and more efforts are shifting from kernel space to user-space, and user-space is more promising to the emerging technologies like NFV. By applying tool kits like DPDK [14], the user-space datapath can achieve equivalent or even better performance than kernel space. However, these user-space tools are not as mature as kernel technologies. Their usability, stability and reliability still need further evaluation, which is out of the scope of this thesis. Regarding the performance characteristics of kernel and user-space datapaths, OpenvSwitch and OFsoftswitch are evaluated to investigate their similarities and differences.

3.2 Evaluation environment

This section mainly focuses on the performance evaluation of software switch in virtualization environment. The testbeds in previous works ([73][60][51][68]) all follow the same design as shown in Fig. 3.2: two servers A and B are used as load generators and packet counters respectively, and the Device under Test (DuT) runs software switch on a third server. Both server A and B are connected to C with 10GbE (Gigabit Ethernet) NICs. The limitations of this testbed is manifold: 1) since all traffic load is generated outside the DuT, the maximum load is constrained by the NIC bandwidth, and 10Gbps is not sufficient to overload software switch when using MTU size packets; 2) the testing scenario is not practical with only two physical ports, and no further results with more than two ports are provided; 3) the deployment

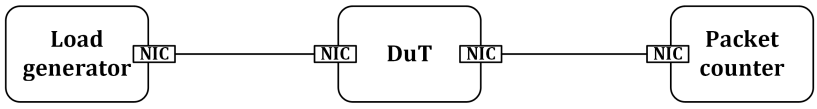


Fig. 3.2 Server setup in previous works

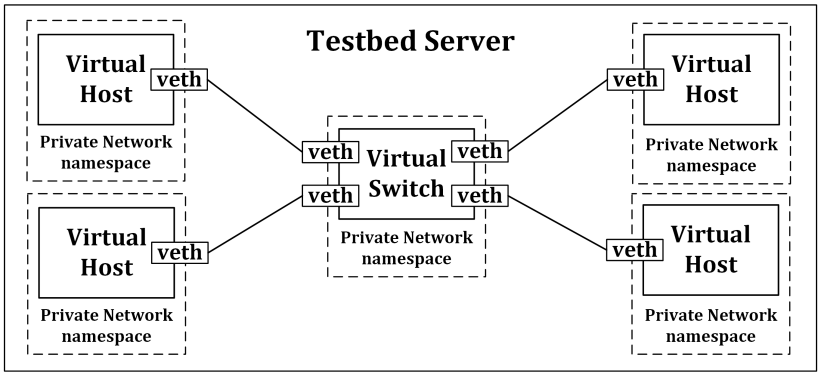


Fig. 3.3 Testbed configuration example

of software switch in network virtualization environment is usually overlooked or not well-studied. In order to address above limitations, our testbed is built on a single server and all components are virtualized. In virtualization context, virtual machines/containers coexist with software switches on the same server. They not only generate substantial CPU load for computation, but may also generate substantial traffic load for mutual communication.

3.2.1 Experimental setup

Fig. 3.3 explains how to set up the testbed in fully virtualized environment. There are 3 necessary components for software switch performance evaluation:

- **Virtual host**: Similar to the physical servers in Fig. 3.2, virtual host is the source/destination of data traffic. In performance evaluation, iperf and netperf are used in virtual host to generate testing traffic. In order to isolate multiple virtual hosts, a lightweight virtualization mechanism "Network namespace" is used instead of traditional virtual machine technology like KVM [15]. Network namespaces are containers for network states. They provide specific processes with exclusive ownership of interfaces, ports, and routing tables. Virtual or real devices can be added to each network namespace. LXC [47] and Docker [8] are all based on network namespace to provide operating-system-level virtualization. Compared with full virtualization such as Hyper-V [99] or VMware ESX [123], operating-system-level virtualization imposes little or no overhead, because the programs in containers use normal system calls and interfaces

without the need to be running in intermediate virtual machines. That is why more and more companies have deployed containers in production in the past few years.

- **Virtual switch**: In performance evaluation, OpenvSwitch (OVS) and OFsoftswitch (OFS) are used as virtual switches. In Linux there are built-in virtual ethernet devices such as Linux bridge and TAP device. From the perspective of implementation, OpenvSwitch relates to Linux bridge while OFsoftswitch is similar to TAP device. Both OpenvSwitch and Linux bridge reside in kernel and follow the same design to capture packets from interfaces. OpenvSwitch is gradually replacing Linux bridge, since it provides more flexibility in managing the network in dynamic virtualization environment without performance degradation. OFsoftswitch and TAP devices both provide packet reception and transmission for user-space programs, and all the operations on packet forwarding and modification are executed in user-space.
- **Virtual link**: Virtual network interfaces (veth) pair is used as virtual link to connect virtual devices. Veths always exist in pairs. A pair of veths are connected as a pipe, i.e., any packet received by one veth interface will come out from the other peer veth interface. Veth can be associated with virtual switches or virtual hosts, as shown in Fig. 3.3. From the perspective of virtual devices and virtual hosts, veths are treated as normal network interfaces, and related network parameters (e.g., MAC addresses, IP addresses, MTU, etc.) can be customized according to requirements.

Mininet [20] is a network emulator that allow to create a SDN network of virtual hosts, switches, controllers, and links on a single Linux server, using container virtualization. Mininet also provides a rich set of Python APIs to manage the virtual networks and devices in a unified manner. In our performance evaluation, mininet is used to establish testing topology as well as execute test scripts. By default in mininet, virtual hosts are running in containers with private network namespace, and all virtual switches are running in the same root network namespace. We modified it to also run virtual switches in container for better isolation. Ryu [40] is used as a remote controller instead of default ovs-controller. OpenvSwitch and OFsoftswitch are connected to the controller using out-of-band control for simplicity[1]. The loopback interface is used for communication between switches and the controller in order to eliminate potential bottleneck. The version of OpenFlow specification is v1.3. Only IPv4 traffic is used for performance evaluation.

The testbed server has one Intel Xeon E5-1620 CPU with 4 cores (8 logical cores when enabling Hyper-Threading) each running at 3.6GHz. The memory configuration is 4×2GB

[1] OpenvSwitch supports both in-band and out-of-band control while OFsoftswitch only supports out-of-band control. More details of in-band control and the comparison between two control schemes in OpenvSwitch are provided in following chapters.

Table 3.2 Software version

Software	Version
Ubuntu 14.04	Kernel 3.13.0
OpenvSwitch	2.4.0
OFsoftswitch	-
Controller(Ryu)	3.17
Mininet	2.2.0+
Python	2.7.6
iperf	2.0.2
netperf	2.7.0
Wireshark	1.10.6

DDR3 with 1600 MHz. The software information is listed in Table 3.2. For OFsoftswitch, no formal version number is provided. Furthermore, we modify it for performance improvement, which is to be explained next.

3.2.2 OFsoftswitch performance improvement

Although OFsoftswitch is based on user-space datapath, we find that there is still room for improvement after analyzing its source code. OFsoftswitch uses Netbee library [44] to parse and classify the packets. Instead of creating codes for customized packet processing in applications, NetBee provides powerful tools (such as the NetPDL protocol definition language or the NetVM packet processing virtual machine) to implement the same functionality more quickly and concisely. However, Netbee framework sacrifices the performance to meet the requirement of generic packet processing. In order to replace original Netbee library for improvement, we rewrite a packet classification function that exactly targets the matching fields that are pre-defined in OpenFlow. For further improvement, we not only simplify the processing pipeline by removing optional features according to OpenFlow specification, but also optimize the parameters like polling cycle.

The performance comparison between original and our modified versions is listed in Table 3.3. For Round Trip time, our modified version is 1.61× faster. When measuring the maximum available bandwidth for TCP traffic, our modified version achieves 4.11× throughput of original version. Since the performance can be improved significantly and no obvious side effect is observed, we use our modified OFsoftswitch instead of original version in latter evaluations. For more details about OFsoftswitch modification, please refer to Appendix A.

Table 3.3 OFsoftswitch performance improvement

	Original	Modified	Modified/Original
RTT (ms)	0.29	0.18	1.61
Throughput (Mbps)	195	803	4.11

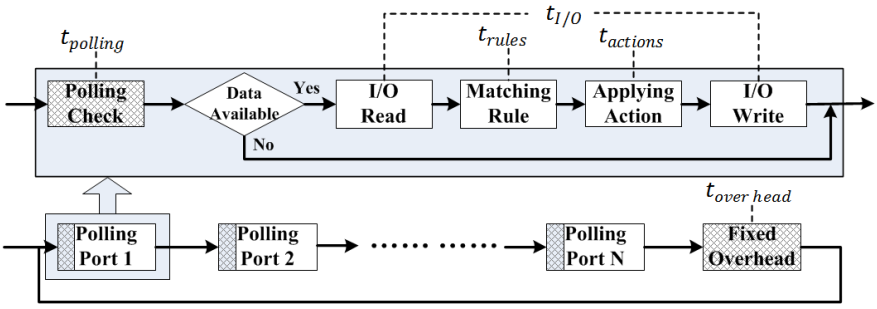
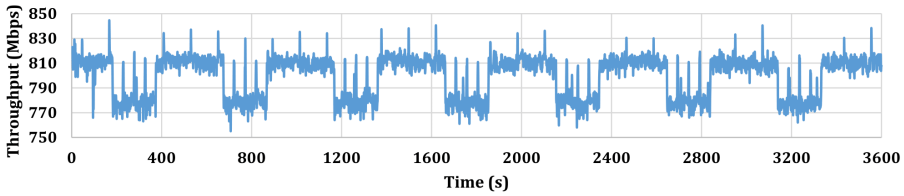


Fig. 3.4 Packet processing flow chart

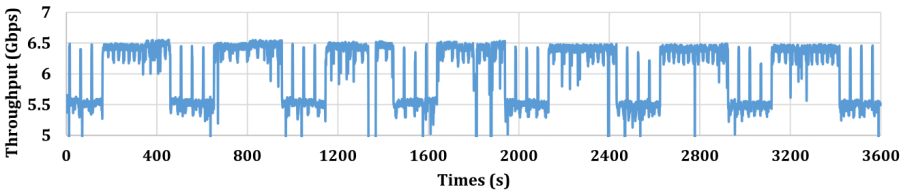
3.3 Performance factors

In this section, we investigate main performance factors and evaluate their impacts. Fig. 3.4 summarizes a standard flow chart for packets processing in software switch according to OpenFlow specification. The theoretical time for processing a packet should be: $T = t_{polling} + t_{I/O} + t_{rules} + t_{actions} + t_{overhead}$. Software switch uses polling mode to check available data on ports. $t_{polling}$ represents this time cost for polling check. If a new packet arrives at the port, I/O module first reads the packet and copies it into memory. Then this packet is matched with rules in flow table pipeline[2] in order of priority until one rule is hit. t_{rules} refers to total matching time. If no rule matches the packet, this packet is submitted to the controller for further instructions. Next, associated actions listed in matched rule are applied to the packet sequentially. The time for executing the actions is represented by $t_{actions}$. Finally, I/O module sends the packet out from memory. For simplicity, $t_{I/O}$ is the total time of packet reading and sending. After processing one port, software switch loops back to the polling step and repeats above process on next port. $t_{polling}$ is treated as an overhead, as it always exists no matter whether there are available packets. In each polling round, there exists another fixed overhead $t_{overhead}$. It is due to software design like recycling resources or registering event handler. All these overheads are marked with grid lines in the flow chart. According to the flow chart, we divide total processing time into independent parts and measure their performance impacts respectively.

[2]OpenFlow specification v1.0 only has single flow table, while v1.3 supports multiple flow table which can be used as a pipeline.



(a) OFsoftswitch



(b) OpenvSwitch

Fig. 3.5 Periodic performance of software switch

3.3.1 Periodic performance

In performance evaluation, it is important to guarantee the accuracy of the result and minimize the variation among numerous measurements. The oscillation of performance on software switch is observed in stress test. We measure the maximum bandwidth of TCP traffic on OpenvSwitch and OFsoftswitch for an hour, and the results are shown in Fig. 3.5 respectively. We can see a clear periodic cycle of performance on both switches. Because many background services have to run periodically to ensure the functionality of operating system. These services share a certain amount of CPU resource with software switches and lower their performance. The range of oscillation in OFsoftswitch is around 6% of average performance, while the oscillation is larger in OpenvSwitch at around 18%. Since the datapath of OpenvSwitch and the background services both reside in kernel, it is reasonable that OpenvSwitch faces a more fierce contention on CPU than OFsoftswitch and has a more significant performance oscillation. It is necessary to take this performance oscillation into consideration when measuring the available bandwidth. The periodic cycle of performance oscillation is 495s on our testbed[3]. Hence, in the latter performance evaluations, the testing time is always set as an integral multiple of 495s in order to obtain a more accurate result with less variation.

[3]We also measure this periodic cycle on different servers as well as other Linux operating systems, all results are around 500s.

Table 3.4 Baseline performance

Scenario	RTT (ms)	TCP (Mbps)	UDP (Mbps)
Direct veth pair	0.023	8460	12020
LinuxBridge+veth	0.037	5205	7274
OpenvSwitch+veth	0.034	7150	7780
OFsoftswitch+veth	0.095	803	1130

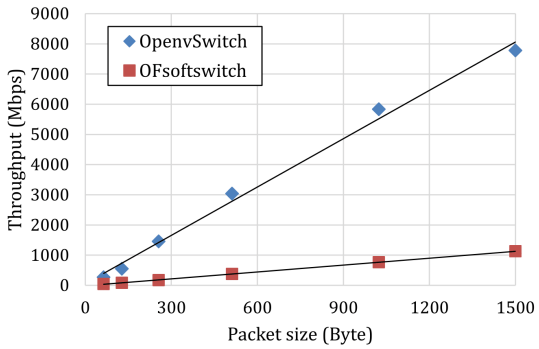
3.3.2 Baseline

We start our measurement with the most simple configuration, which is used as a baseline. Two virtual hosts (containers) are connected to the same software switch/Linux bridge. For comparison, an extremely basic scenario where two virtual hosts are connected directly by one veth pair is also evaluated. A veth pair can be treated as a simplified software hub with only two ports. All parameters of veth are set as default value[4].

Table 3.4 lists the RTT time as well as TCP and UDP throughput of a single flow. RTT is defined as round trip time, hence the total time is composed by the transmission time on link and the processing time in software switch: $t_{total} = t_{link} + t_{switch}$. In the scenario with direct veth pair, t_{link} equals to 2 times of unidirectional transmission delay on veth pair: $t_{link} = 2 \times t_{veth}$. In other scenarios, t_{total} contains 4 times of unidirectional transmission delay on veth pair and the processing time inside the software switch: $t_{link} = 4 \times t_{veth} + t_{switch}$. Since virtual hosts share the same clock on the same server, it is convenient to calculate the transmission delay on veth pair by comparing the two timestamps of the same packet that are captured on each veth. Wireshark is used to capture the packets. The result shows that t_{veth} is around 2.5$\sim$3 µs. Based on above result and analysis, we can further calculate the processing time inside each switch: t_{bridge} is around 8$\sim$10 µs; t_{ovs} is around 5$\sim$7 µs; t_{ofs} is around 60$\sim$70 µs. Between two kernel implementations, OpenvSwitch runs faster than Linux bridge. OFsoftswitch is one order of magnitude slower as a user-space implementation, because each packet has to endure kernel-to-user-space transition which results in additional data copies and large overhead.

For throughput, the throughput of UDP traffic is higher than TCP in all cases. Unlike TCP, UDP generates no ACK packet, hence UDP can achieve the same throughput as TCP with less packets. In hardware switch, each network interface works independently, and no interference occurs among them. While in software switch, there exists CPU contention among multiple virtual interfaces since they share the same physical resource. Hence the total amount of packets that can be processed on all interfaces has a upper limit. Furthermore, UDP

[4]TCP Segmentation Offload (TSO)/Generic Segmentation Offload (GSO)/UDP Fragmentation Offload (UFO) is off, and txqueuelen in veth is set as 1000.



Processing Latency in Switch

Switch	OpenvSwitch		OFsoftswitch	
Size (B)	Avg (µs)	Std	Avg (µs)	Std
64	6.2	0.53	63.4	4.9
512	6.3	0.49	64.1	6.5
1024	6.4	0.67	64.3	5.7
1500	6.4	0.65	64.5	6.8

Fig. 3.6 Impact of packet size on I/O

requires no congestion control or error correction, thus it costs less computation resources. That is why UDP can achieve higher throughput than TCP in software switch. According to the results, OpenvSwitch performs around 25% better on TCP and 7% better on UDP than Linux bridge. OFsoftswitch still provides only 803 Mbps on TCP and 1130 Mbps on UDP. Due to better performance and more flexibility, OpenvSwitch is gradually replacing Linux bridge in most scenarios.

3.3.3 I/O operation

I/O includes both packet reception and transmission, and the main factor in I/O is packet size. In this part, the impact of packet size on I/O operation is investigated. We use unidirectional single UDP flow to measure the maximum bandwidth with various packet sizes (from 64 Bytes to 1500 Bytes).

As observed in Fig. 3.6, in spite of the difference in implementation, the throughput of both OpenvSwitch and OFsoftswitch is proportional to the packet size. The degree of linear fitting is over 99.5% and 99.95% respectively. This result implies that "packets per second (pps)" value is independent of packet size. The further calculation shows that pps is around 650k in OpenvSwitch and 90k in OFsoftswitch. OFsoftswitch is a user-space implementation, so memory copy between kernel and user-space costs most of CPU time and acts as bottleneck. While in OpenvSwitch, a large amount of CPU time is spent on kernel system calls $softirq$ and $spin\_lock$. This is related to the design of veth interface which acts as potential bottleneck. More details are explained in Section 3.3.7.

We further measure the processing time of various packet sizes inside software switch under light traffic load. The table in Fig. 3.6 shows that the processing time is relatively constant with small variation. The difference between processing large and small packets

Table 3.5 Offload performance

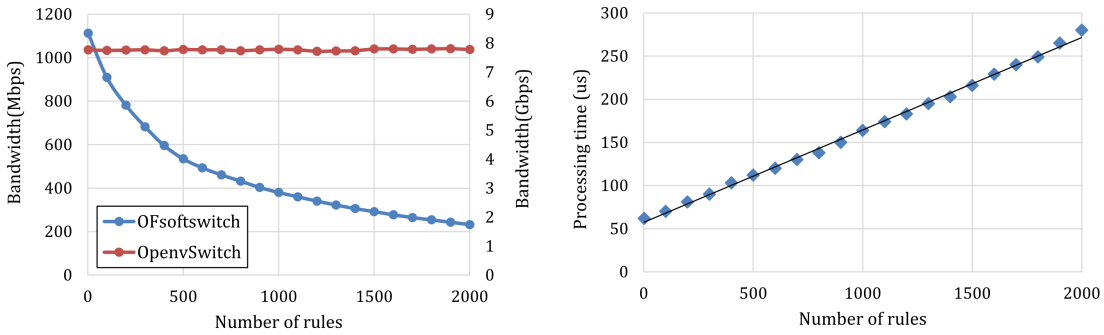
Stream	TCP (Mbps)			UDP (Mbps)		
Offload	Disable	Enable	Ratio	Disable	Enable	Ratio
OpenvSwitch	7150	25600	3.6	7780	50038	6.4
OFsoftswitch	803	9560	11.9	1130	20045	17.7

can be neglected. Our results indicate that packet size has no influence on the packet rate. The explanation is as follows: The virtual interface handles all incoming packets in the same manner regardless of packet size; furthermore, *sk_buff*s are always allocated to fit a maximum sized packet; the forwarding decision in software switch is only based on packet header which is a fix length.

In modern NIC, traditional segmentation/fragmentation techniques on TCP/UDP stream that have to be done by CPU can be offloaded to NIC for performance improvement. These operations include TCP Segmentation Offload (TSO), Generic Segmentation Offload (GSO) and UDP Fragmentation Offload (UFO). However, veth has no real implementation of offload. When enabling offload on veth, it simply sends out jumbo frames (around 64KB) without segmentation. If MTU is also set as the same large value as frame size, the throughput can be significantly improved. As shown in Table 3.5, the throughput can reach at least 10Gbps even for OFsoftswitch. To achieve 10Gbps throughput with 64KB packet, only 20k pps is needed. This value can be easily provided by OFsoftswitch (90k pps at maximum). It should be noted that the actual throughput is even higher than the value listed in the Table 3.5. Because we notice that the benchmark tool iperf/netperf has become the real bottleneck in the evaluation. In network virtualization scenarios where large MTU is available, it is reasonable to enable offload features and allow jumbo frame transmission for performance improvement. In latter measurements, we disable offload feature by default in order to obtain more accurate results on performance evaluations.

3.3.4 Rule-based forwarding

OpenFlow-enabled switch uses flow-based traffic forwarding. Each packet processed by the switch is compared against the rules in flow table based on the order of priority until one rule is matched. Thus, the number of rules has a great effect on software switch performance due to matching operation. OFsoftswitch uses a linear table for matching rules, hence t_{rules} is proportional to the number of rules it matches. We force each incoming packet to be compared with a given number of rules before matching the right one. Fig. 3.7a shows the relationship between the number of rules and maximum bandwidth of single flow in OFsoftswitch. As expected, when the number of rules increases, OFsoftswitch costs



(a) Throughput (b) Processing time in OFsoftswitch

Fig. 3.7 Impact of number of rules

more CPU time on rule matching, which lowers the total throughput. With 2000 rules, OFsoftswitch can only achieve 200 Mbps throughput (around 17k pps). We further measure the processing time t_{rules} with different numbers of rules to be matched. As shown in Fig. 3.7b, the processing time t_{rules} is proportional to the number of rules, and the degree of linear fitting is over 99.5%. According to this linear relationship, the time for matching single rule is 0.09∼0.1 µs.

OpenvSwitch uses a two-layer rule matching framework: a kernel hashtable and a user-space daemon. The kernel hashtable works as a cache to store recently matched rules for fast lookup[5]. The packet is first checked in kernel hashtable. If missed, it is submitted to user-space daemon for further lookup, and the matched rule in user-space will be installed in kernel hashtable. User-space daemon also adjusts the size and evicts rules in kernel hashtable in order to keep the most active flows in the kernel while not incurring too much overhead on keeping track of flow stats. Each flow rule in kernel hashtable has a default idle timeout as 10 seconds. In this measurement, we only focus on the impact of rules in kernel hashtable and avoid the interaction with user-space. The kernel hashtable should guarantee that matching time is a constant value regardless of the number of rules as long as the packet hits the hashtable. As proved in Fig. 3.7a, OpenvSwitch provides stable maximum bandwidth, and the number of rules has no impact on total throughput.

3.3.5 Impact of rule actions

Each flow rule is associated with zero or more actions that indicate how to handle the matching packets. If no action is present, the packet will be dropped. The switch must

[5]The kernel module in OpenvSwitch is designed as a microflow cache in which a single cache entry exact matches with all the packet header fields supported in OpenFlow specification.

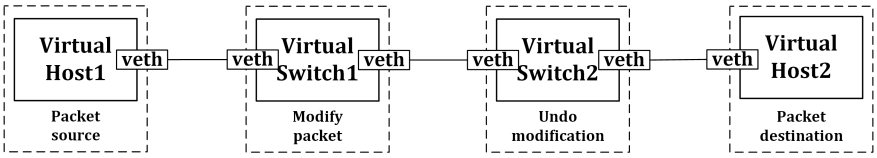


Fig. 3.8 Topology for action measurement

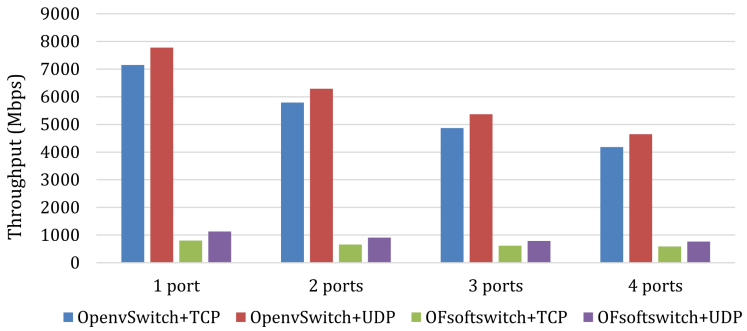


Fig. 3.9 Throughput with multiple output ports

execute all action in the lists according to the specified order. However, packet output ordering is not guaranteed within a port.

The actions that can be applied are various from basic operations like rewriting packet headers to complex ones like packet encapsulation (VLAN or MPLS). In software switch, the time cost of action depends on its complexity. In order to measure the impact of various actions, we design a testing scenario as shown in Fig. 3.8. The source and destination hosts are connected by two software switches linearly. Virtual switch1 first modifies the packet, and then switch2 undo the modification (For VLAN/MPLS, switch1 inserts the shim layer and then switch2 removes it). The baseline is set as basic forwarding without any modification on packets. The comparison shows that the throughput has no difference with various actions. The further measurement shows that the time cost of various actions differs from 0.01 µs to 0.02 µs. This overhead is so tiny to be ignored when compared with total processing time.

The most basic action is packet forwarding. In a multicast or monitoring system, one packet should be copied and sent out on more than one port. This additional copy and transmission degrades the total throughput. Fig. 3.9 shows the relationship between throughput (TCP and UDP) and the number of output ports. The throughput of OpenvSwitch decreases around 40% with 4 output ports compared with one port. The impact on OFsoftswitch is smaller, the result with 4 ports can still achieve 75% of the throughput with one port. Moreover, the result with 3 ports and 4 ports are close to each other. Since a significant performance degradation can be observed, the deployment of software switch in such a multicast or monitoring system should be designed more carefully.

3.3.6 Polling & Overhead

Different from hardware implementation where multiple ports can run in parallel, software switch uses polling mode to check available packets on ports. As shown in Fig. 3.4, besides the fixed overhead in a polling round, checking available packets on each port introduces additional overhead. For example in OFsoftswitch, each port is polled actively in a Round-Robin manner from user-space, and all ports share the same CPU core. For each polling round, at most one packet can be processed on each port. Therefore, the maximum "packets per polling round ($pppr$)" is equal to the number of ports. When the number of ports is fixed as n, we define an indicator called "Effective packet processing ratio (r)" which represents the ratio between actual time for packet processing and the total time including polling and overhead. The theoretical value of r is defined as:

$$r = \frac{pppr \times t_{processing}}{n \times t_{polling} + pppr \times t_{processing} + t_{overhead}}$$

where $t_{processing} = t_{I/O} + t_{rules} + t_{actions}$ and $pppr \leq n$. Clearly, the total packets can be processed "pps" is proportional to r. OFsoftswitch is a single thread user-space implementation, therefore the maximum CPU resource it can use is fixed as one core. In order to maximize total pps, OFsoftswitch should try to send out packets as many as possible in one round to lower the share of overhead. According to the design of OFsoftswitch the number of packets it can process depends on the number of the ports and the traffic pattern on each port.

In order to evaluate such aspect, we design a crafted testing scenario: OFsoftswitch has 2,4,6,8 ports in each test, the packets arrived on each port are carefully controlled in order to guarantee that only given number of packets can be processed by OFsoftswitch in each polling round. The results in Fig. 3.10 show that when the number of ports is fixed, the larger the $pppr$, the larger total pps achieved. When $pppr$ is fixed, more ports results in performance degradation, as more CPU time is wasted on polling check. When $pppr$ is equal to n, along with the increase of n, r tends to reach its upper limit as $t_{processing}/(t_{processing} + t_{polling})$. As a consequence, when n is over 6, the total throughput tends to converge to a constant value around 110k.

Differently from OFsoftswitch that adopts a simple user-space polling method, OpenvSwitch uses NAPI (default packet reception API in Linux kernel) to handle arrived packets more intelligently. Instead of processing only one packet on each port in each round, NAPI allows to poll a certain number of packets on the same port simultaneously in each round when traffic load is heavy. This effectively lower the share of $t_{overhead}$ in each round. NAPI also adopts interrupt-based processing when traffic load is light, which eliminates useless polling check on empty ports. Furthermore, the kernel datapath of OpenvSwitch supports

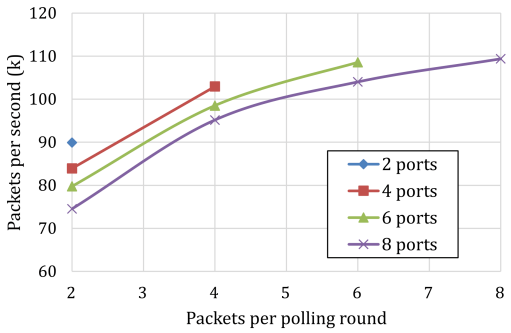


Fig. 3.10 Impact of traffic pattern in OF-softswitch

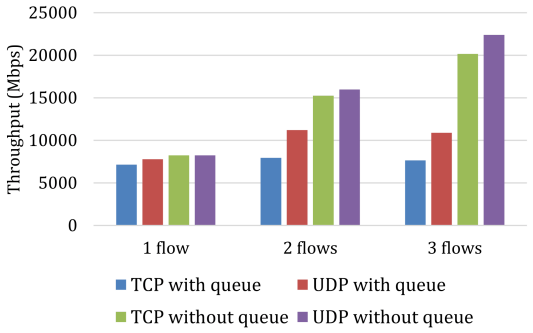


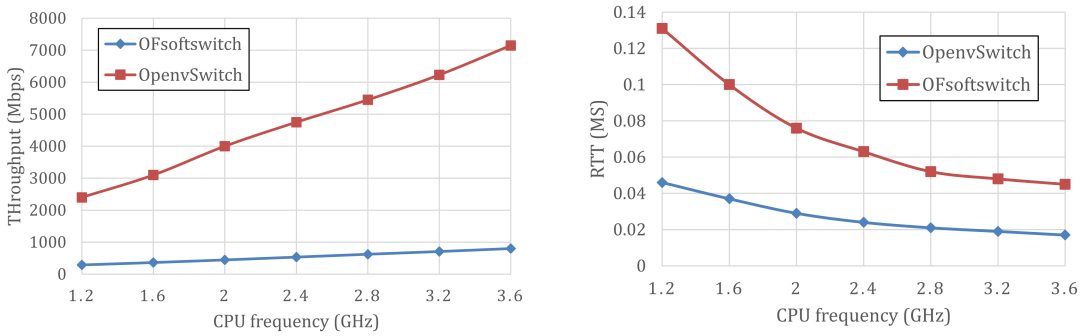
Fig. 3.11 Impact of veth queue in Open-vSwitch

multi-core processing. Hence, the impact of traffic pattern on the performance of Open-vSwitch theoretically can be significantly alleviated as long as the kernel resource is still available. For evaluation, we start a single flow on each port of OpenvSwitch simultaneously. The result shows that the throughput of each flow is as same as the value when it is running alone, which proves our above analysis.

3.3.7 Veth interface

By default, veth provides a "qdisc" queue [46] in transmission side, which is a single queue with default length as 1000. This queue is used for traffic shaping as well as providing QoS. However due to its single queue design, it can not scale well for multi-thread programs or multi-core system. As shown in Fig. 3.11, with queue design, when multiple flows are running on one port at the same time, the total throughput has no obvious improvement. Most of CPU time is spent on kernel system call *spin_lock*, which means that there is intensive lock contention on the queue among multiple flows. This "qdisc" queue becomes the bottleneck in this scenario.

In order to boost the performance, this queue is removed and the measurement is repeated. The total throughput is roughly proportional to the number of flows, which has clear improvement compared to previous result with queue. For OFsoftswitch case, the real bottleneck is still on user-space packet processing, thus veth queue has no impact on its performance. Although the performance can be improved by removing veth queue, this also sacrifices the features in fine-grained control on traffic. In latter evaluations, we still keep the use of queue due to its functionality on traffic control.



(a) Throughput (b) Round Trip Time

Fig. 3.12 Impact of number of rules

3.3.8 Impact of CPU running frequency

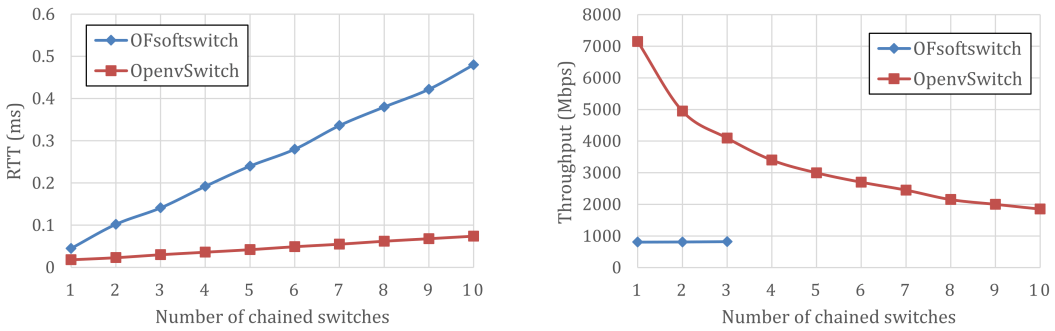
Software switching is essentially a CPU-intensive task, and the performance of software switch greatly depends on the power of CPU. Thus we measure the relationship between CPU running frequency and software switch performance. As shown in Fig. 3.12, the throughput is proportional to the frequency of CPU. The higher the frequency, the lower the RTT value. The RTT value is roughly proportional to the reciprocal value of CPU frequency.

In above measurements, we manually set the running frequency of CPU. Intel CPUs have features to change its running frequency according to workload. This behavior brings benefits in energy saving on the one hand, but results in unstable and unpredictable performance on the other hand. For instance, light traffic load motivates CPU to lower its running frequency at only 1.2GHz, while the frequency is set at 3.6GHz with heavy traffic load. There is a tradeoff between energy saving and high performance. Thus the deployment of software switch should find a balance between QoS and energy aspects according to real needs.

For OFsoftswitch, it is convenient to monitor its CPU usage and allocate CPU resource by its user-space process. While it is impossible to control OpenvSwitch in the same way. The kernel datapath in OpenvSwitch can not be scheduled or monitored directly from user-space. It simply runs in-line in *softirq* on the CPU that the packet is originated on. Since it resides in kernel, it always has a higher priority in execution than normal processes. Hence, OpenvSwitch is not suitable for the scenarios where explicit CPU control is required.

3.3.9 Chaining software switches

In Openstack [34] scenario, it is common to chain multiple virtual switches to provide network virtualization in different levels and granularities. More generally in NFV scenario, multiple virtual network functions are assembled in a chain to provide a series of complex



(a) Round Trip Time (b) Throughput

Fig. 3.13 Chained switches

services. In order to connect and manage these network functions efficiently, multiple switches are usually chained together, and network functions are connected to chained switches. For measuring the impact of the number of chained switches, multiple software switches are chained in a line between two virtual hosts. The throughput and latency results with different number of chained switches are shown in Fig. 3.13. As expected in Fig. 3.13a, RTT value is proportional to the number of switches in both OpenvSwitch and OFsoftswitch cases. In OpenvSwitch, RTT value still can be kept below 0.1ms even with 10 switches.

OFsoftswitch and OpenvSwitch show different behaviors on throughput result in Fig. 3.13b. The throughput of OFsoftswitch is independent of the number of chained switches. The total throughput of a chain depends on the minimum throughput (potential bottleneck) among all switches on the chain. In the measurement, each OFsoftswitch is running on an isolated CPU core, so each can achieve its maximum throughput due to its single thread design. Hence the total throughput with chained switches is as the same as with only one switch. Since we have only 4 cores, the maximum number of OFsoftswitches is limited to 3 in order to guarantee each switch can occupy a single core. For OpenvSwitch, a significant performance drop is observed with the increase of chained switches. The throughput with 4 switches only achieves less than 50% performance of single switch, and the throughput with 10 switches achieves only 25%. This is due to the internal contention inside Linux kernel. More switches raise more interrupts for receiving packets, so most CPU resource is wasted on handling queued software interrupts instead of forwarding packets. Furthermore, the performance variation is also greatly enlarged with more switches. As discussed in Section 3.3.1, the performance oscillation of single OpenvSwitch is around 18%, while it reaches up to 40% with 10 switches. This implies that OpenvSwitch is not capable to provide a stable performance with multiple datapaths inside the same kernel.

3.3.10 Tiered latency in SDN

SDN adopts a centralized control plane to install flow rules on each connected switch. When the first packet of a new flow arrives at the switch, it is forwarded to the controller for forwarding decision. This reactive behavior introduces additional latency overhead in processing the first packet of flows, compared with traditional proactive routing scheme. After the new rule is installed on the switch, the latter packets from the same flow is forwarded directly inside switch without further interaction with the controller. We define the path go through the controller as "Control path" which is marked in Fig. 3.14. As explained in Section 3.3.4, OpenvSwitch has a two-layer rule matching framework, with a kernel module as a cache and a user space daemon (ovs-vswitchd) as the actual flow table. The kernel module only stores the recently matched rules to forward the packets in kernel without going through user-space. This is treated as "Fast path". While OFsoftswitch has no "Fast path" without kernel implementation. Since the kernel cache has a default idle timeout as 10s, if one flow is inactive over 10s, the related forwarding rule will be evicted from the kernel module even if it is still in the flow table in user-space. The next arrived packet will miss in the kernel, and it requires user-space daemon to reinstall the rule in the kernel module. This user-space path is named as "Slow path" as shown in Fig. 3.14. So in OpenvSwitch, there are totally 3 different paths (Fast, Slow and Control) to forward a packet. While OFsoftswitch has only slow path and control path.

Table 3.6 lists the RTT results for a packet passing through different paths. In the measurement, the controller is running on the same server as software switches, so the transmission delay between the controller and the switch is quite tiny. The RTT value of three paths in OpenvSwitch are clearly in three different orders of magnitude. Kernel path is fastest, which is around 10 times faster than user-space path, and 100 times faster than control path. OpenvSwitch performs worse than OFsoftswitch in slow path. This is due to the additional matching in kernel module and the interaction between kernel module and user-space daemon, which introduces more overhead than the processing in OFsoftswitch. The results of control path prove that OpenvSwitch is more efficient in handling OpenFlow messages than OFsoftswitch. Short-lived flows or inconsecutive flows may frequently go through slow path or control path instead of fast path, so it is necessary to take different paths into account when designing latency-sensitive SDN network.

3.4 In-band control

In OpenFlow network, there are two different types of traffic: control traffic and data traffic. Control traffic contains all the control messages defined in the OpenFlow specification, while

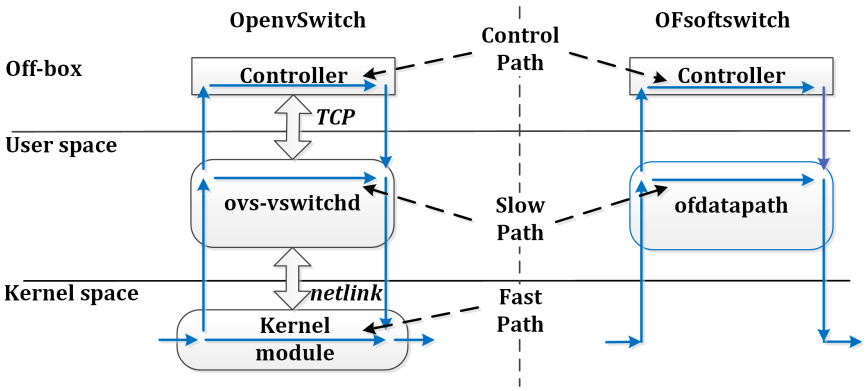


Fig. 3.14 Tiered latency in software switch

Table 3.6 Tiered Processing Latency – RTT (ms)

Path	Fast	Slow	Control
OpenvSwitch	0.019	0.23	2.3
OFsoftswitch	-	0.095	2.7

data traffic includes all the other traffic. The network that the control traffic traverses over is treated as control network, similarly, there is data network for data traffic.

Each OpenFlow switch needs to establish and maintain a TCP connection to its controller. There are two basic categories on how this connection traverses the network: it is either using a dedicated network which is completely different from the one controlled by the switches, or it is overlapping the network that the switches control. The former case is treated as "Out-of-band control", while the latter case is "In-band control".

Out-of-band control has the following benefits:

- **Simplicity**: Out-of-band control simplifies the switch implementation.
- **Reliability**: Switch traffic volume can not interfere with control traffic.
- **Integrity**: Devices not on the control network can not impersonate other devices.
- **Confidentiality**: Devices not on the control network can not snoop on control traffic.

In-band control, on the other hand, has the following advantages:

- **No dedicated port**: There is no need to dedicate a physical switch port to control, which is important on specific switches that have few ports (e.g., wireless routers).
- **No dedicated network**: There is no need to build and maintain a separate control network, which is helpful to reduce proliferation of switches and wiring.

Fig. 3.15 shows the differences between out-of-band and in-band control frameworks. We simplify the internal structure of OpenFlow switch as two parts: a "FlowTable" that forwards incoming packets based on the flow rules installed by the controller, and a "OpenFlow (OF)" core module that communicates to the controller as well as manipulates the FlowTable according to the instructions from the controller. In out-of-band control, there exists one dedicated control port and one dedicated control network. So OF module can communicate to the controller directly without involving FlowTable. Control traffic and data traffic are separated by ports (control port and data port) and traversing on different networks (control network and data network). While in in-band scenario, without the dedicated control network, control traffic and data traffic are mixing together on the same network and arriving at FlowTable simultaneously.

In order to make a fresh OpenFlow-enabled switch start to work, it has to connect to the controller at first. Thus, the existence of control network is a prerequisite for switches. In in-band scenario, the control network is the same one as the data network. This will result in a contradiction when booting up the network: the switches require a control network to contact the controller, but the control network can not be established until the switches can contact the controller. The contradiction implies that switches must recognize control traffic without involving the controller.

3.4.1 In-band solution

The booting up problem of in-band control requires that the control traffic can be forwarded correctly by the switches before establishing the TCP connections between switches and the controller. Fig. 3.16 shows a general framework to solve this problem. When mixed control traffic and data traffic are arriving at the switch, a filter is first used to recognize all the control traffic and then deliver the control traffic to a forwarding module (Data traffic are

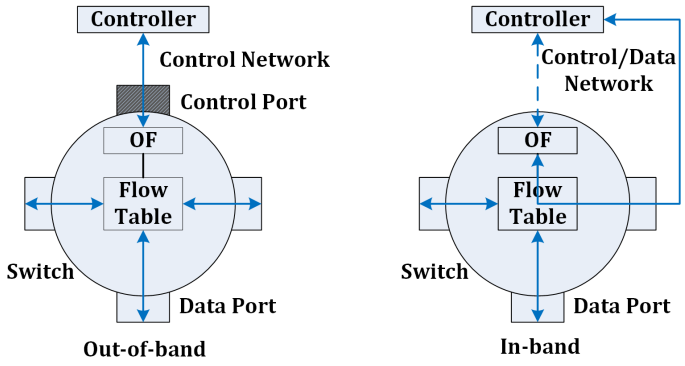


Fig. 3.15 Out-of-band control and in-band control

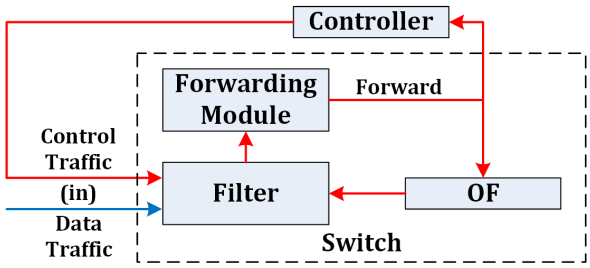


Fig. 3.16 Booting up framework for in-band control

dropped by filter until the switch can connect to the controller). The forwarding module is responsible to establish the connection between OF module of each switch and the controller. It must make the correct forwarding decision independently, which requires itself to be a stand-alone system differently from default OpenFlow framework.

This framework also raises a new problem. After the connection is established, the OF module should continue to forward the control traffic by itself or not? It is reasonable to assume that the controller should take over the control traffic from forwarding module, on the basis that the controller should be in full charge of the switch. However, this is not practical. Since any misconfiguration on flows or network events (e.g., link failure) can result in the collapse of the control network. Hence, a more reasonable design should prevent the controller interfering the forwarding of control traffic in order to guarantee the availability of the control network. The control traffic should be invisible to the OF module.

3.4.2 OpenvSwitch in-band implementation

OpenvSwitch supports both out-of-band and in-band control. The design principles behind in-band control in OpenvSwitch are explained in [105]:

1. In-band control must be implemented regardless of whether the switch is connected to the controller or not.
2. The switch must recognize all control traffic.
3. In-band control must override flows set up by the controller.

These three principles can exactly match the explanation in Section 3.4.1. The first two principles are claimed to realize a control network without involving the controller, while the last principle emphasizes that the control traffic should own a higher priority than normal data traffic and be out of the control scope of OpenFlow framework.

Fig. 3.17 demonstrates the internal logic of in-band control in OpenvSwitch, which is also a practical implementation of the framework in Fig. 3.16. It is helpful to understand the

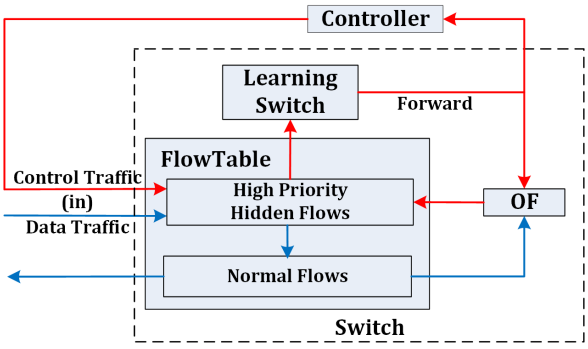


Fig. 3.17 In-band control in OpenvSwitch

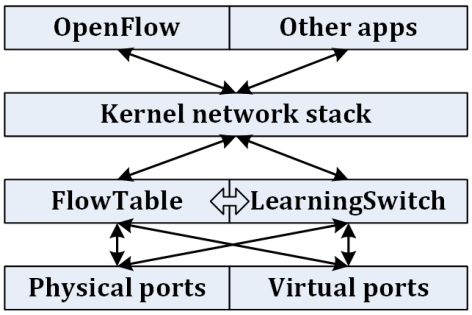


Fig. 3.18 Network stack for in-band control

implementation details by comparison between two figures. In order to recognize the control traffic, the FlowTable can be used as a filter. In OpenvSwitch, in-band control is implemented as "hidden" flows (they are invisible to OpenFlow) and at higher priorities than flows set up by the controller. Since the controller can not interfere with control traffic, it avoids the potential risk of breaking the connectivity of control network by misconfiguration. A learning switch is used in the OpenvSwitch to forward control traffic (corresponding to forwarding module in Fig. 3.16). It can work independently to guarantee the connectivity of control network. Once the control network starts to work, the data traffic can be forwarded as normal flows according to the instructions from the controller. In OpenvSwitch implementation, there are actually two network stacks, learning switch and OpenFlow, that are handling control traffic and data traffic separately. Fig. 3.18 displays the layer structure of these two network stacks. The layer of FlowTable and LearningSwitch is inserted between port layer and default kernel network stack. This is implemented by hooking functions that is similar to Linux bridge. Although FlowTable and LearningSwitch are working independently, they are sharing the same underlying ports and they also have interactions with each other. They use a specific port called "internal port" to connect to original kernel network stack. Various applications including OpenFlow module are still running above kernel stack.

3.4.3 Learning switch with selected flows

To establish the control network, the correct flows should be selected by FlowTable and then forwarded to the learning switch. These flows are divided into two groups: the flows that achieve connectivity of the control network and the flows that recognize the control messages. In OpenFlow, all control messages are encapsulated in TCP. Thus, it is convenient to recognize the control traffic by two simple flow rules:

(a) TCP traffic to the controller's IP and port.

(b) TCP traffic from the controller's IP and port.

The learning switch works in MAC layer. In order to achieve connectivity, only ARP related flows are needed, which also can be defined by two simple flow rules:

(c) ARP replies to the local port's MAC address.

(d) ARP requests from the local port's MAC address.

The packet that matches any of above rules will be forwarded by the learning switch. The goal of these rules is to be as narrow as possible to allow a switch to join a network and communicate with the controller. Since these rules have higher priority than the controller's rules, if they are too broad, they may prevent the controller from implementing its policy.

Fig. 3.19 is a simple example that demonstrates the procedure to establish the control network step by step. We assume that the controller (c) is directly connected to the switch s1. In Step one, (1) s1 needs to establish TCP connection to the controller, and it only knows the IP address of the controller and the destination port (6633 by default). So the kernel network stack first generates an ARP request. This request matches flow rule (d), so it is forwarded to learning switch. The learning switch then floods this ARP request. (2) When the controller side receives the ARP request, its network stack sends out ARP reply back to s1. ARP reply matches flow rule (c). The learning switch learns the MAC address of the controller. After knowing the MAC address, the TCP connection can be established. (3) Once connected, both the switch and the controller must immediately send a $Hello$ message with the version field set to the highest version supported by the sender. OF module sends out a $Hello$ message to the controller. This message matches flow rule (a). According to the MAC address table, the learning switch sends out the message on the right port to the controller. (4) The controller also needs to send out a $Hello$ message to s1. This message matches flow rule (b). The learning switch then submits it to OF module. If the negotiation of OpenFlow version succeeds, the control network between s1 and the controller is established.

In Step 2, we further assume that another switch s2 is connected to s1. s2 also needs to communicate with the controller. (1) Similarly, the ARP request is first flooded to s1. (2) The ARP request can not match any flow rule in s1. Since s1 is already connected with the controller, this ARP request is encapsulated in a *Packet_In* message and forwarded to the

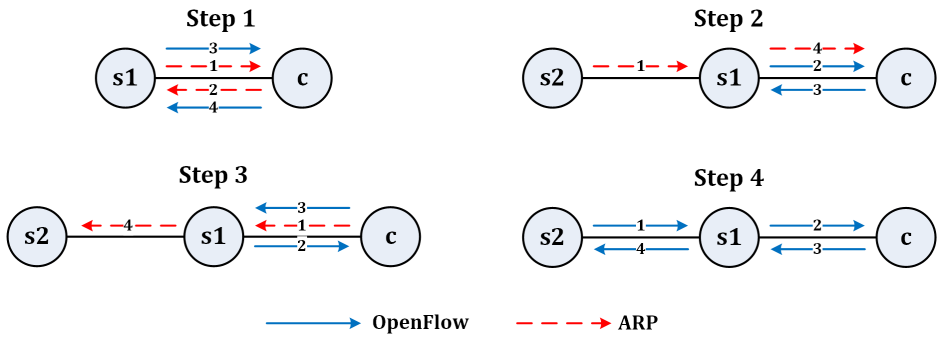


Fig. 3.19 The procedure to establish in-band control connection

controller. (3) When receiving the *Packet_In* message, the controller sends out a *Packet_Out* message to s1 with forwarding instruction. (4) s1 forwards the ARP request to the controller based on the instruction in the *Packet_Out* message. Step 3 is quite similar to Step 2. In Step 3, s1 forwards the ARP reply from the controller back to s2.

After establishing the TCP connection, the controller and s2 exchange *Hello* message in Step 4. (1) s2 sends out a *Hello* message. (2) The *Hello* message first arrives at s1 and matches flow rule (a). So learning switch further receives it and learns s2 MAC address. Then this message is sent out to the controller. (3) The *Hello* message from the controller to s2 arrives at s1. (4) The *Hello* message matches flow rule (b) in s1 and is forwarded to s2 according to the MAC address table. Now the control network between s2 and the controller is also established.

Any other switch that connects to s1 or s2 can follow the similar procedure to connect to the controller. Hence, we have proved the in-band control can be implemented by only 4 flow rules. It should be noted that in this example we assume that both switches and controller are in the same subnet. However, additional flow rules are needed in more complex scenarios (e.g., through gateway, between VMs).

3.4.4 In-band control latency

The reactive behavior to install flow rules in SDN has introduced additional latency in processing new flows compared with traditional routing scheme, since the first packet of a new flow have to be forwarded to the controller through control network. Without a dedicated control network in in-band control scenario, this latency can be further enlarged due to the topology and the link state of data network.

Fig. 3.20 displays a basic example of in-band control scenario: n switches are linearly connected; there is one host on each end of switch chain; the controller connects directly to

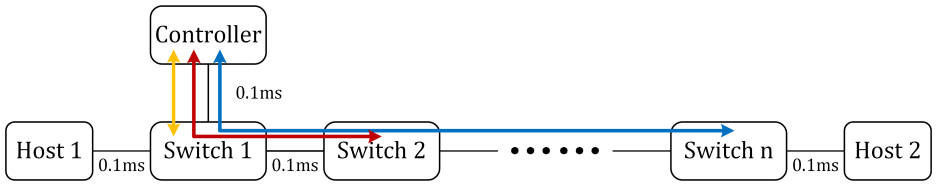


Fig. 3.20 In-band control example

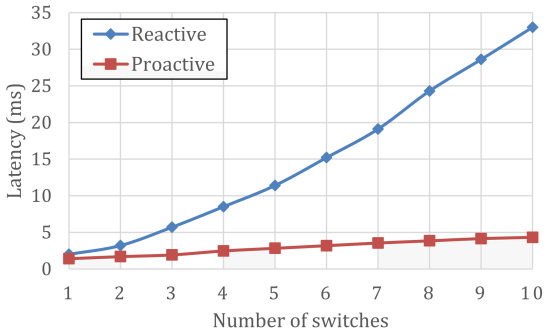


Fig. 3.21 Reactive vs. Proactive

switch 1. In order for switch n to communicate with the controller, the control messages have to pass through all the other switches (with a serial number smaller than n) to arrive at the controller. This behavior results in large latency due to long control path. When host 1 generates a new flow to be received by host 2, the first packet of this flow will arrive at each switch sequentially. Each switch has to communicate with the controller through a long control path. The total control latency is the sum of the latency on each switch. We set link delay as 0.1ms to simulate the real scenario. Fig. 3.21 shows the processing latency of in-band control with different number of switches based on the topology in Fig. 3.20. The latency increases quickly up to 33ms with 10 switches, which is already not suitable for latency-sensitive environment. In order to address this problem, a proactive flow installation scheme is proposed. Instead of waiting for requests from each switch, the controller should install rules on all related switches at the same time once it receives the request from the first switch. As shown in Fig. 3.21, a proactive scheme can significantly reduce the latency into an acceptable range. Because this scheme not only eliminates the accumulative effect of latency on each switch, but also installs rules more efficiently by proactively pushing rules.

3.5 Summary

In this chapter, we focus on the performance analysis of two OpenFlow-enabled software switches, namely “OpenvSwitch” and “OFsoftswitch”, in virtualization environment. Our

measurements are carried out in a fully virtualized environment, which is different from previous studies that usually aim at physical platform and use software switches in the same way as dedicated hardware. The roles of software switch are diverse when combined with new networking technologies such as SDN and NFV. It is no longer a dumb device with limited intelligence or functionality. Instead, it is extended and generalized to support various network services. For example in NFV, software switch is the basis to assemble multiple virtual network functions in the right order to deliver expected services in more flexible manner. In SDN, the OpenFlow-enabled software switch can also act as NAT or stateless Firewall by simple configuration. Furthermore, the immaturity of SDN and NFV makes the deployment of software switch even more challenging. Hence our evaluation is based on this context to investigate the performance characteristic of software switches deployed in network virtualization environment. This is also helpful to identify the real bottleneck and lead to more rational deployment decisions.

In the course of analysis, we first analyze the OpenFlow-enabled packet processing and divide it into several main factors, and then evaluate these factors on two software switches separately. Although the implementation of OpenvSwitch and OFsoftswitch are quite different, they still share some common conclusions. For instance, the pps throughput of single flow is independent of packet size; the impact of various actions on performance can be ignored. On the contrary, they also show quite different behaviors in other aspects. Although OpenvSwitch has a kernel datapath that brings high performance for packet forwarding, it results in larger performance oscillation due to intensive contention with other kernel modules and operating system background services. OFsoftswitch is a single-thread user-space implementation which can only achieve total throughput around 1Gbps. It allows accurate and fine-grained control on CPU resource as well as provides stable and predictable performance. Since more and more efforts are working on performance improvement for user-space datapath recently, the user-space datapath implementation is more promising than the kernel one.

Besides the performance factors inside software switch, we further investigate the external factors such as veth pair and CPU frequency. Since we mainly focus on a container environment, veth pair is the standard way to connect different network namespaces. Although veth provides rich features, it has performance concerns with default qdisc queue design. It can not utilize multiple CPU cores efficiently and can not scale well with multiple concurrent flows. This default queue can be removed from veth for performance improvement, but meanwhile all related features such as traffic shaping or QoS are also disabled. There is a trade-off between performance and security features. Similarly, modern CPU has an adaptive frequency setting to fit various workloads based on the consideration of energy saving. It

is necessary to balance energy cost and performance according to the real needs. The more complex scenario where multiple software switches are chained together is also evaluated. This indicates the limitation of OpenvSwitch to run multiple kernel datapaths. The reactive scheme in SDN framework results in tiered latencies through different paths (control, slow, fast). It is necessary take these tiered latencies into consideration when designing a latency-sensitive network.

In-band control is convenient for deployment without requiring a dedicated control network. However, it is rarely mentioned in previous studies. We systematically discuss the design principles behind in-band control. Combined with the in-band control implementation in OpenvSwitch, we further explain why and how in-band control works in real scenarios. In-band control introduces larger latency in handling new flows due to long control path. A proactive rule installation scheme is proposed to address latency problem. The simulation result shows that the latency can be reduced to an acceptable range by using proactive scheme.

In summary, our studies investigate and summarize the deployment issues of OpenFlow-enabled software switch from the perspective of performance. Various performance factors are evaluated in a fully virtualized environment. Our results can be used as a guideline to design a practical SDN-enabled network based on software switches. They are also helpful to achieve fine-grained resource control on software switch, which is explained in Chapter 5.

Chapter 4

Controller Performance Evaluation

The controller is a critical component in the SDN paradigm, since it provides key support for networking control logic, in accordance with the policies defined by network operators. SDN is mainly based on a relatively centralized control plane in emerging deployment scenarios (e.g., datacenter and cloud). Because of such centralized approach, the performance of the controller directly determines SDN scalability in such scenarios, which becomes crucial to the success of SDN ecosystem.

The implementations of controller is very diverse, with more than 30 controllers proposed by both industry and academic. These implementations use various programming languages, different run-time technologies, design approaches, and feature sets. The usability of controllers differs from conceptual prototype to production quality. In such a context, a performance evaluation is essential to understand these implementations and identify their bottlenecks, but also to select suitable controller for a given scenario. Number of works have proposed a partial performance evaluation in past few years. These works mostly proposed a new controller and utilized selected performance benchmark to verify their advantages over others. However, such advantages were only temporary, since other controllers were also evolving and improving in the meanwhile. Thus, the performance evaluation of controllers had a hard time in converging on one conclusion. Now, the majority of the controllers are mature enough in their development that is time to re-investigate their performance in a fair manner, in order to check what is the reality.

4.1 Centralized controller performance evaluation

Our evaluation targets exactly this goal: having a fair evaluation of the most relevant controllers, and providing an indication of which one is suitable in which scenario. The focus is on centralized controllers, namely: Ryu [40], Pox [3], Nox [2], Floodlight [10], and

Beacon [75]. Although several implementations of distributed controller, generally treated as network orchestrators, have been already proposed, they are still under heavy development. Hence, they are not ready for a performance evaluation aiming at providing long term results. Furthermore, Heller et al. [82] prove that one controller is often sufficient to meet existing reaction-time requirements by placing it at a suitable location.

In order to guarantee that the proposed results are fully reproducible, only open-source controllers and benchmark tools have been chosen. Moreover, our results are compared to previous works in order to have an insight on what has changed and why. However, unlike other prior evaluations, we go beyond the simple usage of benchmark tool an investigate the performance impact of systems settings (e.g., the use of Hyper-Threading, the interpreter) as well. Based on the outcome of the evaluation, it is helpful to derive some recommendations on which SDN controller is most suitable for which scenario.

4.1.1 Selected controller

Based on the requirement of open source and popularity, five major centralized controllers have been selected for performance benchmark evaluation, namely Ryu, Pox, Nox, Floodlight and Beacon:[1]

Ryu: Ryu [40] is a component-based SDN framework written in Python. Ryu supports various protocols including OpenFlow and Netconf. Ryu fully supports versions 1.0, 1.2, 1.3, and 1.4 of OpenFlow, as well as the Nicira Extensions.

Pox: Pox [3] is a networking software platform written in Python. Pox provides "Pythonic" OpenFlow interface and reusable sample components for path selection, topology discovery, etc.

Nox: Nox [2] is written in C++. It was initially developed side-by-side with OpenFlow and was the first OpenFlow controller. It has been the basis for research projects in the early exploration of SDN.

Floodlight: Floodlight [10] is an enterprise-class, Java-based OpenFlow controller. Floodlight is the core of a commercial controller product from Big Switch Networks. Floodlight can handle mixed OpenFlow and non-OpenFlow networks.

Beacon: Beacon [75] is a fast, cross-platform, Java-based OpenFlow controller that supports both event-based and threaded operation. Beacon runs on many platforms, from high end multi-core Linux servers to Android phones.

[1] Maestro (Java-based [103]) has not been updated since 2011, thus it has been excluded from the evaluation.

Table 4.1 SDN Controllers Summary

Controller	Ver.	Lang.	OF	Release	Thread
Pox	0.2.0	Python	1.0	10/2013	Single
Ryu	3.19	Python	1.0∼1.4	03/2015	Single
Nox	0.9.2	C++	1.0&1.3	02/2014	Mult.
Floodlight	0.90	Java	1.0	11/2012	Mult.
Beacon	1.0.4	Java	1.0	09/2013	Mult.

Table 4.1 summarizes the main characteristics of selected controllers. All controllers have (reasonably) chosen cross-platform languages (i.e., Python, C++, or Java). They also follow a modular design to exploit code re-usability. Their features highly depend on the chosen languages and related libraries. As can be seen, new releases of the selected controllers date back to one year ago, except for Ryu.[2] Both Ryu and Floodlight have support for OpenStack. Only Ryu supports the newest version of OpenFlow specification (v1.4). Ryu also provides complete unit tests, development documents, and compatible checks with existing OpenFlow-enabled switches.

As previously mentioned, network orchestrators (distributed controllers) are still at early stages, impeding an accurate performance evaluation. Nevertheless, for completeness, it is worth to mention two important orchestrator projects, namely OpenDayLight and ONOS:

OpenDayLight: OpenDaylight [28] is the biggest SDN collaborative open source project and is hosted by the Linux Foundation. The project goal is to accelerate the adoption of SDN and create a solid foundation for Network Function Virtualization (NFV). OpenDaylight is leading the transformation to Open SDN by uniting the industry around a common SDN platform.

ONOS: As the main challenger of OpenDayLight, Open Network Operating System (ONOS) [26] aims at high availability, performance and scale-out in real use cases of service providers. Its mission is to produce the Open Source Network Operating System that will enable service providers to build real SDN Networks.

As summarized in Table 4.2, OpenDayLight and ONOS are similar on most aspects. Because of their distributed nature, the design is much more complex introducing large overhead on overall performance and demanding a large amount of computation resources. Hence, they cannot be directly compared to centralized controllers and are excluded from the detailed performance evaluation. However, some discussion on their limited performance and known issues is provided in latter sections.

[2] BigSwitch has already released two new versions of Floodlight, v1.0 and v0.91, on 30/12/2014, but these new versions have compatible problems with Cbench, hence, v0.90 is used in this evaluation.

Table 4.2 Distributed Controller Summary

Orchestrator	Ver.	Lang.	OF	Release
OpenDayLight	Helium-SR3	Java	1.0&1.3	03/2015
ONOS	1.1.0			

4.1.2 Test Environment

The test environment is built on one server that has single Intel Xeon E5-1620 CPU, with 4 cores running at 3.6GHz. The operating system is Ubuntu 14.04 LTS with default networking configuration. For Java-based controllers the JDK used is OpenJDK (v2.4.7). For Python-based controllers, both CPython (v2.7.6) and PyPy (v2.4.0) have been used as interpreters for comparison. Since the benchmark tool Cbench only supports OpenFlow specification 1.0, all controllers have been configured in v1.0 mode. For simplicity, each tested controller only runs a layer 2 learning switch application. In order to unleash all potential of each controller, their configurations have been optimized according to the guidelines proposed by their official websites and development communities.

4.1.3 Cbench

Cbench (Controller benchmarker) [5] is a benchmark tool specifically designed for OpenFlow SDN controllers. Cbench emulates a configurable number of switches, which connect to a controller by sending request message *Packet_In* and watching for reply message $Flow\_Mod$. There are two modes in Cbench, namely "Latency" and "Throughput".

In Latency mode, each configured switch sends a single *Packet_In* message to the controller and waits for a $Flow\_Mod$ message, then repeats this process. The total number of responses received during test period is then used to calculate the average processing latency.

In Throughput mode, each configured switch constantly sends as many *Packet_In* messages as possible, in order to measure the maximum capacity of controller.

Jarschel et al. [84] argue on the limitation of Cbench, while proposing their own benchmark tool. Nevertheless, of the alternative benchmark tools, very few are open source or provide high availability as Cbench does. Furthermore, Cbench has been widely used and tested in both academic and industrial environments, hence, the choice of using such a tool.

4.1.4 Cbench Validation

As Cbench only acts as emulated switches, before using it for benchmark, it should be compared to real software switches in order to validate its results. To this end, OpenvSwitch

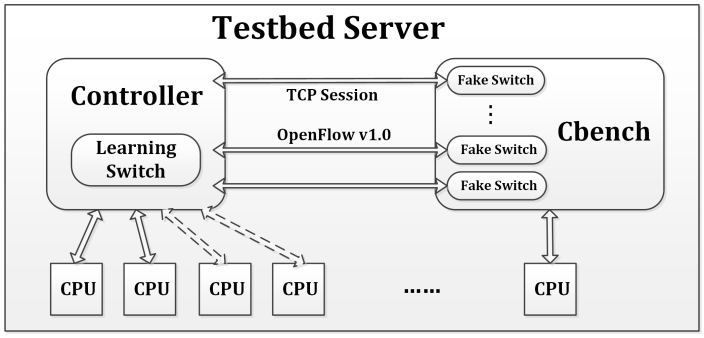


Fig. 4.1 Test Environment configuration.

has been chosen as a reference for comparison. OpenvSwitch has been configured to work in a similar way as Cbench, sending *Packet_In* messages to the controller and receiving *Flow_Mod* messages in return. The maximum throughput that the controller can support has been measured. Taking Ryu as an example, the result given by Cbench is 106K responses/millisecond, while it achieves only 103K in OpenvSwitch scenario. Because OpenvSwitch also sends out other types of messages besides *Packet_In* message, which slightly lowers the throughput. Considering that, the results can be regarded as in agreement with each other.

The difference between Cbench and a real software switch should be mentioned. Cbench's emulated switch is just kind of traffic generator that is able to send *Packet_In* messages as fast as possible, while the real switch usually has a smaller sending buffer and lower sending rate, since it needs to implement internal control logic. Because the focus of the evaluation is to benchmark the controller, rather than investigate the interaction between switch and controller, Cbench provides accurate results for this purpose.

4.1.5 Methodology

Normally, Cbench and controller software should be running on separated servers in order to emulate real scenarios. However, this requires multiple servers and high-speed dedicated network links in order to flood the controller. Due to hardware limitation of the available test environment, instead, in our case both softwares run on the same server. As a workaround, in each experiment linux kernel commands *isolcpus* and *taskset* are used to isolate CPU resources between Cbench and the controller under evaluation, as sketched in Fig. 4.1. The Cbench process is always attached to the same single core. The controller software is evaluated using a variable number of cores. Each emulated switch of the Cbench process connects to the controller through a TCP connection. Since all traffic goes through the local loopback, link bottleneck is eliminated and it is possible to fully focus on the impact of CPU

resources. It is worth to remark that there is always a saturation case, when the controller tries to use all of the CPU cores. In this case, the results show how, because of the CPU contention among controller, Cbench, and operating system, performance behaves.

4.1.6 On the Accuracy of Latency Measurements

In SDN framework, when switch receives the first packet of a new flow, it needs to ask controller for forwarding decision. This reactive behavior introduces a large latency overhead in the first packet of flows, compared with traditional proactive routing scheme. This overhead is composed by processing delay in both switch and controller as well as transmission delay on the link: $t_{overhead} = t_{controller} + t_{link} + t_{switch}$. In order to clarify the concept of latency, we choose OpenvSwitch as an example to explain it in a real scenario. Fig. 4.2 displays a simplified architecture of OpenvSwitch. As explained in Section 3.1 and 3.3.10, OpenvSwitch has a two-layer design, with a kernel module as fast path and a user-space daemon (ovs-vswitchd) for global control. Any unmatched packet arrived at kernel module will be first submitted to user-space daemon and then (if still no match exists) further submitted to the remote controller. Sending/receiving buffers exist on both kernel module and user-space daemon. These queuing systems introduce additional latency. Furthermore, there is also some other internal control logic in OpenvSwitch, which further increases the latency. All these factors contribute to t_{switch}.

The dominant latency factor differs depending on the specific scenarios. In datacenters where t_{link} is usually kept very small, t_{switch} is the main factor. However, t_{link} dominates in WAN scenarios. In our test environment, the Cbench emulated switches have no real queuing system and control logic. Hence, the latency result in Cbench is only responsible for controller and link aspects, while the influence from the switch side is overlooked. This means that the above equation can be simplified to $t_{overhead} = t_{controller} + t_{link}$. Furthermore, t_{link} is quite small in our test environment, because of the use of the loopback interface.

4.2 Evaluation results

We first explore the impact of environment settings, and then present the benchmark results following the order of complexity of configuration. All measurements are running for 500s and repeated 100 times. The relative standard error is always below 8% and hence, for the sake of clarity, not further discussed and represented in the various results.

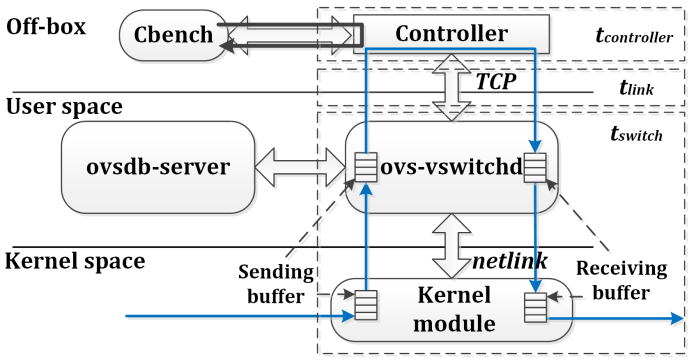


Fig. 4.2 Latency measurement with OpenvSwitch and Cbench.

4.2.1 Python Controllers and Python Interpreters

Python language has gained popularity due to its high code readability and concise syntax. In the context of SDN, there are several controllers developed in Python, including Ryu and Pox, which are two of the selected controllers. However, its inefficiency as a script language prevents its wide deployment in operation-intensive scenarios.

To improve the efficiency of the original CPython interpreter, an alternative interpreter, PyPy, has been developed in recent years. PyPy is a Python interpreter and just-in-time compiler which focuses on speed, efficiency and compatibility with the original CPython interpreter. To investigate the impact of the interpreter for the Python-based controllers, both of them have been used for Ryu and Pox. The obtained results are summarized in Table 4.3.

Concerning latency, PyPy achieves around 3.8 times speed up for both Pox and Ryu, when compared to CPython. When using CPython, Pox achieves only half of the throughput achieved by Ryu. Again, PyPy helps to boost the throughput performance of Pox and Ryu by 9.4× and 4.4× respectively. When using PyPy, Pox and Ryu achieve the same throughput at about 105 thousands responses per second. Because clearly PyPy improves the performance of Python-based controllers significantly, In the rest of the evaluation the results of Python-based controllers presented in measurement are all performed using PyPy.

Inspired by Python case, Java-based and C++-based controllers have been evaluated as well, with different versions of JDK or C++ compilers. The result shows their performance differences can be neglected and are not discussed any further.

4.2.2 Hyper-Threading

Hyper-Threading (HT) is Intel's proprietary simultaneous multi-threading implementation that is used to improve parallelism of computations performed on x86 class microprocessors. When enabled, for each processor core that is physically present, the operating system

Table 4.3 Pyhton Interpreter Impact

Latency (milliseconds)

Controller	CPython	PyPy	PyPy/CPython
Pox	0.156	0.042	3.75
Ryu	0.143	0.037	3.86

Throughput (responses/milliseconds)

Controller	CPython	PyPy	PyPy/CPython
Pox	11.2	105	9.38
Ryu	24.1	106	4.40

actually sees two logical cores, sharing the workload between them when possible. Enabling HT results in performance improvements of 15% in some cases, but also may result in performance degradation in some other cases. How much performance improvement can be achieved by HT highly depends on the software. In order to quantify the impact of HT on the controllers, the measurements are carried out twice, with HT disabled and enabled, so as to obtain a direct comparison.

4.2.3 Controllers Baseline

We start our measurement with most simple configuration, which is used as baseline. Each controller runs in single thread, and the number of Cbench emulated switches is set to one. HT remains disabled. Table 4.4 summarizes the throughput obtained in both latency mode (L) and throughput mode (T) for each controller. Note that "rps/ms" represents *responses per millisecond*, while "1/L" represents the actual latency (recall Cbench counts the number of sequential responses when in latency mode). Beacon performs much better than others in both modes. There is no doubt that Python-based controllers provide the lowest performance even using PyPy. Floodlight and Nox are close to each other, while Nox has a smaller latency. This ranking remains unchanged in most of the results presented in the following tests.

Remark that the ratio of T/L indicates the capability of the controller in handling multiple incoming packets simultaneously. This capability depends on comprehensive effects of language features, related libraries, as well as software design. Beacon outperforms by far the others. Floodlight, the closer competitor, showing less than half the performance of Beacon.

4.2.4 Distributed Controllers Baseline

Although OpenDayLight and ONOS are designed for distributed platform, it is still meaningful to measure their performance on single node. We follow the same configuration as for the controllers baseline. As shown in Table 4.5, the overall throughput of both controllers are at

Table 4.4 Controllers Baseline

Controller	L (rps/ms)	T (rps/ms)	T/L	1/L(ms)
Pox	24	105	4.38	0.0416
Ryu	27	106	3.93	0.037
Nox	56	687	12.2	0.0179
Floodlight	45	670	14.89	0.0222
Beacon	61	2302	37.74	0.0164

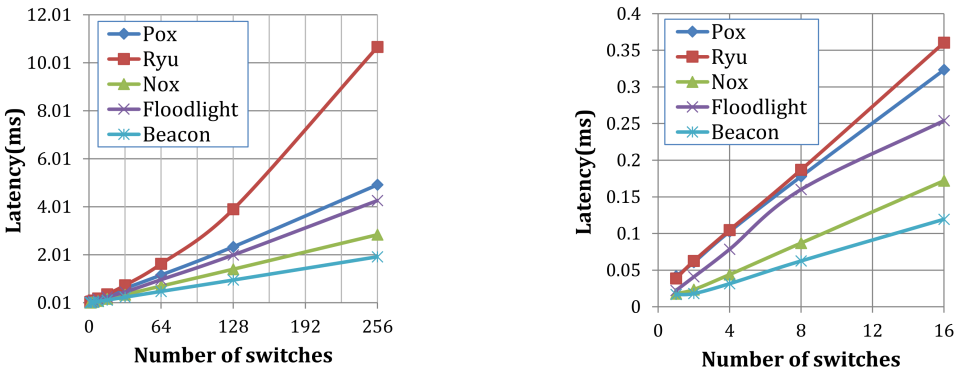
Table 4.5 Distributed Controller Baseline

Controller	L(rps/ms)	T(rps/ms)	T/L	1/L(ms)
OpenDayLight	9	8	0.888	0.111
ONOS	38	49	1.29	0.026

least one order of magnitude smaller than Java-based controllers (i.e., Beacon and Floodlight). OpenDayLight is built on multi-core processing model and needs a large amount of CPU resources. Clearly the single core scenario significantly constrains its performance. ONOS performs far better than OpenDayLight, but still considerably worse than others. During our measurements, number of issues appeared for both OpenDayLight and ONOS. First, both show unpredictable performance degradation when running for a long time. Second, when increasing the number of switches, their overall throughput drops significantly. For instance, ONOS achieves only 8 rps/ms with 16 switches. Third, memory leaks appear under high volume flooded traffic. These problems imply that distributed controller measurements would not provide accurate long term result. Furthermore, due to their complex architecture and functionality, metrics other than Cbench-based latency and throughput should be considered, which is to be discussed in the next section. Because of the above, in the rest of this section we only focus on controllers.

4.2.5 Number of Switches

After getting the baseline performance of each controller, it is time to look at more complex configurations. In this part, the number of emulated switches is changed while keeping other parameters fixed. Fig. 4.3 shows the average per-switch latency for different numbers of switches. Fig. 4.3a shows the overall behavior, while Fig. 4.3b zooms in the case of small number of switches. When the number of switches increases, the latency increases approximately linearly. The latency rises from 0.01ms when there is only one switch, up to more than 10ms with 256 switches. When multiple switches are connected, one TCP connection per switch is maintained, on which the controller usually adopts a Round-Robin poll policy, thus the linear relationship. In the range of small numbers, we see some tiny



(a) Overall (b) Zoom

Fig. 4.3 Per-switch latency with different numbers of switches (single thread).

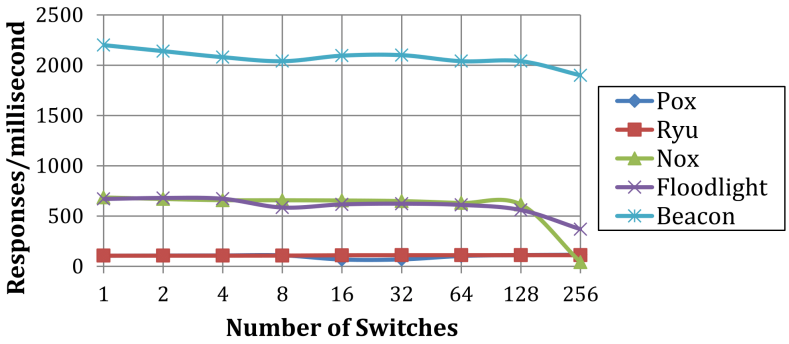
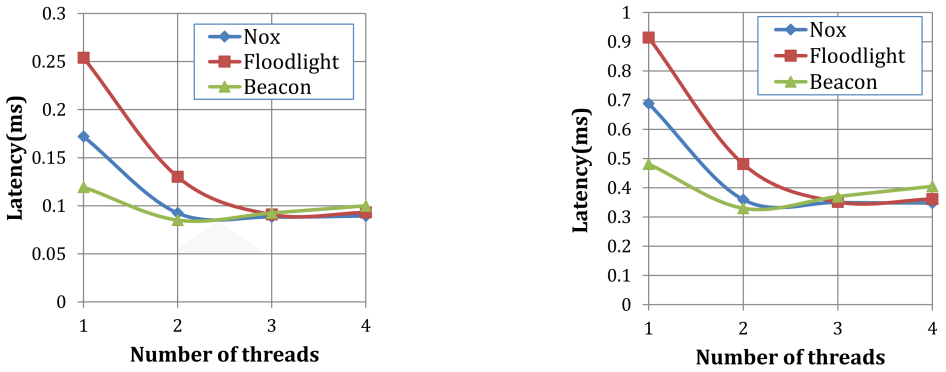


Fig. 4.4 Throughput achieved with different numbers of switches (single thread).

deviations on Floodlight and Beacon, which is mainly due to the default configuration of working pool in the software.

The total throughput (responses/ms) achieved with different numbers of switches is shown in Fig. 4.4. The performances are relatively stable regardless of the number of switches when their number is under 128. Because in throughput mode, the receiving buffer of controller is always full no matter how many switches are connected. When the number reaches 256, a performance degradation appears on all controllers except the Python-based ones. This implies that the maintenance of a large number of connections is expensive. Especially for Nox, which is totally stuck, implying that Nox is not capable to handle a large number of overloaded connections. There is a significant fluctuation around 16 and 32 for Pox, which is due to some compatibility issues of PyPy and needs further investigation. Beacon remains the best performing.



(a) 16 switches (b) 64 switches

Fig. 4.5 Per-switch latency with different numbers of threads (HT-disabled).

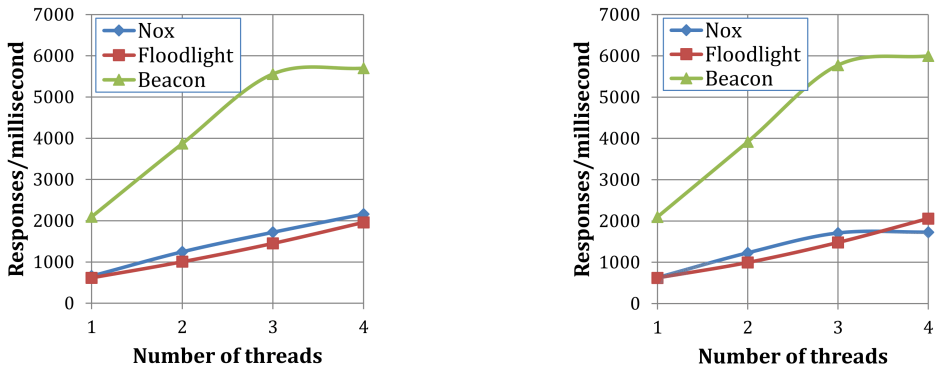
4.2.6 Threads Number – HT disabled

In this part, the impact of the number of threads used by the controllers is investigated, with fixed number of switches. Two numbers of switches is used, 16 and 64, in order to cover different deployment scales. Hyper-threading is disabled in this test. Pox and Ryu are not evaluted because they run only in single thread.

Fig. 4.5 shows the average per-switch latency with different numbers of threads. The results in both 16 and 64 switches cases have the same overall trend, but are different in the latency value. Note that only the results from 1 thread to 3 threads are valuable in analyzing the impact of threads. The result with 4 threads is the saturated case described in Sec. 4.1.5, since only 4 cores are presented.

With the increase of threads, the per-switch latency decreases on all controllers. Floodlight and Nox reach their lowest latency with 3 threads. While Beacon behaves quite differently, it achieves lowest latency with only 2 threads. The lowest per-switch latency of each controller is close to each other and around 0.35ms for the 64-switches case and slightly less than 0.1ms for the 16-switches case.

The results for the throughput are presented in Fig. 4.6. For the interval from 1 to 3, as expected, the performance increases approximately linearly with the increase of threads. In both 16 and 64 switches cases Nox and Floodlight behave almost the same, both going from less than a thousand responses/ms up to around two thousand responses/ms. Beacon can achieve up to 6 million responses per second with 3 threads in the 64-switches case (slightly less for the 16-switches case), with no improvement when using four cores. Except for Beacon, in the throughput case it looks like other controllers achieve different degrees of performance gain with 4 threads compared with 3 threads. This means that the controllers are able to use part of the fourth core, even when competing with Cbench and the operating



(a) 16 switches (b) 64 switches

Fig. 4.6 Throughput achieved with different numbers of threads (HT-disabled).

system, which is the opposite of the result of latency mode. On the one hand, this is due to the fact that, in throughput mode, Cbench uses less CPU compared with latency mode, since the blocking method in latency mode is a CPU consuming operation. On the other hand, the controller has much more traffic to be handled than in latency mode, which leads to a more aggressive demand of CPU resources. From such a point of view, Floodlight appears to be the best at *stealing* CPU resources from Cbench and the operating system.

4.2.7 Threads Number – HT enabled

To further investigate the impact of the number of controllers' threads, HT is enabled in this test, so to evaluate its effectiveness. With HT enabled, each single physical core acts as two logical cores. Similar to previous setting, we let Cbench and operating system run on one logical core and controller on other cores. When setting the number of threads to 7, the controller and Cbench, while running on separate logical cores, are actually sharing one single physical core. So this represents the saturated case presented in Section 4.1.5. For such a reason, there is no use in exploring the performance in the case of eight threads. When adding logical cores to the controller, a fixed order is followed. Since there are 4 physical cores, providing 8 logical cores, they are represented as "(1,2)(3,4)(5,6)(7,8)", which means logical core 1 and 2 are on the same physical core. The adding order is 1-2-3-4-5-6-7. This order is helpful to observe how much gain is achieved by enabling HT. In this test, the number of switches is set to 64.

The average per-switch latency with different numbers of threads is shown in Fig. 4.7. This result is quite different from Fig. 4.5b and contains no clear trend, while varying in the same range of values. Both Floodlight and Nox achieves the lowest latency with 3 or 4

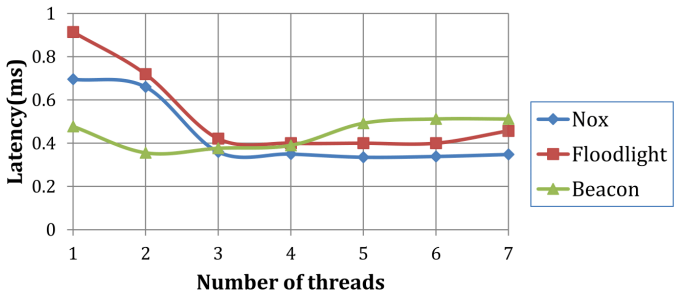


Fig. 4.7 Per-switch latency with different numbers of threads when Hyper-Threading is enabled (64 switches).

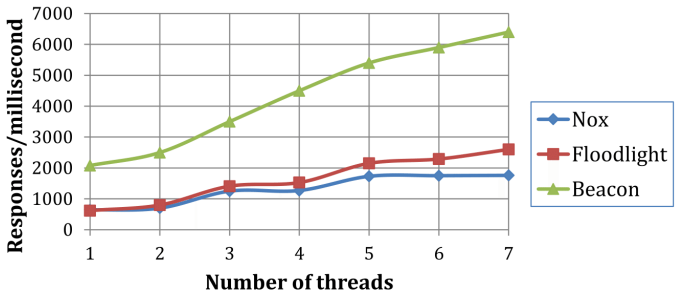


Fig. 4.8 Throughput achieved with different numbers of threads when Hyper-Threading is enabled (64 switches).

threads. Beacon fluctuates slightly all the way. Compared with Fig. 4.5b, we see that the results with HT-enabled is actually slightly worse than HT-disabled.

Fig. 4.8 shows the result of the total throughput with different numbers of threads. Nox and Floodlight show a clear stair-like curve, which means the results between 1 and 2, 3 and 4, 5 and 6 are quite close. This indicates that adding one more logical core from the same physical core can only gain limited performance. Beacon instead shows a behavior close to linear. Compared with the results in Fig. 4.6b, both Beacon and Floodlight achieve around 10% performance gain in average, while Nox has negligible improvement. Thus HT is actually useful for Java-based controllers, but not that much for C++-based controllers.

How to map working threads to cores is critical in multi-thread programing. The incorrect setting may result in performance degradation. Nox and Beacon set thread parameter manually, while Floodlight set it automatically. It is normally recommended to set the number of thread as same as the number of logical cores.[3] Nox is an exception in our measurements, it needs to be set as the number of physical cores to achieve the best performance even when

[3]Recall that when disabling HT, logical core is just the physical core.

Table 4.6 Latency Comparison

Controller	Latency(ms)		
	Empty buffer	Full buffer	Ratio
Pox	0.0417	5.26	126
Ryu	0.0385	2.63	68
Nox	0.0179	149	8358
Floodlight	0.022	76.9	3461
Beacon	0.0163	50	3050

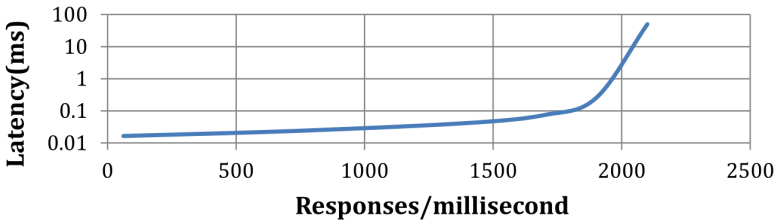


Fig. 4.9 Correlation between throughput and latency in Beacon.

enabling HT. When deploying controller in virtual environment, thread parameter should be selected with more cautions, since the relationship among vCPUs is more complex.

4.2.8 Correlation between Throughput and Latency

In the previous part of the evaluation, the latency is measured with an idle controller. This means that the receiving buffer of the controller is empty and any new arrived packet can be handled immediately. But when the workload increases, the buffer will be filled gradually. This results in additional latency by introducing queuing time in buffer. To measure the latency under heavy workload, an extreme scenario has been chosen, where the receiving buffer is always full. Table 4.6 shows the latency measured with full buffer compared with empty buffer scenario. We see that the latency can be enlarged by thousands times.

Although the latency is unacceptable when buffer is full, the throughput reaches to its peek at the same time. So there is a trade-off between latency and throughput. Fig. 4.9 displays the correlation between throughput and latency in Beacon.[4] We see that Beacon is able to keep latency in a reasonable range (<0.1ms) as well as achieve high throughput (around 1750 rps/ms). This is important for network operators to balance latency and throughput according to practical needs.

[4]Due to the limitation of our platform, the correlation patterns of other controllers are not included.

Table 4.7 Throughput Variation among Multiple Switches

Controller	Pox	Ryu	Nox	Floodlight	Beacon
Variation(%)	3	<1	>50	23	11

Table 4.8 Time in Achieving Fairness

Controller	Pox	Ryu	Nox	Floodlight	Beacon
Time(s)	19∼21	<1	4 or ∞	5∼6	3∼4

4.2.9 Fairness

When connected with multiple switches, the controller should process packets from each switch in a fair manner. Especially when the given workload is larger than the capacity of the controller. Table 4.7 shows the relative standard deviation of throughput among 128 switches connected to the same controller. Python-based controllers show extreme good fairness, followed by Java-based controllers. The fairness variation on Nox is much larger than others.

We further design a specifically crafted scenario to evaluate the speed to achieve fairness. The measurement starts with 128 switches in order to overload the controller. When all the switches are already equally and fairly served, a new emulated switch is added. The time needed by new switch to be served equally as other switches can be used as an indicator of fairness. The smaller the time, the faster fairness the controller achieves. From the measurements in Table 4.8, it results that Ryu is the best again, followed by Beacon, then Floodlight, then Pox. Nox behaves unpredictably, since the new switch has a 0.5% chance to be not served.

4.2.10 Comparison with previous works

Previous works on SDN controller performance benchmark is strongly related to the history of the development of controllers. Hence the versions of controllers and the results are quite different from each other. Table 4.9 presents a summary of how the evaluation presented in our result compares to selected previous works. Although these works have chosen different test setup, they all lead to the conclusion that Beacon is the best performing. The benchmark of Floodlight is inconsistent with our result, because previous work did not use a minimal, optimized configuration. Nox selected in [116] turned out to be the older version that did not support multi-thread.

Table 4.9 Comparison With Previous Work

Throughput (rps/ms) with 3 threads

	[75]	[116]	[117]	Our Results
Beacon	2500	3300	3800	6000
Floodlight	400	900	600	2100
Nox	1800	400	1000	1850

4.3 Distributed controller synchronization

In SDN, distributed controller [93] (also known as controller cluster [26][28]) is designed to improve the resilience and scalability of the control plane, since a centralized controller would be a single point of failure and presents scale-out limitations. Even if multiple controllers are geographically distributed, they still act as a centralized control plane. In order to function correctly and consistently, the synchronization and coordination among multiple controllers is critical for the whole system. As discussed in previous section, the performance evaluation of distributed controller at large scale requires considerable amount of computation resource which is not practical to our testing platform[5]. Instead, we mainly focus on the synchronization and coordination among multiple controller nodes. We study the characteristics of synchronization traffic as well as the latency introduced by the coordination among multiple nodes.

4.3.1 Synchronization in in-band scenario

Synchronization generates a type of controller-controller traffic other than traditional controller-switch control traffic. In an in-band scenario, without a dedicated out-of-band control network, the control traffic and data traffic are sharing the same underlying network links and ports. Thus, they may interference with each other and this impact is still unknown to network operators. For the afore-mentioned reasons, analysis and evaluation of in-band control traffic is paramount in identifying key issues and assessing their impact.

Since ONOS [26] cluster has been already successfully deployed in various scenarios, we selected it to investigate interference among different types of traffic. A controller cluster is essentially a distributed system following Brewer's CAP theorem [64], which basically states how it is impossible to simultaneously provide all three guarantees: $Consistency(C)$, $Availability(A)S$, and $Partitiontolerance(P)$. Because usually **P** is a must, there is a trade-off between **C** and **A**. In the case of ONOS (v1.1.0), Hazelcast [43] is used for clustering, which

[5]The official development communities of OpenDayLight and ONOS have released several basic evaluation on large scale deployment.

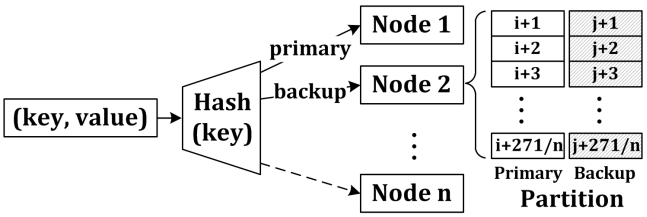


Fig. 4.10 Data partitioning in Hazelcast

is rather an **AP** system (i.e., favoring Availability in the event of a partition). By default, Hazelcast has 271 partitions (i.e., memory segments where data entries are stored) distributed equally among cluster nodes.[6] Fig. 4.10 shows how data is partitioned and stored. In order to store a given $(key, value)$ pair, first the key is serialized and then passed to a hash function to get its partition number. At the end, the value is stored on the node which the partition number belongs to. For reliability, there is also a copy of the value stored on a different node as a backup. ONOS uses the above feature to actually store various data (including hosts, devices, flows, intents, etc.), which, hence, is distributed among the controllers, thus generating synchronization traffic, in order to guarantee consistency.

4.3.2 Synchronization traffic characteristics

In order to simulate a in-band ONOS cluster scenario, Mininet [20] and Docker [8] are combined and modified[7] to satisfy following requirements: i) OpenvSwitch runs in in-band mode in an isolated network namespace; ii) ONOS runs in Docker container; iii) a general simulation tool needed to set up the topology, also providing a generic unified API; iv) a packet capture system so to monitor the control traffic. The measurement is based on a linear 10 switches topology with 10 hosts on each switch. The switches are evenly distributed to controller nodes.

By default, ONOS creates a logically full-mesh cluster. The cluster-head is the oldest controller in the cluster, which periodically sends the partition table to other controllers. When a new controller joins/quits the cluster, re-partitioning is required. We measure the volume of data the cluster-head sends to each controller in case of a new controller joining. This is shown in Fig. 4.11, with the relationship between the number of controllers and the volume of traffic they receive from the cluster-head. The more the controller the lower the amount of data sent to each of them, however, by multiplying the traffic volume by the controller number, the results tend to be a constant value. This indicates that a fixed-size data of partition table is distributed evenly among controllers, every time the cluster members

[6] Hazelcast uses 271 because of its DHT internals when distributed among less than 100 nodes.
[7] For more details about this demo scenario, please refer to Appendix A.

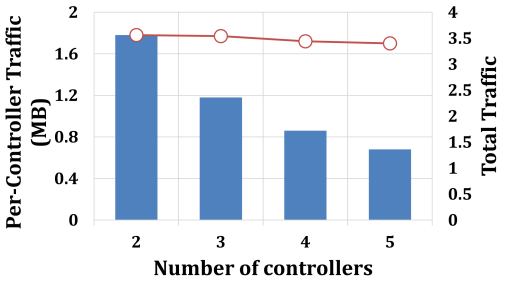


Fig. 4.11 Joining: Per-controller (bars) vs Total (dash line) traffic.

Table 4.10 Traffic vs. primary controller (Mbps).

Primary	$c1 \leftrightarrow c2$	$c1 \leftrightarrow c3$	$c2 \leftrightarrow c3$	Total
c1	31.6	30.1	8.6	70.3
c2	89.6	12.3	27.0	128.9
c3	14.2	90.3	21.3	125.8

change. This re-partitioning duration is usually kept small (<200ms), and the peak rate can go up to 40Mbps.

We analyzed as well the cross-correlation of traffic between different controller pairs. The traffic between the primary (**c1**) and two backups (**c2** and **c3**) is highly correlated (98% coefficient), which is reasonable because of **c1** cluster-head role. The traffic between two backups is correlated as well with above traffic, but with a coefficient of only 80%.

In an idle state, the average controller-controller traffic rate is below 5Mbps. While in a heavy-loaded scenario where thousands of *Packet_in* messages are received by controller, the traffic rate reaches to 90Mbps. Table 4.10 shows the average traffic (with full-loaded controller) between each controller pair when a different primary controller is chosen. We can see that the traffic patterns are significantly different due to the data partitions. When **c1** is primary, the overall traffic is much smaller than other cases. Clearly further research on the selection of the primary controller is necessary to understand and optimize the synchronization traffic.

Based on above results, the total synchronization traffic (90Mps for worst case) is considered to be negligible compared to today's bandwidth (over 10Gbps). Hence, the impact of the control traffic on data traffic is minimal. On the other side, the impact of the data traffic on control traffic can be alleviated by adopting higher priority queue for control traffic.

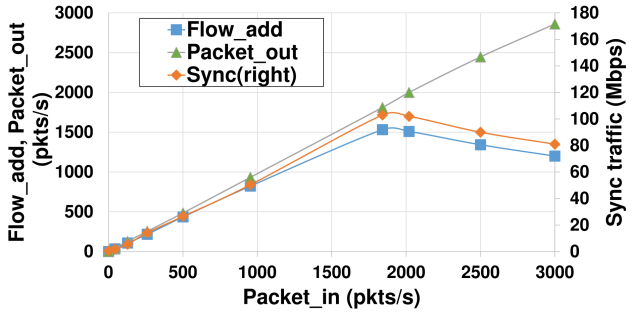


Fig. 4.12 Relationship among various control messages.

4.3.3 Control traffic contention

We further examine the conflict among various types of control traffic. In ONOS cluster, when the controller receives a *Packet_In*, it first generates a *Flow_Add* and then a *Packet_Out*. For certain packet, e.g., ARP broadcast, it is unnecessary to add a new flow rule, so only *Packet_Out* is generated. Controller cluster needs two-layer synchronization. The first layer is betweeen the controller and switches: when the controller adds a new rule, besides *Flow_Add*, a *Barrier_Request*[8] is also sent to the switch immediately after *Flow_Add*. The controller then waits for a *Barrier_Reply* to confirm the installation of the rule. The second layer is among controller nodes: once confirmed that the rule is already installed, the controller starts to synchronize this rule message with other nodes.

Fig. 4.12 shows the relationship among various control messages when different *Packet_In* workloads are given. As expected, the synchronization traffic is roughly proportional to *Flow_Add* all the way. Because any new installed flow triggers synchronization. The ratio between synchronization traffic and the number of *Flow_Add*: 60Kbps per *Flow_Add*. *Packet_Out* is always approximately equal to *Packet_In*.

When *Packet_In* rate is below 1800 pkts/s, *Flow_Add* rate is proportional to *Packet_Out* and a bit lower. However, when *Packet_In* rate is over 1800 pkts/s, both *Flow_Add* and synchronization traffic start to decrease along with the increase of *Packet_In*. Because the controller has reached the upper limit of its processing capacity, and there exists resource competition between *Packet_Out* and *Flow_Add*. It usually takes hundreds of microseconds for switch to install a new rule, which is much slower than processing a *Packet_Out* message. Furthermore, there exists certain percentage of *Packet_In* that only results in *Packet_Out* (i.e., ARP broadcast request). Hence, it is reasonable that *Packet_Out* can win over *Flow_Add* in competition. Since the synchronization traffic is positively correlated to *Flow_Add*, it is also impacted by overloaded *Packet_Out*. This reveals the fact that there is no priority among

[8] A barrier request can be used by the controller to set a synchronization point, ensuring that all previous state messages are completed before the barrier response is sent back to the controller.

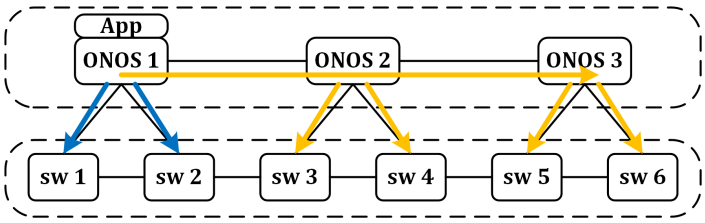


Fig. 4.13 Intent installation testbed

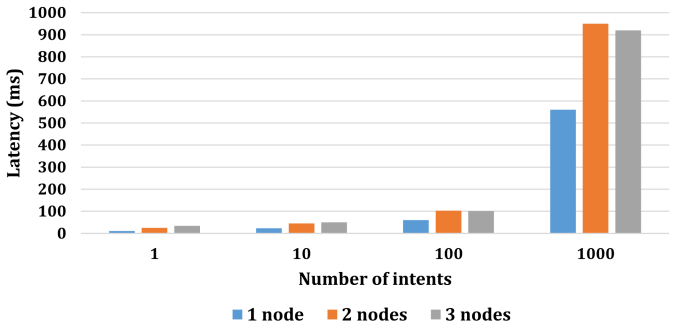


Fig. 4.14 Intent installation latency

various control messages. All these messages are handled sequentially and equally in both controller and switch. However, the processing order of control messages can impact the total throughput in an indirect way, especially when the given workload is larger than the capacity of the controller.

4.3.4 Coordination latency

The coordination among multiple controller nodes introduces additional latency to install flow rules, especially when the switches belong to other controller nodes. Fig. 4.13 shows a controller cluster testbed with 3 nodes. 6 switches are linearly connected, and each node controls 2 switches as shown in the figure. “Intent” is the concept defined by ONOS which specifies the network control desires in the form of policy rather than mechanism. ONOS can translate the intent into a set of flow rules to be installed on related switches. For example in our testing case, the test script that is used to generate batches of “intents” is running on ONOS1. This intent is to allow that any host on sw1 and sw6 can communicate with each other. According to the definition of the intent, ONOS1 should generate a set of rules to build the bidirectional connection between sw1 and sw6. In ONOS cluster, only the node that directly controls the switch can install rules on it. Hence even if the rules for sw3 and sw4 are generated in ONOS1, they have to be first delivered to ONOS2 and then installed by ONOS2. This behavior enlarges the latency to install rules. In order to evaluate this latency,

we compare the latency results with different number of controller nodes. Fig. 4.14 shows the latency results to install different number of intents with different number of nodes. As expected, larger latency exists in a multiple nodes scenario compared with standalone node due to the coordination among nodes. For large size of intents (i.e., 100 and 1000), when the cluster size increases, the latency tends to decrease. Because each node only processes a smaller number of intents due to distributed workload.

4.4 Summary

In this chapter, a comprehensive performance evaluation of five major open-source SDN controllers is first carried out. The measurements have been set up to be fair and easily reproducible. Beyond a simple benchmark, general system wide settings like Python interpreter and Hyper-Threading are also examined in order to evaluate their impacts on controllers performance. We also design several crafted scenarios to provide a comprehensive understanding of controller performance and its limitation.

Based on the evaluation, some general conclusions can be derived. Beacon wins on almost every aspect. It is very fast, also providing good fairness. As the first SDN controller, Nox still suffers from scalability and fairness issues. Floodlight ranks as average, although it is marked as enterprise-class. With the help of PyPy, both Pox and Ryu can handle 100K packets per second, which is sufficient in a large number of scenarios. Ryu is more competitive due to its active development community. Its master branch code is updated weekly to support latest OpenFlow specification. The combination with new technology as OpenStack [34] and Docker [8] helps Ryu to evolve with new features more quickly than others. From an academic perspective, each controller is a good starting point for SDN related research. While in industrial community, Floodlight and Ryu are more popular due to their rich set of features. The performance is no longer the only dimension in choosing controllers. The usability, reliability and security are equally important. Since the functionalities of controller are growing more complex in a fast pace, it is hard for independent controller developer to follow, and a large community is usually needed. OpenDayLight is founded by Linux Foundation; NTT is behind Ryu; ONOS also has close cooperation with leader companies in the communications industry. During measurement, we also see a clear limitation of centralized controller in large scale scenarios, which implies that the distributed controller is necessary for SDN further development and deployment.

Different from centralized controller, distributed controllers are designed to suit large scale network as well as provide fault tolerance. They are gaining momentum, with number of projects already existing, e.g., Onix [92], OpenDayLight [28] and Open Network Operating

System (ONOS) [26]. Distributed controller is supposed to run on a cluster of servers through mutual synchronization and coordination, which makes their performance evaluation much more complex than centralized ones. Hence, instead of carrying out an enhanced performance evaluation on distributed controller, we mainly focus on the synchronization and coordination behaviors among controller nodes. We use an in-band control scenario to measure the characteristics of synchronization traffic. Although the result indicates that the total volume of synchronization traffic can be ignored compared to today's bandwidth, there exists conflict among various control messages, which lowers the efficiency of synchronization. This implies that the control message should be associated with a priority when processed by the controller and switches. The distributed controller can increase the total throughput by processing simultaneously on multiple nodes. Meanwhile, the latency due to coordination and synchronization is also enlarged. Based on these facts, a more efficient and effective east/westbound API among controller nodes still needs further investigation.

Chapter 5

Fine-grained Resource Control

Software switch brings more flexibility in network management and deployment compared with dedicated hardware solution. For instance, when combined with SDN, software switch provides a centralized flow-based forwarding scheme by decoupling data plane and control plane; in NFV, software switch is the basis to provide customized network service (or service chain) dynamically on demand regardless underlying physical topology. Furthermore, software switch stimulates the development of open-source "White box" switches that are built on generic hardware platforms such as commodity servers, which is helpful in reducing both OPerational EXpenditures (OPEX) and CAPital EXpenditures (CAPEX).

However, the performance of software switch built on commodity server is not as fast and stable as dedicated hardware. Besides the overhead introduced by the network stack in Linux kernel, there exist intensive CPU contention among software switches, operating system and other services. For example, the packet processing load can be concentrated on a particular CPU core. To address this problem, Receive Side Scaling (RSS) [41] is designed for NIC to distribute the load into multiple CPU cores. But RSS is vendor-specific technology which mainly focuses on hardware interrupts. Similar study can be found in [119] which also proposes a generic load balancing framework for multi-core system and multi-queue NIC. Although Virtual Switch Extension (VSE) [101][102] can adaptively distribute SoftIRQ requests on multiple cores based on CPU load, it is designed to manage only one instance of virtual switch and provide no QoS control. While in NFV scenario, it is common that multiple software switches and virtual network functions coexist on the same server, and each switch and service should be granted a certain amount of CPU resource to fulfill its functionality. Furthermore, as mentioned in Section 3.3.8, modern CPUs have features to change their running frequency adaptively according to workload. Thus the performance of software switch can not be kept consistent due to changeable frequency. Even if the CPU

frequency is locked, the background services of operating system are running periodically or sporadically, which also impacts the total performance as described in Section 3.3.1.

For above reasons, fine-grained resource control is important to the success of software switch deployment. It not only helps to deliver stable and predictable performance to meet user-defined QoS requirements, but enforces the allocations among contended resource in order to minimize the interference as well as maximize the overall performance. In this chapter, we first explore the general packet processing in Linux to understand the background of resource contention. The related resource control mechanisms in Linux are further explained. Then we implement a prototype of Service Function Chaining (SFC) architecture where multiple switches and service functions are required to coordinate with each other. In SFC scenario, we examine the effectiveness of fine-grained resource allocation. Finally, combined with previous studies on software switch performance evaluation, we propose an automate resource allocation runtime by introducing classical control theory. This runtime not only aims to orchestrate multiple virtual service nodes to boost the overall performance, but also provides fine-grained control to satisfy SLA requirements adaptively without overprovision.

5.1 Resource contention and allocation

In cloud or data center where virtualization technology is widely deployed, multiple virtual machines (VM) share the same underlying server to run customized compute-intensive applications, and resource contentions among VMs can not be avoid. The emerging technologies like network virtualization and NFV are facing the same resource contention problem. In order to guarantee the required network performance and network functionality, multiple software switches and virtual network functions are consolidated and orchestrated.

5.1.1 Received packet processing in Linux

Fig. 5.1 shows the general packet processing in Linux kernel. When the packet arrives, the hardware NIC first raise a HardIRQ (Hardware Interrupt ReQuest) to notify a specific CPU core. Then the packet is transferred from the NIC into the ring buffer inside Linux kernel. After that, a SoftIRQ (Software Interrupt ReQuest) is issued on the same CPU core. Next, the kernel TCP/IP network stack handles the packet according to its header space. Finally, the packet is passed to process-level socket buffer to be further processed by targeted network application. The packet processing can be divided into 3 steps: HardIRQ, SoftIRQ and Process handling. In fully virtualized testbed where all traffic load is generated and

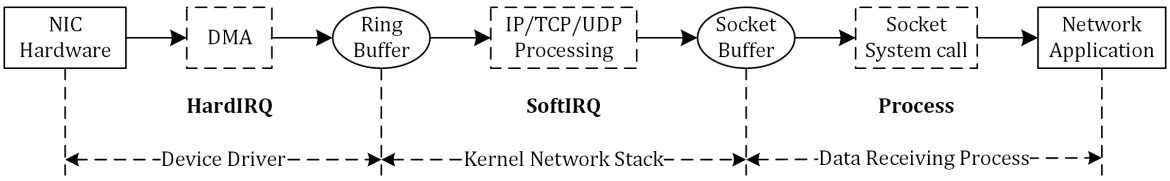


Fig. 5.1 Packet processing in Linux Networking

consumed inside the same server, HardIRQ step can be excluded, since no packet is received and processed by hardware NIC. Thus we only focus on SoftIRQ and Process steps.

By default in Linux, SoftIRQs and processes are distributed among multiple cores dynamically and randomly by operating system. This introduces additional overhead for copying process contexts among cores and prevents flexible and accurate control on CPU resource. For HardIRQ, DMA is responsible for memory copying without involving CPU. While SoftIRQ is fully performed by CPU. Especially when numerous SoftIRQs concentrate on the same core, the overall performance significantly degrades due to locking behavior. Since SoftIRQ has higher priority than normal process, intensive SoftIRQ processing may also prevent the execution of normal processes. For user-space software switch, it is necessary to guarantee the CPU resource for user-space processing as well as SoftIRQ.

5.1.2 CGroups and CPUFreq

CGroups (Control Groups) [6] is the Linux kernel feature that is used to control resources usage of single process or a collection of processes. It is designed to provide a unified interface to realize fine-grained control over allocating, prioritizing, denying, managing, and monitoring various system resources (e.g., CPU, memory, disk I/O, network, etc). CGroups is also the foundation for operating system-level virtualization such as Docker [8], LXC [47], OpenVZ [35], etc.

In CGroups, a subsystem (also called resource controller) represents a single resource, such as CPU bandwidth or memory usage. There are two main subsystems that are related to CPU control, namely “cpu” and “cpuset”. “cpu” is used as the scheduler to provide CGroup tasks access to the CPU. It supports two types of schedulers: Completely Fair Scheduler (CFS) is a proportional share scheduler which divides the CPU time proportionately among multiple processes (or process groups) according to the priority assigned to each process; Real Time scheduler (RT) allows to explicitly specify the total amount of CPU time that real-time tasks can use. “cpuset” is used to specify individual CPU cores and memory nodes that specific tasks can access.

CPU frequency scaling enables the operating system to scale the CPU frequency adaptively in order to save power. CPU frequencies can be scaled automatically depending on the system load, in response to ACPI events, or manually by userspace programs. CPU frequency scaling is also implemented in the Linux kernel, which is called "CPUFreq". CPUFreq allows to change the clock speed of CPUs on the fly by user-space programs.

The combined use of CGroups and CPUFreq can achieve fine-grained CPU control. In order to fully control the CPU resource of software switch, we use OFsoftswitch instead of OpenvSwitch. Because as a kernel implementation, SoftIRQs in OpenvSwitch are managed and distributed by kernel without any explicit API for user-space. While SoftIRQs in OFsoftswitch are always associated with its user-space process, which allows us to manage the CPU usage of OFsoftswitch through its user-space process. Furthermore, more and more projects (e.g., DPDK [14], mTCP [87], etc.) are creating a new network stack fully in user-space to replace the default kernel one for fast processing. Hence, OFsoftswitch is chosen as a representative of user-space implementation in the following sections.

5.2 Resource allocation for Service Function Chaining

We extend our vision from the performance of individual software switch to global optimization among multiple switches based on fine-grained resource allocation. The emergence of SDN and NFV is accelerating the development and deployment of software switches and virtual network functions to satisfy various stringent requirements of network softwarization. Service Function Chaining (SFC) is widely recognized as an important and promising application under this background. Hence, we extend OFsoftswitch to support SFC and demonstrate the effectiveness of resource allocation in SFC scenarios.

5.2.1 Service Function Chaining (SFC)

The delivery of end-to-end services often requires various network functions to provide key security, network management and performance guarantees. These network functions include traditional firewalls and IP Network Address Translator (NAT), as well as other application-specific functions. A service chain is defined as an ordered set of abstract service functions that must be applied to packets/flows selected as a result of classification. The concept of Service Function Chaining (SFC) [78][95] allows to steer service-specific traffic to traverse network service functions in the given order. SFC refers to decouple the deployment of network functions from underlying infrastructures.

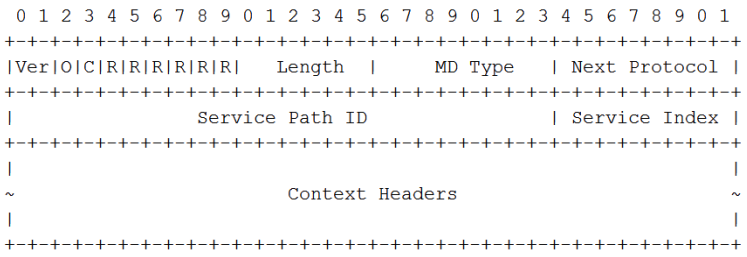


Fig. 5.2 Network Service Header Format

The current deployment model of network service functions still suffers from being static, which is tightly coupled to underlying network topology and physical resources. This constrains the flexible and dynamic service delivery and potentially inhibits the network operator optimizing their service resources. Instead of static service insertion that requires modifications on topology and routing, service functions are treated as virtual resources in SFC with associated attributes that is available for scheduled consumption. The specific traffic can be steered to the requisite service functions according to predefined policies, along with metadata information to realize policy enforcement. Service overlay is introduced to implement SFC. Service overlay first classifies incoming traffic based on policy rules, then encapsulates the packets in a network transportation header and delivers them to the initial service function in a service chain. The packets must be re-classified at each service function node and then steered to the next one in the chain. Network Service Header (NSH) defines the header format to create a dedicated service overlay which is independent of underlying transport layer.

5.2.2 Network Service Header (NSH)

Network Service Header [77] is the shim layer added to a packet or frame that is used to create a service overlay. The packets and the NSH are boths encapsulated by an outer header for transport. NSH is transport agnostic. Thus it can be imposed between the original packet and various outer network encapsulations such as MPLS, VXLAN or GRE.

Fig. 5.2 illustrates the format of NSH. First 4 bytes are defined as a “Base Header”, next 4 bytes are “Service Path Header”, and then either 16 bytes predefined “Mandatory Context Headers” or “Optional Variable Length Context Headers” are attached. Base header provides basic information of service header and payload protocol. More specifically, the “Ver” (version) field occupies 2 bits and is set to 0x0 by default. “O” bit indicates whether this is an operation and management (OAM) packet. “C” bit indicates whether a critical metadata TLV in optional context headers is present. All other flag fields are reserved for

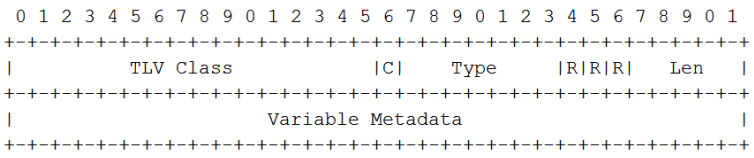


Fig. 5.3 TLV format of optional context header

future use. The "Length" field counts the total bytes including the Base Header, the Service Path Header and the Context Headers. NSH further defines two MetaData formats ("MD Types") for Context Headers: 0x1 (Mandatory) and 0x2 (Optional Variable Length). The "Next Protocol" field indicates the protocol type of the original packet.

Service Path Header contains two parts: Service Path Identifier and Service Index. Service Path Identifier defines a service path, and all participating nodes must use this identifier for Service Function Path (SFP) selection. Service Index provides the current location within the SFP. Service Index must be modified by service functions after performing required services.

Context headers are used to carry opaque metadata and customized variable length information. When MD Type is set as 0x1, four mandatory context headers, 4-byte each, are added after service path header. If MD Type is set as 0x2, optional variable length context headers are added after service path header instead of mandatory context headers. It must be of an integer number of 4 bytes and follows the TLV (Type, Length, Value) format as shown in Fig. 5.3. "TLV Class" defines the scope of the "Type" field that is used to identify a specific vendor or specific standards body allocated types. "C" bit corresponds to the flag bit in the Base Header. "Type" indicates the specific type of information being carried. The exact value of a given type is provided in "Variable Metadata".

5.2.3 Implementation of SFC

Fig. 5.4 shows the overview of SFC framework as well as the packet processing in service overlay. The main components are explained as follows:

Ingress(Classifier)/Egress: When a flow arrives at the boundary of service domain, the ingress switch first checks its headers. Normally, 5-tuple (IP source/destination, protocol type, source/destination port) in header space is used to decide which service should be applied. If application-layer service differentiation is required, the payload should also be checked. Then all the packets from this flow are encapsulated in UDP and NSH. A Service Path ID is assigned to indicate the unique service path. Corresponding to ingress, egress switch is responsible for decapsulating NSH and ends the service chain.

Service switch: Service switch is in charge of steering service-specific traffic to given network functions in the correct order according to NSH. This demands service switch to

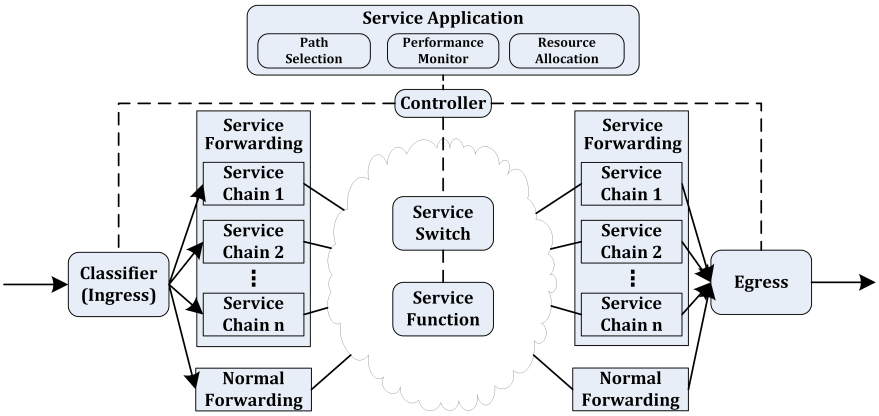


Fig. 5.4 Framework of Service Function Chaining

be aware of NSH and to modify NSH when needed. Service switch can be implemented by modifying OpenFlow-enabled software switch. The OpenFlow controller also should be modified to be capable of NSH related operations. Service switch can modify Service Index value to indicate which service function has already been applied in a chain. If necessary, service switch can also modify the value of Service Path ID. Besides service overlay forwarding, service switch also supports normal packet forwarding.

Service function: Service functions are running as applications inside containers/VMs which are directly connected to service switches. So service switch acts as a proxy to decide which service functions should be applied to the traffic. Instead of placing on physical topology, this form of virtual service functions can be inserted into or moved from service path dynamically and conveniently. It is further assumed that all service functions should also be aware of NSH and process the packets according to its original header instead of NSH outer header.

Service application: Service application is running above the controller to fulfill service chaining enforcement. It contains three main modules: Service Path Selection, Performance Monitor and Resource Allocation. Service Path Selection module is used to determine the service path for specific service function chain. We assume that all the topology information of switches and service functions are registered in the controller. Thus the controller is able to calculate the shortest service path for service chain and install forwarding rules on service switches. Performance Monitor module is able to collect statistics of specific flow from switches in order to monitor real-time running status and calculate available bandwidth. Resource Allocation module further uses gathered information to optimize the resource allocation among multiple service nodes according to various operation requirements, e.g., maximizing the overall performance, satisfying specific QoS, etc.

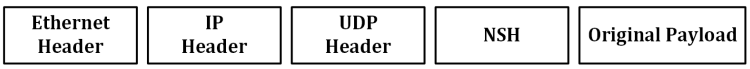


Fig. 5.5 Network protocol stack in NSH

In order to implement SFC, we modify OFsoftswitch and Ryu controller to support NSH. We follow IETF draft ("UDP Transport for Network Service Header") [94] to encapsulate NSH and original packet in UDP without requiring additional headers as overhead[1]. The network protocol stack is shown in Fig. 5.5, which is similar to VXLAN. In NSH, for simplicity, no optional Context Header is attached after Service Path Header. The source port used in the UDP overlay is set as 60001. The checksum in UDP header is set to zero for performance consideration. In OFsoftswitch, the flow table is extended to support encapsulation/decapsulation of NSH as well as matching/modifying NSH fields. Hence, the modified OFsoftswitch can be used as ingress/egress switch and service switch. Furthermore, OFsoftswitch can also act as proof-of-concept service functions, e.g., stateless Firewall, simple NAT, gateway, etc. For control plane, Ryu is selected and modified to support NSH related operations. So Path Selection module is able to install NSH-related rules on service switch. The Performance Monitor module is implemented by standard OpenFlow control messages for flow statistics. An external API based on CGroup and CPUFreq is also provided to achieve fine-grained CPU resource control. For more details about the implementation of SFC, please refer to Appendix A.

5.2.4 Resource allocation on SFC

The deployment of SFC based on software switch and virtual network function is challenging. First, both software switch and service function are required to bind to a large amount of computation resources. For instance, in order to realize 10 Gbps line rate forwarding speed for 64B packet on single port, one CPU core running at 3 GHz is needed [107], not to mention offering equivalent performance for complex network functions like Deep Packet Inspection (DPI) or web proxy. Second, data center and cloud are usually virtualized to support multi-tenant processing and operating. There should be isolated software switches or virtual services deployed for each tenant. It is common that multiple service instances coexist on the same physical hardware and share limited resources, which shifts performance issue to a more critical level. Moreover, certain VMs running intensive computation applications may also occupy considerable CPU resource. Third, different from server virtualization

[1] According to this design, the total encapsulation overhead is: $14(MAC)+20(IP)+8(UDP)+8(NSH)=50(bytes)$. In order to prevent fragmentation of UDP overlay transport, it is necessary to reconfigure the MTU value or the size of original packets to accommodate this encapsulation overhead.

where each VM usually runs as a standalone part with negligible inter-traffic with other VMs, the performance of SFC depends on the coordination of all the service functions along the service path. When multiple software switches and service functions coexist on the same server, CPU resource becomes the bottleneck which limits the overall performance. Instead of over provisioning of computation resources, we aim to orchestrate multiple services and switches within limited resources to achieve optimized total throughput.

Our idea is similar to NaaS [71] that involves the optimization of resource allocations by considering network and computing resources as a unified whole. By default on multi-core operating system, the processes are distributed among multiple cores dynamically and randomly. For SFC, the operating system is not aware of how many CPU resources should be assigned to each function or switch for given service chain. All of them are treated equally without priority or QoS guarantee. Hence, a customized and fine-grained control mechanism on CPU resource is necessary for improving SFC performance.

To this end, we design a simple scenario to demonstrate the effectiveness of resource allocation in SFC. The given scenario is described as follows: the network service domain is constituted of 3 intermediate service switches (S1, S2, S3) connected in a line. S1 is ingress switch while S3 is egress switch. 2 service functions (F1, F2) are needed in service path following the order as first F1 then F2, and both functions can be attached to any switch. For simplicity, F1 and F2 are also set up as service switch which simply sends out all incoming traffic, and no real service function executes inside. We further assume that these 5 switches (S1, S2, S3, F1, F2) are sharing 3 CPU cores (C1, C2, C3).

Moreover, Section 3.3.6 implies that the number of ports and the traffic pattern on each port impact the performance of OFsoftswitch. We further conclude that the network topology has influence on OFsoftswitch performance, since it defines the connection graph and port configuration. In SFC scenario, different connections between service functions and service switches result in different service topologies. This inspires us to combine the topology design with resource allocation coordinately for optimized throughput in scenarios like SFC where the network topology can be customized. We formulate this optimization problem as follows: 1) according to given service chain, we first pick out representative topologies among all possible ones; 2) in each selected topology, we then applied different resource allocation mechanisms among multiple service nodes (switches and service functions) based on their needs on CPU resource; 3) we choose the combination of topology and resource allocation mechanism that provides best performance.

Fig. 5.6 enumerates 2 representative topologies. Since the service functions are also set up as same as switches, other topologies are either redundant or isomorphism to these 2 topologies. The service path is listed under each topology. The table in Fig. 5.7 indicates

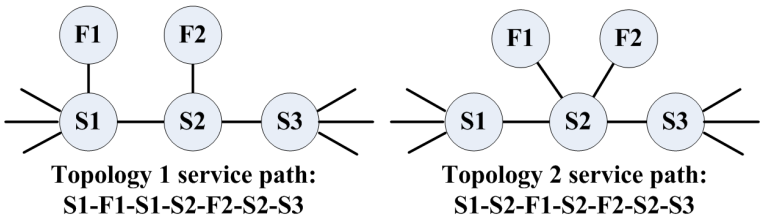


Fig. 5.6 Service Function Chaining topologies.

CPU Allocation					
Allocation	S1	S2	S3	F1	F2
Method 1	C1	C2	C3	C1	C2
Method 2	C1	C2	C3	C1	C3
Method 3	C1	C2	C3	C2	C3
Default	Deployed randomly by OS				

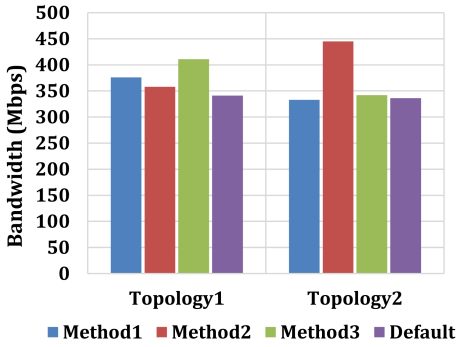


Fig. 5.7 Simulation on resource allocation.

how CPU cores are assigned to each software switch. 3 customized CPU allocation methods are listed in detail to compare with default scheduling method by operating system (OS). When enabling customized allocation, each switch process can only run on a single core. We apply these allocation methods to each topology and measure the maximum available bandwidth. From the results in Fig. 5.7, default OS scheduling shows slight performance difference on two topologies, which is 341 and 336 Mbps respectively. In both topologies, each customized CPU allocation provides equivalent or better performance than default OS scheduling. In Topology 1, Method 3 achieves best performance as 411Mbps while Method 2 achieves best performance as 445Mbps in Topology 2. Hence the best coordination between topology and resource allocation is “Topology 2 + Method 2”, which achieves over 30% performance improvement than default scheduling by OS.

In OS default scheduling, each software switch is not attached to specific cores, thus its running core is frequently changed. The context switch introduces additional overhead, especially when each CPU core has been already fully occupied by numerous SoftIRQs. In Topology 2, clearly S2 requires more CPU than other switches, since it directly connects with 2 service functions. Hence it is reasonable for S2 to occupy a single core. In Method 2, the workloads on each core are distributed evenly, and each core is almost fully occupied. That is why Method 2 achieves best overall performance. In Topology 1, both S1 and S2 are connected with one function, so both of them need more CPU resource than others. In

this case, the workloads can not be evenly distributed, and part of CPU is still idle. Thus the performance improvement is not as significant as in Topology 2.

There is still room for further improvement in resource allocation. First, in this case, we aim to provide an optimized bandwidth for SFC in a best-effort basis. However, in practice, the bandwidth usage in cloud is used for pricing. So it is necessary to provide a fixed and stable bandwidth for customers. This can be achieved by controlling CPU in a fine granularity. Second, we iterate all possible CPU allocation methods manually and statically to find out the best solution, which is not practical and costly in time. Hence, an automated CPU provision runtime is needed to perform resource allocation intelligently. The runtime should be to adjust the CPU affinity as well as CPU resource for a given task dynamically and efficiently. Third, more complex scenarios should be taken into consideration for further validating the effectiveness of resource allocation.

5.3 Automated fine-grained provision

In cloud environment, the pay-as-you-go model has been widely adopted to provide Infrastructure-as-a-Service (IaaS). Similarly, NaaS is a business model for delivering network services virtually in cloud on a pay-per-use basis. And SFC can be treated as one of typical and important applications of NaaS. SFC requires users to explicitly specify the total amount of resource to reserve. However, the users can only coarsely estimate the resource needed due to ignorance on the performance of service providers' platform. Hence, users are usually prone to oversubscribe the resource from service provider, which leads to inflated cost. From the perspective of service provider, overprovision lowers the utilization of underlying physical resources. The emergence of SDN and NFV enables SFC to be deployed in a fully virtualized manner. This brings great flexibility in network management, especially in fine-grained resource allocation. Based on these facts, we aim to present a resource allocation runtime for SFC that can satisfy the user-defined Service-Level Agreement (SLA) as well as leverage the utilization of underlying hardware.

5.3.1 Case study

In order to provide motivations and contexts for automated fine-grained provision, we present a case study for demonstration. Fig. 5.8 shows a simple SFC scenario where a Firewall (FW) is first configured to block specific hosts from accessing Internet and then a NAT is used to translate private IP addresses to public ones. Two service switches (S1 and S2) are used as ingress and egress of service domain respectively. FW connects to S1, while NAT

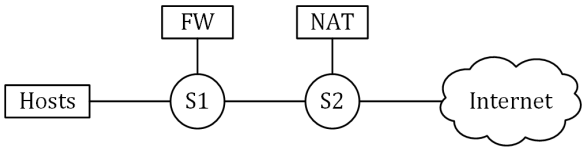


Fig. 5.8 Case study for SFC

Table 5.1 CPU bandwidth required for each service node (1500B packet)

Service node	S1	S2	FW	NAT
CPU BW (%)	22	20	11	12

connects to S2. Both FW and NAT are proof-of-concept service functions that are modified by OFsoftswitch. According to user-defined SLA, the input bandwidth between hosts and S1 is required to be 100Mbps.

According to the SLA that defines the service chain and input bandwidth, service provide needs to decide the CPU bandwidth assigned to each service node (FW, NAT, S1, S2) to guarantee the 100Mbps input bandwidth. In existing solutions, static CPU bandwidth allocation is adopted to provide 100Mbps. The result of required CPU bandwidth for each node is listed in Table 5.1. According to Section 3.3.1, due to the interference of background services, the real-time bandwidth provided by static allocation is not a constant value. It is sometimes higher than 100Mbps and sometimes lower. The 100Mbps bandwidth achieved only represents an average value of real-time bandwidth over a period of 495s[2].

In order to quantify the deviation of real-time bandwidth, the Mean Absolute Percentage Error (MAPE) is used as a metric. MAPE [19] is defined as a measure of prediction accuracy of a forecasting method in statistics. It usually expresses accuracy as a percentage. MAPE is formulated by equation:

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|$$

where A_t is the actual value at time t and F_t is the forecast value at t. In our SFC scenario, A_t represents the real-time bandwidth while F_t represents the input bandwidth defined in SLA. The total time period is 495s, and we sample real-time bandwidth per second. Hence, the MAPE value in our case is calculated to be 11%. In order to minimize MAPE, fine-grained and dynamic provisions are needed. This is also helpful to build an accurate pricing model as well as reduce OPEX.

In former case, only 1500-byte packets are used to generate traffic from the hosts. According to previous study in Section 3.3.3, the pps (packets per second) throughput in

[2]Section 3.3.1 indicates that the period of our platform is 495s. To set testing period as 495s is helpful to acquire more accuracy result.

Table 5.2 CPU bandwidth required with 512B packet

Service node	S1	S2	FW	NAT
CPU BW (%)	62	57	29	33

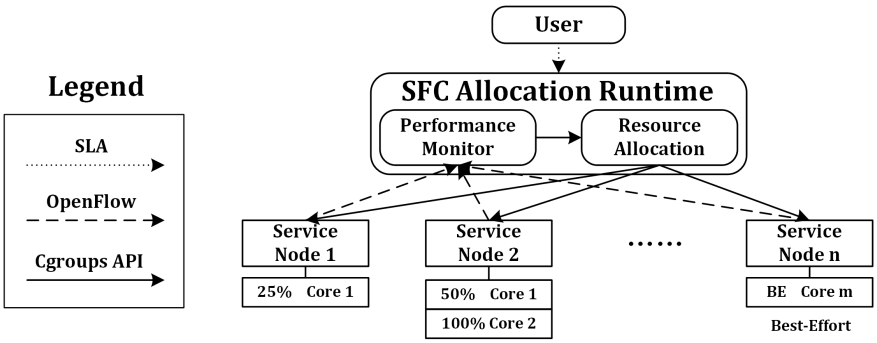


Fig. 5.9 Runtime framework

OFsoftswitch is quite stable regardless of packet size. So if the hosts generate only 512-byte packets, in order to still provide 100Mbps, more CPU bandwidth is required for each service node. The result of required CPU bandwidth for 512-byte packets is shown in Table 5.2. We can see that all CPU bandwidths are approximately increased by 3 times. Moreover, in Fig. 5.8, the actual traffic volume received by S2 and NAT depends on the packets allowed by FW. The result in former case is obtained based on the fact that all the traffic are approved to access Internet. However, if part of hosts or traffic are blocked by FW, the required CPU bandwidth for S2 and NAT will decrease according to the percentage of traffic blocked. Considering the impact of traffic characteristics, an automated runtime should be combined with fine-grained provision to allocate resources dynamically and adaptively.

5.3.2 Runtime Framework

As discussed above, a SFC resource allocation runtime is needed to support dynamic allocations based on SLA requirement and traffic characteristics instead of static methods. It provides the ability for service provider to bridge SLA requirement and the actual resource needed dynamically in order to reduce OPEX or maximize the utilization of underlying infrastructure. In SFC scenario, the resource to be allocated is CPU bandwidth. The runtime is deployed above SDN controller as an application.

Fig. 5.9 depicts the overview of SFC runtime. First, the user defines the SLA, i.e., input bandwidth of a predefined service chain. We assume that the topology of service chain is fixed. Second, the runtime determines the resource needed for each service node to satisfy SLA requirement, then the CPU bandwidth is assigned to each node by CGroups API.

Third, the runtime periodically monitors real-time bandwidth through standard OpenFlow API or other external APIs. OpenFlow provides internal counters to record the number of packets that match specific flow table or flow entry, which can be used to calculate real-time bandwidth. The error between SLA and real-time bandwidth is further used to adjust the assigned CPU bandwidth. Since this is a closed-loop system that provides feedback of the actual state, it is supposed to provide a stable bandwidth with minimized oscillation.

5.3.3 Best-effort based SLA

Besides the SLA that defines the input bandwidth of specific SFC, there is a different type of SLA that subscribes a fixed amount of physical/virtual resources. It requires service provider to implement specific SFC base on subscribed resources in a best-effort manner. The demo in Section 5.2.4 belongs to this type of SLA. In that case, we manage to maximize the throughput of specific SFC by setting CPU affinity for each service nodes. The runtime proposed should support both two types of SLA.

The implementation of resource allocation in best-effort basis is quite different from providing fixed bandwidth. Since multiple service nodes are competing CPU resource, the main idea is to distribute service nodes on different cores by setting CPU affinity for each node in order to minimize the interference among them as well as maximize the CPU utilization. It is also helpful to avoid frequent context switches. However, the default operating system scheduling does not perform well for this purpose. In order to find out the best allocation, we first need to formulate this scheduling problem. We assume that there are n service nodes running coordinately to provide a service chain. B_i represents the weight value of required CPU bandwidth for service node i. We further assume that there are $m(m \leq n)$ available

Algorithm 1: Greedy algorithm

Input: $Array(B) = [B_1, B_2, \ldots, B_n]$, $Array(C) = [C_1, C_2, \ldots, C_m]$ where C_j is an empty array

Output: $Array(C)$

$Array(S) = [S_1, S_2, \ldots, S_m]$ and S_j=0;
while $Array(B)$ *is not empty* **do**
    Select B_k where $B_k = max(B)$;
    l = index of S_l where $S_l = min(S)$;
    Remove B_k from $Array(B)$ and add it to C_l;
    $S_l = S_l + B_k$;
end
return $Array(C)$;

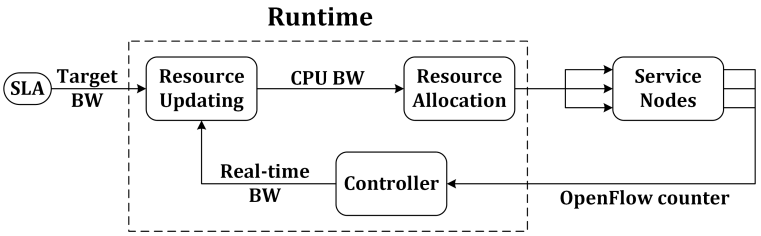


Fig. 5.10 Runtime control schema

CPU cores and C_j represents core j. Each service node should be assigned to a specific CPU core[3]. C_j is further defined as a set, $B_i \in C_j$ means B_i is running on C_j. The goal is to find one allocation that distributes multiple service nodes (B_i) among multiple cores (C_j) most evenly, which can achieve maximum CPU utilization. This is a NP-hard problem, the time complexity of a brute force search is $O(n^m)$. For efficiency, we propose a simple greedy algorithm which time complexity is $O(n * \log n)$. The evaluations of best-effort resource allocation and proposed greedy algorithm are further provided in Section 5.4.1.

5.3.4 Feedback control

Compared to best-effort based service, it is more complex to provide a fixed bandwidth automatically and dynamically. Since traditional static allocation is no longer suitable, we introduce a feedback loop system to enhance dynamic allocation. Fig. 5.10 illustrates the overall architecture of control schema. The runtime periodically compares the real-time bandwidth against the target one defined by SLA and updates the CPU bandwidth assigned to service node iteratively. Each update process can be divided into following steps:
(1) Since the runtime is deployed above SDN controller, it can use standard OpenFlow API. The service nodes (service switches and service functions) in our model are all modified from OFsoftswitch, so they are compatible with OpenFlow. OpenFlow provides counters to record the number of packets processed by specific flow table or flow entry. By periodically gathering this information, the real-time bandwidth can be calculated.
(2) Resource Updating module compares the real-time bandwidth with SLA bandwidth. For instance, the bandwidth defined by SLA is represented as B_{SLA}, while B_t represents the real-time bandwidth measured at time t. The CPU bandwidth assigned to achieve B_t is defined as C_t. By default, all previous real-time bandwidth and CPU bandwidth information are stored. The new CPU bandwidth C_{t+1} for time $(t+1)$ should be determined based on collected information. Theoretically, the throughput of software switch is proportional to CPU bandwidth, which implies this is a linear system. Hence, according to classical control

[3]The service node in our case is running in single thread, so it can be only attached to single core.

theory [54], we adopt a P (Proportional) controller to determine C_{t+1}. This is mathematically expressed as: $C_{t+1} = P * (B_{SLA} - B_t) + C_t$ where P is the proportional gain. P is a tuning parameter which is used to adjust the control loop for desired control response. P is usually decided empirically based on specific platform and control requirement. Moreover, we further modify the basic P controller model to better fit our scenario. We find that CGroups may act inaccurately sometimes and result in some deviations on real-time bandwidth during a short period of time less than 1s. For this kind of deviation, it is not necessary to reallocate CPU bandwidth, because this is a false positive alarm. Hence we add a threshold on bandwidth deviation as D_{bw}. Only if $|B_{SLA} - B_t| > D_{bw}$, the reallocation is triggered. We also replace B_t with $\bar{B}_t$ that represents the average value of previous observed real-time bandwidth during a fixed time window. The window size is defined as w, so $\bar{B}_t = (\sum_{t-w+1}^{t} B_i)/w$. This time window is a buffer to check whether a real bandwidth change happens. Correspondingly, after one reallocation is triggered, its impact on $\bar{B}_t$ also takes at least w time to be fully observed. So there should be a cooldown time after reallocation. During cooldown time, no reallocation can be triggered. The cooldown time cd is equal to w by default.

(3) Once the reallocation is triggered, Resource Allocation module then applies the new CPU bandwidth to service node by CGroups-based API. We assume that no resource scarcity occurs during resource allocation process, the remaining CPU resource is always available. The smallest unit in the allocation is 1% CPU time of single core. Although CGroups supports finer control granularity, the results do not perform well in practical use. Hence the runtime avoids the use of granularity smaller than 1% CPU time for accuracy.

The runtime monitors the real-time bandwidth and updates the resource model every second. We choose this empirical value to yield precise resource allocation and responsiveness of the system. Because higher updating frequency impacts the accuracy of CPU resource allocation performed by CGroups, while lower frequency leads to slow response to the variation of real-time bandwidth.

5.4 Evaluation

In this part, two different types of SLA in SFC, namely “Best-effort basis” and “Fixed bandwidth” are evaluated separately. The testbed is the same one as described in Section 3.2.1 and Table 3.2. More specifically for CPU configuration, we disable Hyper-Threading, Enhanced Intel SpeedStep and Turbo Boost Technologies in order to avoid the interference of external factors and enhance the accuracy of the control schema. In the evaluation, we manually set CPU frequency at 3.6GHz instead of using its default adaptive policy.

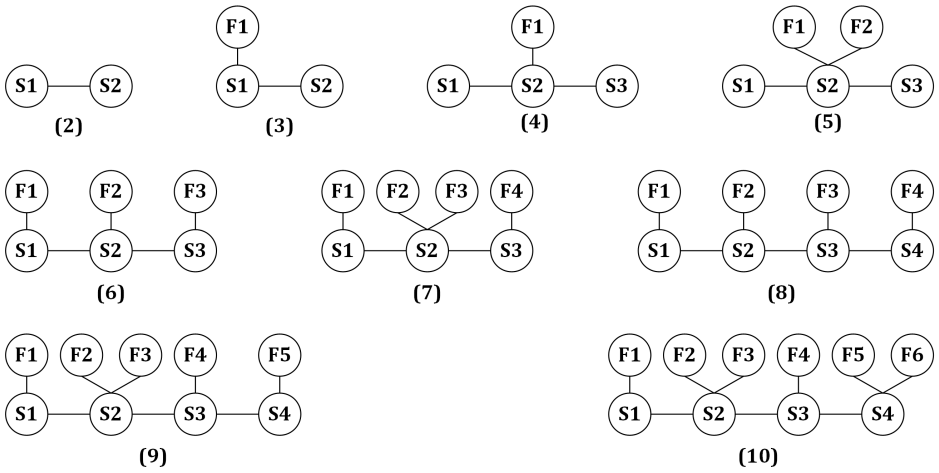


Fig. 5.11 Various topologies for best-effort basis SFC

5.4.1 Best-effort allocation

In Section 5.2.4, we have already demonstrated the effectiveness of resource allocation in performance improvement on a topology with 5 nodes and 3 CPU cores. In order to acquire one more general conclusion, we include more testing topologies with different number of nodes that are shown in Fig. 5.11. The number of nodes in each topology increases from 2 to 10. Switches (S) are always connected in a line, while service functions (F) are attached to switches. Each switch connects at most 2 service functions. All the service functions are configured as same as in Section 5.2.4. In each topology, different numbers of CPU cores (2/3/4 cores) are provided for resource allocation. The goal of this evaluation is two-fold: to quantify the improvement achieved by resource allocation and to validate the effectiveness of proposed greedy algorithm.

We start the evaluation by using default OS scheduling and obtain its throughput. Then we record the CPU usage for each service node, which is then used as the weight value B_i defined in Section 5.3.3. Next, according to the number of cores provided, the greedy algorithm is used to decide the allocation. Moreover, we also find out the best allocation by brute force search, which is used as a reference. Finally, the throughputs by applying above allocations are compared with the result by default OS scheduling to calculate the percentage of improvement. The result is shown in Fig. 5.12. We find that the allocation decided by the greedy (or called "greedy allocation") algorithm is always in agreement with the best allocation, so we only show the result by applying greedy allocation in Fig. 5.12 for simplification. There is only one exceptional case when 7 nodes are sharing 3 cores. The best allocation can achieve 20% improvement which is marked in dash line, while the improvement by greedy allocation is a bit smaller at around 15%. The percentage of

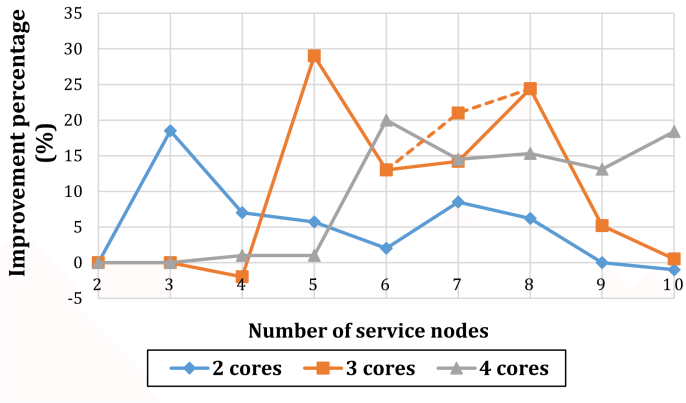
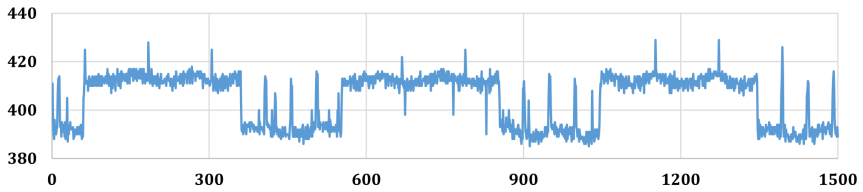


Fig. 5.12 Improvement achieved with resource allocation in best-effort basis

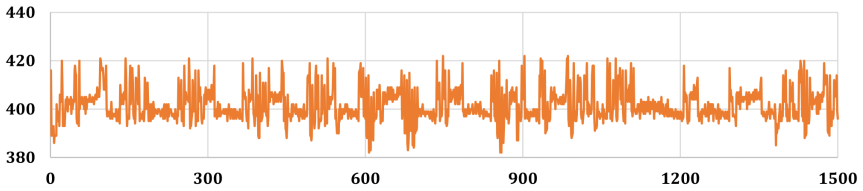
improvement achieved is various based on different numbers of service nodes and cores. By observing the overall trend of improvement, the greedy allocation always performs better than or equally to default OS scheduling. The only performance degradation occurs when 4 nodes are sharing 3 cores. Since the degradation is only 2% that can be ignored. When the number of service nodes (N_s) is smaller than the number of cores (N_c), no improvement is shown, because each node can occupy the resource of a full single core in both OS scheduling and greedy allocation cases. When N_s is significantly larger than N_c, the improvement is also negligible. Due to the large number of nodes, the intensive CPU contention and mutual interference on limited number of cores are unavoidable no matter what kind of allocation is applied. Most of significant improvements that are over 15% usually happen when $N_c < N_s < 3 * N_c$. Based on above results, our greedy allocation is proved to be more effective compared to default OS scheduling.

5.4.2 Runtime dynamic allocation

The evaluation on runtime dynamic allocation aims to validate its capability in providing stable bandwidth defined by SLA. We assume that SLA only defines the specific service chain and its input bandwidth. In order to satisfy the requirements, the runtime just fixes the input bandwidth at ingress switch according to SLA and guarantees that the CPU resources assigned to other service nodes are sufficient for actual needs and will not act as bottleneck in service chain. We further assume that no resource scarcity occurs on service nodes. Therefore, the impact of service nodes other than ingress switch can be ignored, and the runtime only needs to focus on the resource control of ingress switch. Based on above analysis, the evaluated service chain can be extremely simplified as topology (2) shown in Fig. 5.11. Only two switches are presented in the topology. S1 is ingress while S2 is egress.



(a) Static allocation



(b) Runtime allocation

Fig. 5.13 Static allocation vs Runtime allocation

The service path is S1-S2. The input bandwidth in SLA is defined as 400Mbps. Iperf is used to measure the real-time TCP bandwidth controlled by runtime dynamic allocation.

We first compare the stability of real-time bandwidth between static allocation and runtime allocation. The packet size is fixed as 1500 bytes. Fig. 5.13 shows the real-time bandwidth provided by two allocations over a period of 1500s. As shown in Fig. 5.13a, although the average bandwidth provided by static allocation is close to 400Mbps, the real-time bandwidth is either 10Mbps higher or 10Mbps lower than target value. As described in Section 5.3.1, MAPE is used to quantify the deviation between real-time bandwidth and target bandwidth. In static allocation, MAPE is 2.8%. In runtime allocation, the parameters defined in Section 5.3.4 are set as follows: we empirically select proportional gain (P) as 1/5; the threshold of bandwidth deviation (D_{bw}) that triggers reallocation is set as 6Mbps; Both time window (w) and cooldown time (cd) are set as 5s. As shown in Fig. 5.13b, most of the time the real-time bandwidth can be fixed at around 400Mbps. MAPE in runtime allocation is 1.2%, which is only half of the value in static allocation. This means that the runtime is capable of providing a more accurate real-time bandwidth. However, in runtime allocation, we can see more intensive fluctuations than in static allocation. This is the side effect of CPU bandwidth reallocation. Every time CGroups reallocates the CPU bandwidth, there exists a short period of time that the CPU bandwidth can not be assigned correctly as defined. Even so, our runtime is still proved to outperform static allocation with smaller MAPE.

We further check the dynamic behavior of runtime allocation when traffic characteristics change. All the parameters are configured as same as former case except D_{bw}. We enlarge D_{bw} as 20Mbps. Because the real-time bandwidth fluctuates more intensively in this evaluation, a larger D_{bw} is helpful to avoid frequent and unnecessary reallocations. In the beginning,

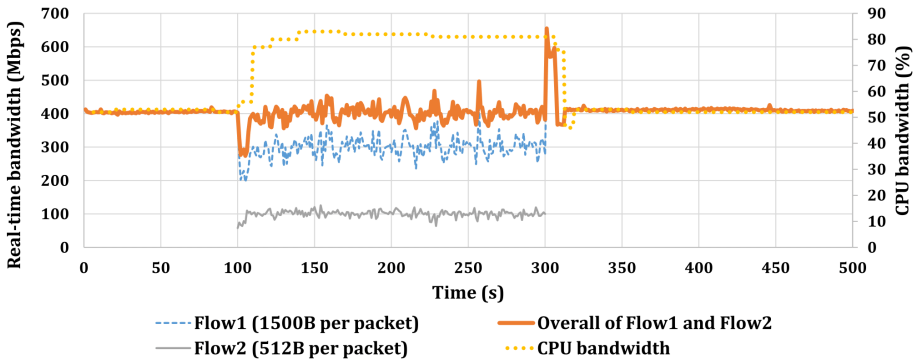


Fig. 5.14 Dynamic behavior of runtime allocation

the traffic contains only 1500-byte packets that belong to Flow1, and the runtime provides stable real-time bandwidth. At 100th second, Flow2 that contains only 512-byte packets joins the traffic. The dynamic reactions of real-time bandwidth and CPU bandwidth allocation are shown in Fig. 5.14. As explained in Section 5.3.1, smaller packets reduce the overall bandwidth, since pps (packets per second) is a constant value with fixed CPU bandwidth. Consequently, the overall bandwidth decreases from 400Mbps to 300Mbps. By monitoring real-time bandwidth, the runtime can detect this change and react accordingly. Due to the existence of time window (w), it takes at least 5s for the runtime to be aware of bandwidth change. And then CPU bandwidth reallocation is triggered. The runtime executes reallocation twice to achieve 400Mbps again, so the total reaction time is about 10s. Next, Flow2 quits at 300th second. Contrary to the behavior when it joins, the overall bandwidth first significantly increases and then returns to 400Mbps under the control of runtime. The runtime executes reallocation 3 times and the total reaction time is 15s.

Above evaluation result demonstrates that the runtime is capable of providing stable real-time bandwidth even when there are variations on traffic characteristics. The feedback system always introduces a control latency (or called reaction time). In our case, the reaction time is 10-15s. This time depends on the values of parameters P and w. With larger P and smaller w, the reaction time of the system can be reduced, but the inaccuracy and fluctuation of the system is also enlarged at the same time. Moreover, due to the side effect of CGroups, it is not practical to reallocate CPU bandwidth frequently. Hence we select these values in consideration of balancing between the reaction time and the stability of the system.

5.5 Summary

In this chapter, we extend our vision from the performance of individual software switch to global optimization among multiple switches. For this purpose, we first explore the

general packet processing in Linux in order to understand the background of resource contention among multiple software switches. Then we introduce two Linux kernel tools, namely "CGroups" and "CPUFreq", which are used to achieve fine-grained CPU control. Next, Service Function Chaining (SFC) is presented as one typical scenario where multiple software switches/virtual network functions coexist are sharing the same underlying physical resources. SFC combines the advantage of both SDN and NFV to extremely simplify the deployment of composite service that are constructed from one or more L4-L7 network functions. Since SFC is a new approach for service delivery and operation, the work on its definition and implementation is still active and ongoing under IETF. We follow IETF drafts to implement a prototype of SFC by using Network Service Header (NSH). The prototype not only supports service chaining control by extending standard OpenFlow API, but also provides performance monitor and fine-grained CPU control for service node. Based on the prototype, we propose a simple SFC scenario where 5 service nodes are sharing 3 CPU cores. We demonstrate that the overall SFC performance in a best-effort basis can be improved over 30% by applying suitable resource allocation compared to default OS scheduling. This result proves the necessity and effectiveness of resource allocation in SFC, which also further inspires us to develop an automate runtime for fine-grained resource allocation.

The pay-as-you-go model has been widely adopted to provide Infrastructure-as-a-Service (IaaS) and Network-as-a-Service (NaaS). SFC is one of typical applications of NaaS to deliver network services on a pay-per-user basis. It is important for service providers to meet the requirements of user-defined SLA. Meanwhile, users are usually prone to oversubscribe the resource from service provider, which lowers the utilization of underlying physical resource. For above reasons, fine-grained resource allocation is necessary and helpful in SFC deployment. We find that static resource allocation can not perform well in complex scenario where the characteristics of traffic can change. In order to overcome the shortcoming of static allocation, an automated runtime is proposed to support dynamic resource allocation.

The runtime is implemented as an application that is running on SDN controller. It supports two different types of SLA, namely "Best-effort basis" and "Fixed bandwidth". In "Best-effort basis" SLA, the user subscribes a fixed amount of physical/virtual resources and requires service provider to implement the specific service chain base on subscribed resources in a best-effort manner. We formulate this allocation problem and it turns out to be NP-hard. For efficiency, we propose a greedy algorithm to solve it. Various testing topologies with different number of service nodes and CPU cores are used to evaluate the effectiveness of resource allocation. The result shows that the greedy algorithm can always correctly find out the best allocation except only one case. And runtime allocation always performs better than or equally to OS default scheduling. Especially when the number of service nodes

(N_s) and the number of cores (N_c) satisfy the following relationship as $N_c < N_s < 3 * N_c$, the average improvement is about 15%.

"Fixed bandwidth" SLA aims at providing a fixed real-time bandwidth automatically and dynamically, which is quite different from best-effort basis. Since traditional static allocation is no longer suitable, we introduce a feedback loop system to enhance dynamic allocation. According to classical control theory, a simple P (Proportional) controller is used to perform resource allocation. We further modify the basic P controller model to better fit our scenarios by introducing new parameters (bandwidth deviation threshold, buffer window and cooldown time). We empirically choose the parameter values in consideration of balancing between precise resource allocation and responsiveness of the system. In the evaluation, we first compare the stability of real-time bandwidth between static allocation and runtime allocation when the traffic characteristics remain unchanged. The MAPE value in runtime allocation is only 1.2% compared to 2.8% in static allocation, which proves that runtime dynamic allocation outperforms static allocation in providing accurate real-time bandwidth. Then we check the dynamic behavior of runtime allocation when traffic characteristics change. The result demonstrates that runtime allocation is capable of providing stable real-time bandwidth even when there are variations on traffic characteristics. The reaction time is 10-15s.

The runtime is proposed as a general framework that can be extended to support different types of resources and various service-level objectives. In our SFC scenario, CPU bandwidth is the unique bottleneck resource. In order to control other bottleneck resources, it is needed to implement related API for monitoring and allocating the given resource. For instance, the resource control on memory can be easily supported by using memory subsystem in CGroups. Moreover, in addition to the SLAs we have evaluated, other service objectives like energy saving or maximization of infrastructure utilization can be achieved as long as related control algorithms or policies are added to the runtime.

Chapter 6

Conclusion & Future Work

Due to the growing trend of "Softwarization", virtualization is becoming the dominating technology in data center and cloud environment. Software switch is indispensable to the success of network virtualization, especially when combined with emerging SDN and NFV. For this purpose, this thesis exactly targets the deployment issues of software switch in SDN-enabled network virtualization environment, which is also helpful to indicate its deployment in practical use.

6.1 Thesis summary

SDN and NFV are emerging networking technology that are both designed to facilitate the deployment and management of network by open standard APIs instead of traditional vendor-specific manner. Software switch is the powerful tool to implement SDN and NFV related services. However, the combination of software switch and NFV/SDN is still far from well studied. Based on these facts, we first explore the current studies in Chapter 2. And then we summarize the main problems into 4 categories according to the hierarchy of SDN: data plane, interaction between OpenFlow and data plane, control network and control plane. Next, we indicate the missing subjects in previous works such as in-band control and fine-grained resource allocation. Finally, in order to improve previous studies and fill the gap, we carry out our study in following 3 directions: software switch evaluation, controller evaluation and fine-grained resource control.

In Chapter 3, due to the concern on software switch performance, a systematic performance evaluation is carried out. We select two representatives of OpenFlow-enabled software switch, namely OpenvSwitch and OFsoftswitch. Since the implementations of two switches are quite different in various aspects, the comparison between them is helpful to lead to a more general conclusion. Based on the analysis of OpenFlow packet processing, we select

several main performance factors and evaluate their impacts. For completeness, other factors related to operating system and underlying hardware are also investigated. More specially, we discuss the efficiency of veth pair as virtual interface and how it is balancing the performance and security features when combined with software switch. We draw conclusions of both differences and similarities between two selected switches. These conclusions point out the suitable scenario for each software switch. For instance, OpenvSwitch is capable of providing high-speed packet processing, while OFsoftswitch is suitable for the scenarios where fine-grained resource control is required. Moreover, we demonstrate the design principle behind in-band control. Compared with out-of-band control, in-band control introduces larger latency in handling new flows due to long control path. This problem can be solved by proactive rule installation scheme.

Since the controller is the cornerstone to the success of SDN architecture, a fair and fully reproducible performance evaluation of controller is provided in Chapter 4. Beyond a simple benchmark, we not only examine general system wide settings like Python interpreter and Hyper-Threading to evaluate their impacts, but also design specifically crafted scenarios for further measurements. Based on the outcome of the evaluation, we find that performance is no longer the only dimension in choosing controllers. The usability, reliability and security are equally important. Since SDN is evolving in a fast pace and results in growing complexity of the controller, it is hard for independent developer to follow, instead a large group of communities or companies is needed. We also see the necessity of distributed controller in large scale scenarios because of the performance limitation of centralized controller. Due to the importance of synchronization traffic in implementing the functionality of distributed controller, a preliminary study is provided to investigate its traffic characteristic. The result shows that the conflict among various control messages lowers the efficiency of synchronization. This study also implies that the control message should be associated with a priority when processed by the controller and switches, which further requires combining the design of east/westbound API and southbound API.

Chapter 5 investigates the resource contention and resource allocation among multiple software switches. We first explore the general packet processing in Linux to understand the background of resource contention. Then we build a prototype of Service Function Chaining architecture where multiple switches or service functions are sharing the same underlying hardware. Next our case study demonstrates that the overall performance of the SFC prototype can be significantly improved by suitable resource allocation. This inspires us to propose an automated runtime for fine-grained resource allocation. The runtime supports two different types of SLA for SFC, namely “Best-effort basis” and “Fixed bandwidth”. For “Best-effort basis” SLA, we formulate the problem and propose a greedy algorithm to

solve it. The result shows that our allocation always performs better than or equally to OS default scheduling. For "Fixed bandwidth" SLA, we introduce classical control theory to enhance dynamic resource allocation. The runtime is proved to be capable of providing stable real-time bandwidth even when the traffic characteristics change. The runtime is presented as a general framework that can be extended to support different types of resources and various service-level objectives.

6.2 Publication

We report here the list of already published papers and submitted ones, which are related to this manuscript. Each paper is also associated with its related chapters.

1. Y. Zhao, L. Iannone, M. Riguidel. Measuring In-Band Overhead of SDN Controller Clusters. In Proceedings of CoNEXT Student Workshop '15, Heidelberg, Germany, Dec, 2015. (Chapter 4)
2. Y. Zhao, L. Iannone, M. Riguidel. On the Performance of SDN controllers: A Reality Check. In Proceedings of 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), San Francisco, CA, USA, Nov, 2015. (Chapter 4)
3. Y. Zhao, L. Iannone, M. Riguidel. Software Switch Performance Factors in Network Virtualization Environment. In Proceedings of 2014 IEEE 22nd International Conference on Network Protocols (ICNP), Raleigh, NC, USA, Oct, 2014. (Chapter 3 and Chapter 5)

6.3 Discussion & Future work

In this thesis, we attempt to tackle a number of critical problems in software switch deployment in network virtualization environment, which raises a couple of open problems. Therefore, our study can be extend and enhanced in following directions.

The evaluations should be carried out on an enhanced testbed. Our testbed is equipped with single 4-core CPU, which only meets the minimum requirement for multi-thread testing. This directly constrains the scale of our testing scenario. When evaluating the greedy algorithm, the maximum number of service nodes are limited within 10. In order to comprehensively evaluate the effectiveness of our resource allocation scheme, more CPU cores are needed. Similar issue also exists in distributed controller evaluation. Since distributed controllers usually bind to a large amount of resources for parallel computation,

our testbed can not be comparable with official testing platform that at least two 10-core CPUs are provided [25]. With a powerful testbed, we can further replace the emulated benchmark tools with real traffic tests. For example, we can use numerous software switches instead of Cbench to evaluate the controllers, which is more close to the reality. Many global research networks for academic purpose such as Ofelia [24] or PlanetLab [38] could be taken into consideration to build such a testbed for large scale deployment.

Distributed controllers designed for large scale networks are gaining momentum, with number of projects already existing. The evaluation of distributed controller is more complex than centralized controller. Because the performance of distributed controller not only relies on the underlying computation resources, but also depends on the efficiency of synchronization as well as the placement of controller nodes. However, the specifications of east/westbound APIs are still under discussion, and their implementations are various in different products. Our investigation on synchronization traffic is a start point to evaluate the effectiveness of existing implementations, which is helpful to indicate the potential bottleneck and could be used as a feedback to modify the existing system. More efforts and investigations could follow this direction to establish an ecosystem for effectively developing and testing east/westbound APIs.

As mentioned in Chapter 5, the runtime is rather a general framework that can be enhanced to support various types of resources and service-level objectives. In order to control the CPU resource for service node more precisely, we could build a resource–performance model for each service function. For this purpose, both offline and online profiling can be applied. NFV-VITAL [65] is a general offline framework for performance characterization of various Virtual Network Functions (VNFs) in a real private cloud deployment. Bartolini *et al.* propose AutoPro [57] as a runtime that adopts control theory to build and update the resource-performance model through online estimation. Besides resource–performance model, traffic predication is also helpful to enhance resource allocation. Because the resource required for each function directly depends on traffic characteristics, and accurate traffic predication allows the system to react in advance to reduce the response time.

References

[1] 6WINDGate and SDN.
http://www.6wind.com/software-defined-networking/6windgate-sdn/.

[2] About Nox. http://www.noxrepo.org/nox/about-nox/.

[3] About Pox. http://www.noxrepo.org/pox/about-pox/.

[4] Bridge Linux Foundation.
http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.

[5] Cbench. https://github.com/andi-bigswitch/oflops/tree/master/cbench.

[6] CGroups - The Linux Kernel Archives.
https://www.kernel.org/doc/Documentation/cgroups.

[7] Cisco Nexus 1000V Switch for VMware vSphere.
http://www.cisco.com/c/en/us/products/switches/nexus-1000v-switch-vmware-vsphere/index.html.

[8] Docker. https://www.docker.com/.

[9] European Telecommunications Standards Institute (ETSI). http://www.etsi.org/.

[10] FloodLight OpenFlow Controller. http://www.projectfloodlight.org/floodlight/.

[11] HP ProCurve. https://en.wikipedia.org/wiki/ProCurve.

[12] Huge Pages - The Linux Kernel Archives.
https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[13] Indigo virtual switch. http://www.projectfloodlight.org/indigo-virtual-switch/.

[14] Intel Data Plane Development Kit. http://www.dpdk.org.

[15] Kernel Virtual Machine (KVM). http://www.linux-kvm.org/page/Main_Page.

[16] LINC OpenFlow software switch. https://github.com/FlowForwarding/LINC-Switch.

[17] Linux Foundation. NAPI.
http://www.linuxfoundation.org/collaborate/workgroups/networking/napi.

[18] Linux Kernel Contributors. Packet_mmap.
https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt.

[19] Mean absolute percentage error.
https://en.wikipedia.org/wiki/Mean_absolute_percentage_error.

[20] Mininet An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/.

[21] Netmap - a framework for fast packet I/O. https://github.com/luigirizzo/netmap.

[22] ntop: High Performance Network Monitoring Solution. http://www.ntop.org/.

[23] ntop. PF_RING ZC (Zero Copy).
http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[24] OFELIA: A European project providing OpenFlow-based experimental facilities.
http://www.fp7-ofelia.eu/.

[25] ONOS Blackbird Performance Evaluation.
https://wiki.onosproject.org/display/ONOS11/Blackbird+Performance+Evaluation.

[26] Open Networking Operating System. http://onosproject.org/.

[27] Open vSwitch. http://openvswitch.org/.

[28] OpenDayLight Platform. https://www.opendaylight.org/.

[29] OpenFlow 1.3 Software Switch. https://github.com/CPqD/ofsoftswitch13.

[30] OpenFlow Reference Switch. http://yuba.stanford.edu/git/gitweb.cgi?p=openflow.git.

[31] OpenFlow specification v1.0.
http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[32] OpenFlowClick. http://archive.openflow.org/wk/index.php/OpenFlowClick.

[33] OpenOnload. http://www.openonload.org/.

[34] OpenStack. https://www.openstack.org/.

[35] OpenVZ Virtuozzo Containers Wiki. https://openvz.org/Main_Page.

[36] OPNFV Linux Foundation. https://www.opnfv.org.

[37] Pantou: OpenFlow 1.0 for OpenWRT.
http://archive.openflow.org/wk/index.php/Pantou_:_OpenFlow_1.0_for_OpenWRT.

[38] PlanetLab: An open platform for developing, deploying and accessing planetary-scale services. https://www.planet-lab.org/.

[39] QEMU Open source processor emulator. http://wiki.qemu.org/Main_Page.

[40] Ryu SDN Framework. http://osrg.github.io/ryu/.

[41] Scaling in the Linux Networking Stack.
https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[42] Softswitch Wikipedia. https://en.wikipedia.org/wiki/Softswitch.

[43] The Leading Open Source In-Memory Data Grid. http://hazelcast.org/.

[44] The NetBee Library. http://www.nbee.org/doku.php.

[45] The Rise of Soft Switching Part IV: Comments on the Hardware Supply Chain. http://networkheresy.com/2011/09/29/the-rise-of-soft-switching-part-iv-comments-on-the-hardware-supply-chain/.

[46] Traffic Control HOWTO. http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[47] What's LXC? https://linuxcontainers.org/lxc/introduction/.

[48] Network Functions Virtualization – Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf, Oct. 2012.

[49] The software defined data center. http://www.emc.com/collateral/presentations/7-vmware-afternoon-sessions.pdf, 2013.

[50] Network Functions Virtualization – Use Cases. http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf, June 2014.

[51] V. Abidi, Y. Chen, and M. Hamilton. An Automation Framework and Methodology for Measuring OVS Performance. http://openvswitch.org/support/ovscon2015/17/1050-abidi.pptx.

[52] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A. Moore, and P. Owezarski. OSNT: open source network tester. *Network, IEEE*, 28(5):6–12, September 2014.

[53] P. Arlos and M. Fiedler. A Method to Estimate the Timestamp Accuracy of Measurement Hardware and Software Tools. In *Proceedings of the 8th International Conference on Passive and Active Network Measurement*, PAM'07, pages 197–206, Berlin, Heidelberg, 2007. Springer-Verlag.

[54] K. J. Åström and B. Wittenmark. *Adaptive control*. Courier Corporation, 2013.

[55] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou. An analytical model for software defined networking: A network calculus-based approach. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, pages 1397–1402, Dec 2013.

[56] S. Azodolmolky, P. Wieder, and R. Yahyapour. Performance Evaluation of a Scalable Software-Defined Networking Deployment. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 68–74, Oct 2013.

[57] D. B. Bartolini, F. Sironi, D. Sciuto, and M. D. Santambrogio. Automated Fine-Grained CPU Provisioning for Virtual Machines. *ACM Trans. Archit. Code Optim.*, 11(3):27:1–27:25, July 2014.

[58] A. Beifus, D. Raumer, P. Emmerich, T. Runge, F. Wohlfart, B. Wolfinger, and G. Carle. A study of networking software induced latency. In *Networked Systems (NetSys), 2015 International Conference and Workshops on*, pages 1–8, March 2015.

[59] V. Bhuvaneswaran, B. Anton, and et al. Benchmarking Methodology for OpenFlow SDN Controller Performance. draft-bhuvan-bmwg-of-controller-benchmarking-00, Sept. 2014.

[60] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. OpenFlow Switching: Data Plane Performance. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5, May 2010.

[61] K. Blaiech, S. Hamadi, A. Mseddi, and O. Cherkaoui. Data plane acceleration for virtual switching in data centers: NP-based approach. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 108–113, Oct 2014.

[62] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[63] F. A. Botelho, F. M. V. Ramos, D. Kreutz, and A. N. Bessani. On the Feasibility of a Consistent and Fault-Tolerant Data Store for SDNs. In *Proceedings of the 2013 Second European Workshop on Software Defined Networks*, EWSDN '13, pages 38–43, Washington, DC, USA, 2013. IEEE Computer Society.

[64] E. A. Brewer. Towards Robust Distributed Systems. In *Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing*, (PODC '00), New York, USA, 2000.

[65] L. Cao, P. Sharma, S. Fahmy, and V. Saxena. NFV-VITAL: A framework for characterizing the performance of virtual network functions. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 93–99, Nov 2015.

[66] M. Casado. OpenStack and Network Virtualization. http://blogs.vmware.com/tribal knowledge/2013/04/openstack-and-network-virtualization.html, April 2013.

[67] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: a retrospective on evolving SDN. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 85–90. ACM, 2012.

[68] M. Challa. OpenvSwitch Performance measurements & analysis. http://openvswitch.org/support/ovscon2014/18/1600-ovs_perf.pptx.

[69] Y.-H. Chang, T.-Y. Chung, and Y.-M. Chen. Spring-based resource management for end-to-end services in next-generation networks. In *Ubiquitous and Future Networks (ICUFN), 2015 Seventh International Conference on*, pages 701–706, July 2015.

[70] N. M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[71] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NaaS: Network-as-a-service in the Cloud. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[72] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[73] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Performance characteristics of virtual switching. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 120–125, Oct 2014.

[74] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Assessing Soft-and Hardware Bottlenecks in PC-based Packet Forwarding Systems. *ICN 2015*, page 90, 2015.

[75] D. Erickson. The Beacon Openflow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[76] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. *SIGPLAN Not.*, 46(9):279–291, Sept. 2011.

[77] P. Garg, P. Quinn, R. Manur, J. Guichard, S. Kumar, A. Chauhan, B. McConnell, M. Smith, C. Wright, U. Elzur, J. M. Halpern, W. Henderickx, T. Nadeau, S. Majee, D. T. Melman, K. Glavin, and P. Agarwal. Network Service Header. Internet-Draft draft-quinn-sfc-nsh-07, Internet Engineering Task Force, Feb. 2015.

[78] J. M. Halpern and C. Pignataro. Service Function Chaining (SFC) Architecture. IETF RFC 7665, Nov. 2015.

[79] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, 2015.

[80] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[81] 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 '15, pages 25:1–25:6, New York, NY, USA, 2015. ACM.

[82] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

[83] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity Switch Models for Software-defined Network Emulation. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 43–48, New York, NY, USA, 2013. ACM.

[84] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries. A Flexible OpenFlow-Controller Benchmark. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 48–53, Oct 2012.

[85] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and P. Tran-Gia. OFCProbe: A platform-independent tool for OpenFlow controller analysis. In *Communications and Electronics (ICCE), 2014 IEEE Fifth International Conference on*, pages 182–187, July 2014.

[86] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and performance evaluation of an OpenFlow architecture. In *Teletraffic Congress (ITC), 2011 23rd International*, pages 1–7, Sept 2011.

[87] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association.

[88] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. Research Directions in Network Service Chaining. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pages 1–7. IEEE, 2013.

[89] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[90] X. Kong, Z. Wang, X. Shi, X. Yin, and D. Li. Performance evaluation of software-defined networking with real-life ISP traffic. In *Computers and Communications (ISCC), 2013 IEEE Symposium on*, pages 000541–000547, July 2013.

[91] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, Apr. 2014. USENIX Association.

[92] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[93] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76, 2015.

[94] S. Kumar, L. Kreeger, S. Majee, W. Haeffner, R. Manur, and D. T. Melman. UDP Transport For Network Service Header. Internet-Draft draft-kumar-sfc-nsh-udp-transport-01, Internet Engineering Task Force, Nov. 2015. Work in Progress.

[95] S. Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma. Service Function Chaining Use Cases In Data Centers. Internet-Draft draft-ietf-sfc-dc-use-cases-03, Internet Engineering Task Force, July 2015.

[96] M. Kuzniar, P. Peresini, and D. Kostic. What you need to know about SDN control and data planes. Technical report, 2014.

[97] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 199–212, New York, NY, USA, 2014. ACM.

[98] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 249–262, New York, NY, USA, 2014. ACM.

[99] Microsoft. Hyper-V Windows Server. https://technet.microsoft.com/en-us/windowsserver/dd448604.aspx, 2014.

[100] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. *SIGPLAN Not.*, 47(1):217–230, Jan. 2012.

[101] S. Muramatsu, R. Kawashima, S. Saito, and H. Matsuo. VSE: Virtual Switch Extension for Adaptive CPU Core Assignment in Softirq. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 923–928, Dec 2014.

[102] S. Muramatsu, R. Kawashima, S. Saito, H. Matsuo, H. Nakayama, and T. Hayashi. A Software Approach of Controlling the CPU Resource Assignment in Network Virtualization. *IEICE Transactions on Communications*, 98(11):2171–2179, 2015.

[103] E. Ng. Maestro: A System for Scalable OpenFlow Control. Technical report, TSEN Maestro-Technical Report TR10-08, Rice University, 2010.

[104] M. Obadia, M. Bouet, J.-L. Rougier, and L. Iannone. A greedy approach for minimizing SDN control overhead. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–5, April 2015.

[105] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[106] K. Phemius and M. Bouet. OpenFlow: Why latency does matter. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 680–683, May 2013.

[107] G. Pongrácz, L. Molnár, and Z. L. Kis. Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK. In *Proceedings of the 2013 Second European Workshop on Software Defined Networks*, EWSDN '13, pages 62–67, Washington, DC, USA, 2013. IEEE Computer Society.

[108] G. Pongrácz, L. Molnár, Z. L. Kis, and Z. Turányi. Cheap Silicon: A Myth or Reality? Picking the Right Data Plane Hardware for Software Defined Networking. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 103–108, New York, NY, USA, 2013. ACM.

[109] Y. Pu, Y. Deng, and A. Nakao. Cloud Rack: Enhanced virtual topology migration approach with Open vSwitch. In *Information Networking (ICOIN), 2011 International Conference on*, pages 160–164, Jan 2011.

[110] P. Quinn, J. Guichard, R. Fernando, Surendra, P. Agarwal, R. Manur, A. Chauhan, N. Leymann, M. Boucadair, C. Jacquenet, M. Smith, N. Yadav, T. Nadeau, K. Gray, B. McConnell, and Kevin. Network Service Chaining Problem Statement, draft-quinn-nsc-problem-statement-03. https://datatracker.ietf.org/doc/draft-quinn-nsc-problem-statement/, 2013.

[111] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *INFOCOM, 2012 Proceedings IEEE*, pages 2471–2479, March 2012.

[112] C. Rotsos, G. Antichi, M. Bruyere, P. Owezarski, and A. Moore. OFLOPS-Turbo: Testing the Next-Generation OpenFlow switch. In *European Workshop on Software Defined Networks (EWSDN)*, page 2p, 2014.

[113] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement*, PAM'12, pages 85–95, Berlin, Heidelberg, 2012. Springer-Verlag.

[114] L. Rupprecht. Exploiting In-network Processing for Big Data Management. In *Proceedings of the 2013 SIGMOD/PODS Ph.D. Symposium*, SIGMOD'13 PhD Symposium, pages 1–6, New York, NY, USA, 2013. ACM.

[115] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From Policies to Pipelines. *SIGPLAN Not.*, 49(9):11–24, Aug. 2014.

[116] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced Study of SDN/OpenFlow Controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '13, pages 1:1–1:6, New York, NY, USA, 2013. ACM.

[117] S. Shin, Y. Song, and et al. Rosemary: A Robust, Secure, and High-performance Network Operating System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 78–89, New York, NY, USA, 2014. ACM.

[118] H. Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[119] V. Tanyingyong, M. Hidell, and P. Sjodin. Improving performance in a combined router/server. In *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*, pages 52–58, June 2012.

[120] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On Controller Performance in Software-defined Networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[121] C.-C. Tu, P.-W. Wang, and T.-c. Chiueh. In-Band Control for an Ethernet-Based Software-Defined Network. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, pages 1:1–1:11, New York, NY, USA, 2014. ACM.

[122] D. Turull, M. Hidell, and P. Sjodin. Performance evaluation of openflow controllers for network virtualization. In *High Performance Switching and Routing (HPSR), 2014 IEEE 15th International Conference on*, pages 50–56, July 2014.

[123] VMware. vSphere ESX and ESXi Info Center. http://www.vmware.com/products/esxi-and-esx/overview.html, 2014.

[124] Z. Wang, T. Tsou, J. Huang, X. Shi, and X. Yin. Analysis of Comparisons between OpenFlow and ForCES. *ForCES, IETF*, Sept. 2012.

[125] A. Weissberger. VMware's Network Virtualization Poses Huge Threat to Data Center Switch Fabric Vendors. http://viodi.com/2013/05/06/vmwares-network-virtualization-poses-huge-threat-to-data-center-switch-fabric-vendors/, May 2013.

[126] L. Yao, P. Hong, and W. Zhou. Evaluating the controller capacity in software defined networking. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–6, Aug 2014.

[127] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 351–362, New York, NY, USA, 2010. ACM.

[128] S. Zhang, S. Malik, S. Narain, and L. Vanbever. In-Band Update for Network Routing Policy Migration. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 356–361, Oct 2014.

Appendix A

Source Code

Most of our implementations are modified from previous open source projects that are published on GitHub. So we also upload related codes to GitHub in my personal repositories as https://github.com/zeinsteinz. It is convenient to download and explore the codes. For instance, in order to find out all the modifications on OFsoftswitch, we can find all commitment history in a diff mode by using following link https://github.com/zeinsteinz/ofsoftswitch13/commits/master. A brief introduction of each repository is provided as follows.

A.1 OFsoftswitch

https://github.com/zeinsteinz/ofsoftswitch13

For OFsoftswitch, we have made two main modifications. One is to improve its performance in packet parsing by more efficient codes to replace original Netbee library. The other one is to extend it to support NSH for SFC implementation.

A.2 Ryu

https://github.com/zeinsteinz/ryu

Similar to OFsoftswitch, Ryu is also modified to support NSH related operations such as field matching/modification as well as encapsulation/decapsulation.

A.3 Mininet

https://github.com/zeinsteinz/mininet

Aiming at in-band control simulation, Mininet is modified to allow OpenvSwitch running in private network namespace. It is also extended to support Docker for distributed controller measurement. In addition, several test scripts and customized CLI commands are provided as well.

Appendix B

Résumé en Français

1. Introduction

Une tendance croissante des “ Softwarization ” qui se passe dans presque tous les domaines de la technologie de la communication et Information. La virtualisation des serveurs est maintenant de premier plan dans la majorité des centres de données. De même dans la zone du réseau, tant par Software Defined Network (SDN) et Network Function Virtualization (NFV) sont des expressions différentes de Software Defined Infrastructure (SDI) dans une tendance de transformation globale de “ La Softwarization du Réseau”. Switch logiciel est exactement l’outil approprié et puissant pour soutenir SDN et NFV. En ce qui concerne les défis et opportunités dans la softwarization du réseau, cette thèse vise à étudier le déploiement de switch logiciel dans un environnement de la virtualisation du réseau SDN-enabled. Dans notre étude, nous nous concentrons d’abord sur l’évaluation des performances des switches logiciels OpenFlow-enabled. Ensuite, une vérification de la réalité sur la performance du contrôleur est effectuée en raison de son importance dans le cadre de SDN. Enfin, nous proposons un cadre d’exécution automatique pour fournir une allocation dynamique et adaptative des ressources pour les switches logiciels. Dans cette section, une introduction du switch logiciel et ses scénarios de déploiement prometteurs sont fournis.

Différemment de la définition traditionnelle dans les télécommunications [42] qui fait référence au dispositif central ou d’un logiciel utilisé pour connecter des appels téléphoniques avec différentes autres lignes téléphoniques, nous définissons “ switch logiciel ” en tant que switch virtuel construit sur le système informatique d’usage général pour mettre en œuvre le transfert de paquets ainsi que d’autres fonctionnalités réseau. Plus précisément, le système informatique général dans cette thèse est limitée à l’architecture x86, et d’autres plates-formes comme ARM ou FPGA ne sont pas considérés. Les implémentations typiques

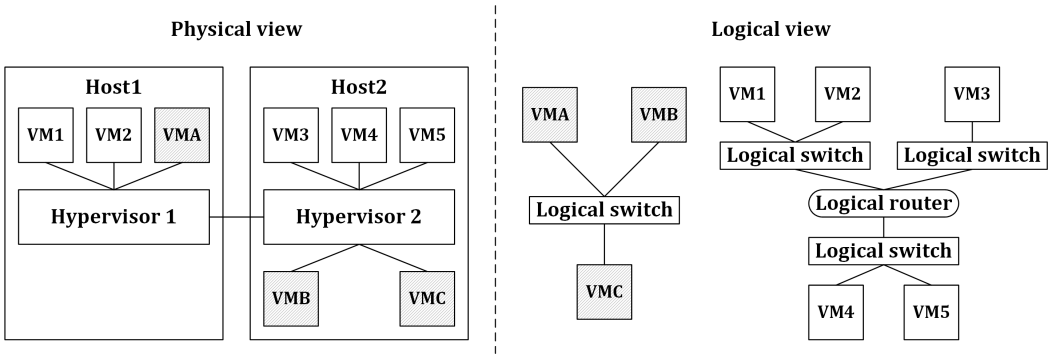


Fig. B.1 La virtualisation du réseau

de switch logiciel incluent Linux Bridge [4], Click routeur [89], switches virtuels VMware ESX [123] ou Microsoft Hyper-V [99], etc

La virtualisation du réseau permet la coexistence de plusieurs réseaux virtuels qui partagent les mêmes infrastructures physiques sous-jacents [70]. Elle sépare la fonctionnalité traditionnelle du réseau en deux rôles: fournisseur d'infrastructure qui construit les infrastructures du réseaux physiques et fournisseur de services qui crée et gère le réseau virtuel et fournit réseau de bout en bout en tant que service. Fig. B.1 montre un scénario typique de la virtualisation du réseau. VMA, VMB et VMC sont dans le même sous-réseau, tandis que VM1 à VM5 appartiennent à l'autre sous-réseau. Bien que les machines virtuelles dans le même sous-réseau sont en cours d'exécution sur les différents hôtes physiques, en utilisant la virtualisation du réseau, ils peuvent être gérés dans des réseaux logiques isolés grâce à des topologies virtuelles personnalisées. Dans la virtualisation du réseau, une superposition à base de tunnel est généralement adoptée dans les centres de données distribués pour fournir une connectivité à travers WAN. Toutes les opérations d'encapsulation / décapsulation de sont mises en œuvre que sur le bord, le switch logiciel. Différents protocoles de tunneling ont été proposés par Internet Engineering Task Force (IETF), y compris GRE, VXLAN, STT, LISP, GENEVE, etc. Ces protocoles choisissent différentes couches d'encapsulation et visent à des scénarios spécifiques. La plupart des protocoles de tunneling sont initialement pris en charge dans le noyau Linux, qui peut être facilement intégré dans le switch logiciel.

L'idée principale du Software Defined Network (SDN) est de briser l'intégration verticale en séparant le plan de contrôle des réseaux à partir du plan de données sous-jacentes. SDN adopte un plan de contrôle relativement centralisé et fournit une vue globale de l'ensemble du réseau. La virtualisation de réseau a acquis une nouvelle traction avec l'avènement de la SDN. Parce que la virtualisation de réseau vise à découpler les ressources du réseau à partir de matériel sous-jacent, SDN offre une interface standard entre les applications de commande et dispositifs de transmission sous-fifre, qui est une plate-forme naturelle pour la virtualisation

du réseau. Par exemple, Network Virtualization Platform (NVP) [91] est conçu pour les centres de données multi-locataires sur la base de SDN. NVP utilise OpenvSwitch dans tous les nœuds de transport (hyperviseurs, les nœuds de service et passerelle) pour transférer des paquets. OpenvSwitch est configurable à distance par le contrôleur NVP pour gérer les tables de débit ainsi que les tunnels de recouvrement. La virtualisation du réseau a été l'un des facteurs à l'origine de l'émergence du switch logiciel. Les switches logiciels SDN-enabled sont plus prometteuses pour fournir une grande souplesse pour permettre la virtualisation du réseau. Outre SDN et la virtualisation de réseau, le switch logiciel trouve aussi sa place dans d'autres technologies de réseau émergentes telles que la Network Function Virtualization (NFV) et le Network-as-a-Service (NaaS).

Le softwarization de réseau est progressivement et inévitablement briser l'intégration verticale classique de l'architecture de réseau spécifique au fournisseur, qui permet multitude de services qui pourraient être créés et fournis par le biais des plates-formes dynamiques et sans marge de ressources logiques, totalement découplée des infrastructures physiques sous-jacents. La transfert de paquets par un logiciel gagne la popularité comme base pour le softwarization du réseau, car elle conduit à des économies de coûts, une plus grande flexibilité et la programmabilité. Plus précisément, le switch logiciel, qui a été largement déployé et testé dans un environnement de virtualisation hôte, est un bon point de départ pour étudier l'impact de la virtualisation du réseau basé sur un logiciel associé à des technologies de pointe telles que SDN et NFV.

2. Contexte

Parce que les switches logiciels traditionnels (par exemple, Click, Linux Bridge, etc.) ont été largement déployées dans divers environnements industriels plus d'une décennie, leurs problèmes de déploiement sont déjà bien étudiés. Les technologies prometteuses, y compris SDN et la virtualisation de réseau stimuler davantage le développement de switch logiciel. Un nombre croissant de switches logiciels SDN-enabled sont attendus dans un avenir proche. Toutefois, en raison de l'immaturité, SDN soulève des questions quant à sa performance et scalabilité. Le switch qui représente plan de données dans SDN joue un rôle indispensable pour répondre à ces questions. Par conséquent, comprendre où la performance et la limitation des switch logiciel SDN-enabled est indispensable à son succès dans le déploiement.

Plusieurs switches logiciels OpenFlow-enabled sont déjà disponibles, tandis que leur utilisation diffère du prototype conceptuel à la qualité de la production. Une sélection d'implémentations open source sont introduits dans le Tableau B.1. Tous les commutateurs logiciels sont soit basés sur OpenvSwitch ou OpenFlow Reference Switch sauf LINC,

Table B.1 Les switches logiciels OpenFlow-enabled

Product	Developer	Code Base	Language	OF version	Update
OpenvSwitch	Nicira	Original	C	v1.0 & v1.3	08/2015
OF ref switch	Stanford	Original	C	v1.0	06/2011
OFsoftswitch	CPqD	OF ref switch	C&C++	v1.3	07/2015
Indigo	Big Switch	OpenvSwitch	C	v1.0	07/2015
LINC	FlowForwarding	Original	Erlang	v1.2 $\sim$ v1.4	08/2015
OpenFlowClick	Stanford	OF ref switch	C	v0.9	08/2009
Pantou	Stanford	OF ref switch	C	v1.0	08/2010

qui indique que les dessins originaux de switches logiciels OpenFlow-enabled sont rares. Presque toutes les implémentations choisir C comme langage de programmation pour garantir une haute performance. Erlang est utilisé uniquement par LINC pour fournir une grande flexibilité, mais sacrifier la performance. OpenvSwitch et Indigo sont développés par des sociétés commerciales, qui peuvent fournir des mises à jour périodiques et les nouvelles versions. Surtout pour OpenvSwitch, en raison de sa popularité et déploiement à grande échelle, son développement est si intensive pour répondre aux exigences de performance dans divers scénarios réels. OFsoftswitch et LINC qui sont des communautés universitaires peuvent également garder à jour en temps opportun et en ajoutant de nouvelles fonctionnalités progressivement. Les autres implémentations n'ont pas été mises à jour depuis longtemps. Pour version prise en charge de la spécification OpenFlow, v1.0 et v1.3 sont deux étapes importantes. Parce que OpenFlow v1.0 est la version officielle d'abord publié et OpenFlow v1.3 représente un autre grand pas en avant en introduisant plusieurs tables de flux et d'autres concepts utiles. Cependant, seulement OpenvSwitch, OFsoftswitch et LINC soutien v1.3, tandis que les autres seulement pour v1.0. Sur la base de notre comparaison sur les performances, la maintenance et version prise en charge OpenFlow, OpenvSwitch et OFsoftswitch sont deux implémentations les plus représentatives.

Cette thèse est censé mener une étude approfondie sur switch logiciel SDN-enabled. Il intègre plusieurs sujets qui ne sont étudiés séparément avant. Fig. B.2 montre un cadre simplifié de SDN pour présenter des domaines de recherche traditionnels en cours. Le switch est composé de deux éléments compacts, le module de transfert et le module de OpenFlow. Le module de transfert est en charge de la transmission du trafic de données, alors que le module de OpenFlow communique avec le contrôleur via le réseau de contrôle et manipule le module de transfert selon les instructions du contrôleur. De même, dans le contrôleur, le module de OpenFlow est utilisé pour interagir avec le switch. L'application intégrée sur le module de OpenFlow implémente la logique de contrôle réel. Les travaux connexes peuvent

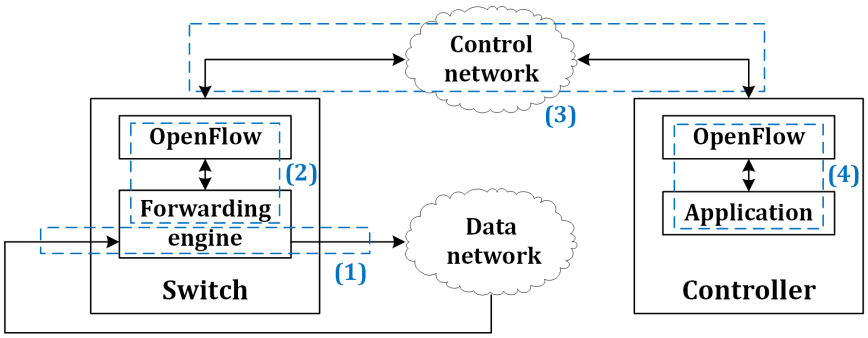


Fig. B.2 L'aperçu des études antérieures

être principalement divisés en quatre catégories en fonction des zones représentées sur la Fig. B.2 (marquée par des blocs dans la ligne du tableau de bord).

(1) Dans le switch (plan de données), la performance du module de transfert de paquets est primordiale pour le succès de l'ensemble du cadre, en particulier dans switch logiciel qui a problèmes de performances par rapport à un matériel dédié. Outre l'évaluation des performances du module de transfert, il y a un certain nombre de travaux visant à l'amélioration des performances en appliquant nouveau cadre I/O. Ils exploitent les progrès potentiels des deux perspectives, matérielles et logicielles.

(2) L'interaction entre le module de OpenFlow et le module de transfert est cruciale pour l'efficacité du plan de données. Car il est important non seulement le temps d'attente de l'installation d'écoulement, mais également de la capacité dans le traitement de nouvelles demandes de débit simultanément. Dans les cas généraux, OpenvSwitch effectue de manière plus efficace et plus prévisible que la plupart des switch matériels spécifiques au fournisseur sur le circuit de contrôle de switch [83].

(3) Dans le cadre SDN, chaque switch doit établir et maintenir une connexion TCP avec son contrôleur. Il existe deux catégories sur la façon dont cette connexion traverse le réseau, à savoir "out-of-band contrôle" et "in-band contrôle". Out-of-band contrôle utilise un réseau de contrôle dédié qui est complètement différent du réseau de données contrôlée par les switches, tandis que in-band contrôle partage même réseau de commutation de données. En conséquence, in-band contrôle introduit une complexité et une latence supplémentaire dans la transmission des messages de contrôle. L'impact du in-band contrôle a besoin de complément d'enquête.

(4) Différemment de réseau traditionnel, SDN adopte un plan de contrôle centralisé appelé contrôleur. Le contrôleur est une fondation critique dans SDN paradigme, car elle fournit un soutien essentiel de toute logique de réseau, conformément aux

politiques définies par les opérateurs de réseau. Par conséquent, le contrôleur détermine directement la scalabilité et la disponibilité de l'ensemble du système. Dans ce contexte, il est nécessaire de comprendre les implémentations de contrôleurs et d'identifier leurs goulots d'étranglement, ce qui en retour fournit des conseils sur la sélection de régulateur approprié pour un scénario donné.

Comme il a déjà été mentionné, le déploiement de switch logiciel SDN-enabled couvre plusieurs aspects des switches et contrôleurs côtés. En outre, en raison du découplage du plan de données et plan de contrôle, SDN introduit de nouveaux défis en plus des problèmes de réseau traditionnel. Les études précédentes ont étudié différents sujets, y compris l'évaluation de la performance, la conception du cadre et de la modélisation analytique. Cependant, il y a encore beaucoup d'aspects qui sont négligés.

Comme cela est expliqué à la Fig. B.2, il existe deux systèmes de se connecter des switches avec le contrôleur dans SDN, à savoir "out-of-band contrôle" et "in-band contrôle". Out-of-band contrôle nécessite un réseau de contrôle indépendant qui est séparé du réseau de données normal. Out-of-band contrôle peut simplifier la mise en œuvre du switch et maintenir le trafic de contrôle loin de l'interférence du trafic de données. Tandis que in-band contrôle est plus facile à déployer sans nécessiter réseau de contrôle supplémentaire. Out-of-band contrôle et in-band contrôle ont leurs propres avantages et sont adaptés à différents scénarios. In-band contrôle est rarement mentionné précédemment, et son impact est encore inconnu à l'exploitation du réseau. Dans le scénario de in-band contrôle, il est difficile de gérer le changement d'état du réseau, comme la défaillance des dispositifs et le reroutage, car toute erreur de configuration ou mauvaise opération peut entraîner d'une panne réseau. Bien qu'un nombre limité d'études commencent à étudier l'impact du in-band contrôle, il est encore loin de bien étudié. Même les principes fondamentaux de conception de in-band contrôle ne sont pas clairement précisées, pour ne pas mentionner d'autres conclusions générales pour in-band contrôle. Afin de mieux comprendre in-band contrôle, il est raisonnable de commencer à partir d'un switch logiciel spécifique comme OpenvSwitch qui prend en charge à la fois in-band contrôle et out-of-band contrôle. La comparaison directe entre les deux système de contrôle est utile d'observer les différences et d'identifier les problèmes potentiels.

En comparaison avec le matériel matériel dédié, switch logiciel apporte plus de flexibilité dans le déploiement et la mise à niveau. Mais switch logiciel conduit également à des performances instables. Plutôt que de se concentrer sur la performance individuelle de switch, davantage d'efforts doivent être faits dans l'orchestration et la coordination entre les plusieurs switches/fonctions du réseau. Dans NFV et NaaS, plusieurs switches logiciels et les fonctions du réseau virtuel coopèrent généralement avec l'autre pour mettre en œuvre une série de services ou de tâches. Un exemple typique est Service Function Chaining (SFC)

qui dirige le trafic spécifique au service pour parcourir les fonctions de service du réseau (ou middleboxes) dans l'ordre donné. Comme tous les switches logiciels et fonctions virtuelles partagent les mêmes ressources physiques sous-jacents, la compétition et l'interférence sur la ressource ne peuvent être évités. l'allocation des ressources à grains fins est également utile dans ces scénarios pour minimiser l'interférence ainsi que de maximiser la performance globale.

La combinaison de SDN et switch logiciel est si prometteur qu'il a gagné beaucoup d'attention à la fois académique et de l'industrie. Mais SDN et switch logiciel ne sont pas des panacées et ont leurs propres limites. Les problèmes de déploiement et de performance de switch logiciel ne sont pas encore systématiquement étudiés dans les études précédentes. Plusieurs limitations existantes se résument comme suit.

- Bien qu'il existe différentes implémentations de switch logiciels SDN-enabled, les évaluations de performance précédentes toutes concentrées uniquement sur OpenvSwitch. En raison de l'absence d'analyse approfondie et une comparaison complète entre les différentes implémentations, leurs conclusions ne peuvent pas fournir les caractéristiques générales du switch logiciel.
- La plupart des études précédentes étudier switch logiciel comme une partie autonome et négliger l'impact des autres composants dans le cadre SDN tels que le plan de contrôle et le réseau de contrôle, qui ne peut pas conduire à des résultats convincible pour l'usage pratique.
- En comparant avec l'amélioration de la performance du switch logiciel individuel, l'orchestration et la coordination entre les plusieurs switches logiciels sont encore plus importants, en particulier dans le réseau de virtualisation environnement où les plusieurs switches logiciels partagent la même ressource physique sous-jacente.
- Même si la performance du switch logiciel peut être grandement améliorée par de nouveaux cadres de I/O, il ne peut toujours pas être comparable avec du matériel spécifique. Par conséquent, il est plus raisonnable de déployer switch logiciel dans les scénarios appropriés sur la base de ses caractéristiques.

3. L'évaluation de la performance du switches logiciel

Afin de résoudre à la contradiction entre la popularité du switch logiciel et ses performances limitées pour le déploiement dans SDN, une évaluation de la performance est nécessaire d'identifier les principaux facteurs de performance et les goulets d'étranglement potentiels. OpenvSwitch et OFsoftswitch sont sélectionnés pour l'évaluation des performances en tant

Table B.2 OpenvSwitch VS. OFsoftswitch

Softswitch	Implementation	Specification	Thread	Flow Table Type
OpenvSwitch	Kernel space	1.0 & 1.3	Multiple	Hashtable
OFsoftswitch	User space	1.3	Single	Lineartable

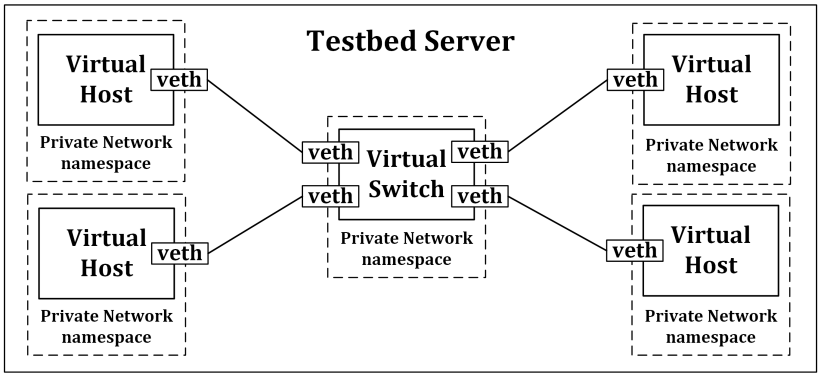


Fig. B.3 La configuration de banc de test

que deux switch logiciels OpenFlow-enabled représentatifs. Selon la brève comparaison dans le Tableau. B.2, OpenvSwitch et OFsoftswitch sont très différentes dans les divers aspects, ce qui est utile de conduire à une conclusion plus générale.

Notre banc de test est construit sur un seul serveur et tous les composants sont virtualisé. Dans le contexte de la virtualisation, les machines virtuelles/conteneurs coexistent avec des switches logiciels sur le même serveur. Ils génèrent non seulement la charge CPU importante pour le calcul, mais peuvent également générer une charge de trafic importante pour la communication mutuelle. Fig. B.3 explique comment configurer le banc de test dans un environnement entièrement virtualisé. Il y a 3 éléments nécessaires à l'évaluation des performances de switch logicielle:

- **Hôte virtuel**: Hôte virtuel est la source/destination du trafic de données. Dans l'évaluation des performances, iperf et netperf sont utilisés dans l'hôte virtuel pour générer du trafic de test. Afin d'isoler plusieurs hôtes virtuels, un mécanisme de virtualisation légère "network namespace" est utilisé à la place de la technologie traditionnelle de la machine virtuelle comme KVM [15]. Network namespaces sont des conteneurs pour les états du réseau. Ils fournissent des processus spécifiques avec la propriété exclusive des interfaces, des ports et des tables de routage. Les périphériques virtuels ou réels peuvent être ajoutés à chaque network namespace.
- **Switch virtuel**: Dans l'évaluation des performances, OpenvSwitch (OVS) et OFsoftswitch (OFS) sont utilisés comme switches virtuels.

- **Lien virtuel**: Virtual network interfaces (veth) paire est utilisé comme lien virtuel pour connecter des périphériques virtuels. Veths existe toujours par paires. Une paire de veths sont connectés comme un tuyau, à savoir, tout paquet reçu par une interface de veth sortira de l'autre interface veth. Veth peut être associée à des switches virtuels ou hôtes virtuels, comme représenté sur la Fig. B.3. Du point de vue des périphériques virtuels et des hôtes virtuels, veths sont traités comme des interfaces réseau normales.

Mininet [20] est utilisé pour établir des tests topologie ainsi que d'exécuter des scripts de test. OpenvSwitch et OFsoftswitch sont connectés au contrôleur en utilisant le out-of-band contrôle. Le loopback interface est utilisé pour la communication entre les switches et le contrôleur afin d'éliminer goulot d'étranglement potentiel. La version de la spécification OpenFlow est v1.3. Seul le trafic IPv4 est utilisé pour l'évaluation des performances. Le serveur de banc de test a un CPU Intel Xeon E5-1620 avec 4 cœurs (8 cœurs logiques lors de l'activation de Hyper-Threading), chacun fonctionnant à 3.6GHz. La configuration de la mémoire est de $4\times$2GB DDR3 1600 MHz.

Nous étudions les principaux facteurs de performance et d'évaluer leurs impacts. Fig. B.4 résume un organigramme standard pour le traitement des paquets dans le switch logiciel selon la spécification OpenFlow. Le temps théorique pour le traitement d'un paquet doit être: $T = t_{polling} + t_{I/O} + t_{rules} + t_{actions} + t_{overhead}$. Le switch logiciel utilise le mode de polling pour vérifier les données disponibles sur les ports. $t_{polling}$ représente ce coût en temps pour polling sur les ports. Si un nouveau paquet arrive au port, module I/O lit d'abord le paquet et le copie dans la mémoire. Ensuite, ce paquet est recherché dans les règles de pipeline de flux avec l'ordre de priorité jusqu'à ce qu'une règle est trouvé. t_{rules} fait référence au temps de recherche totale. Si aucune règle ne correspond au paquet, ce paquet est soumis au contrôleur pour obtenir des instructions supplémentaires. Ensuite, les actions associées dans la règle appariés sont appliquées au paquet séquentiellement. Le temps pour l'exécution des actions est représenté par $t_{actions}$. Enfin, le module I/O envoie le paquet à partir de la mémoire. Pour plus de simplicité, $t_{I/O}$ est le temps total de lecture du paquet, et envoyer le paquet. Après le traitement d'un port, le switch logiciel retourne à l'étape du polling et répète processus ci-dessus sur le port suivant. $t_{polling}$ est traitée comme une surcharge, car il existe toujours, peu importe s'il y a des paquets disponibles. Dans chaque polling ronde, il existe une autre frais généraux fixes $t_{overhead}$. Elle est due à la conception de logiciels, comme le recyclage des ressources ou l'enregistrement gestionnaire d'événements. Tous ces frais généraux sont marqués avec des lignes de la grille dans l'organigramme. Selon l'organigramme, on divise le temps de traitement total en parties indépendantes et de mesurer leurs impacts sur les performances.

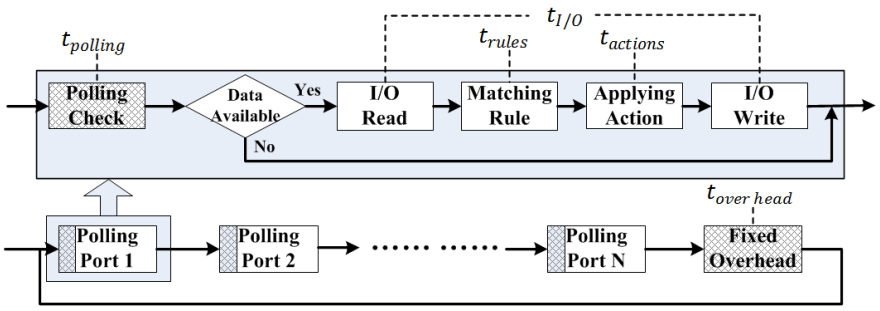


Fig. B.4 L'organigramme de traitement du paquet

Table B.3 La performance de base

Scenario	RTT (ms)	TCP (Mbps)	UDP (Mbps)
Direct veth pair	0.023	8460	12020
LinuxBridge+veth	0.037	5205	7274
OpenvSwitch+veth	0.034	7150	7780
OFsoftswitch+veth	0.095	803	1130

Nous commençons notre mesure avec la plus simple configuration, qui est utilisé comme une base de référence. Deux hôtes virtuels (conteneurs) sont connectés au même switch logiciel/Linux Brdige. A titre de comparaison, un scénario extrêmement basique où deux hôtes virtuels sont directement reliés par une paire de veth est également évaluée. D'après les résultats de la Tableau. B.3, OpenvSwitch effectue environ 25% de mieux sur TCP et 7% de mieux sur UDP que Linux Bridge. OFsoftswitch fournit encore que 803 Mbps sur TCP et 1130 Mbps sur UDP. En raison de meilleures performances et une plus grande flexibilité, OpenvSwitch remplace progressivement Linux Bridge dans la plupart des scénarios.

I/O comprend à la fois la réception et la transmission de paquets, et le principal facteur de I O est la taille des paquets. Dans cette partie, on étudie l'impact de la taille des paquets sur le fonctionnement des I/O. Nous utilisons simple flux UDP unidirectionnel pour mesurer la bande passante maximale avec différentes tailles de paquets (de 64 octets à 1500 octets). D'après les résultats, en dépit de la différence de mise en œuvre, la bande des deux OpenvSwitch et OFsoftswitch est proportionnelle à la taille du paquet. Le degré d'ajustement linéaire est supérieur à 99.5% et 99.95% respectivement. Ce résultat implique que "paquets par seconde (pps)" valeur est indépendante de la taille du paquet.

Le nombre de règles a un grand effet sur les performances du switches logiciels en raison du temps de recherche total. OFsoftswitch utilise une table linéaire pour rechercher les règles de correspondance, donc t_{rules} est proportionnelle au nombre de règles qu'il recherche. Nous obligeons chaque paquet entrant à comparer avec un nombre donné de règles avant trouver

à la bonne. Selon les résultats, lorsque le nombre de règles augmente, OFsoftswitch coûte plus de temps du CPU, ce qui réduit la bande totale. Avec 2000 règles, OFsoftswitch ne peut atteindre 200 Mbps (environ 17k pps). Nous mesurons encore le temps de traitement t_{rules} avec un nombre différent de règles à mettre en correspondance. Le temps de traitement t_{rules} est proportionnelle au nombre de règles, et le degré d'ajustement linéaire est supérieur à 99.5%. Selon cette relation linéaire, le temps pour faire correspondre seule règle est de $0.09 \sim 0.1$ µs. Le hashtable du noyau dans OpenvSwitch garantir que le temps pour faire correspondre est une valeur constante quel que soit le nombre de règles.

Chaque règle est associé à zéro ou plusieurs actions qui indiquent comment gérer les paquets correspondants. Les actions qui peuvent être appliquées sont différentes des opérations de base comme la réécriture têtes de paquets pour les complexes comme l'encapsulation des paquets (VLAN ou MPLS). Le résultat montre que la bande n'a pas de différence avec diverses actions. Dans un système de multidiffusion, un paquet doit être copié et envoyé à plus d'un port. Cette copie et transmission supplémentaire dégrade la bande total. Ls bande de OpenvSwitch diminue d'environ 40% avec 4 ports de sortie par rapport à un seul port. L'impact sur OFsoftswitch est plus petit, le résultat avec 4 ports peut encore atteindre 75% de la bande avec un port. Comme une dégradation significative des performances peut être observée, le déploiement de switch logiciel dans un système de multidiffusion doit être conçu plus attentivement.

Différent de la mise en œuvre du matériel où plusieurs ports peuvent fonctionner en parallèle, switch logiciel utilise le mode de polling pour vérifier les paquets disponibles sur les ports. Sur la Fig. B.4, outre les frais généraux fixes dans un tour de polling, la vérification des paquets disponibles sur chaque port introduit une charge supplémentaire. Afin de maximiser totale *pps*, OFsoftswitch devrait essayer d'envoyer des paquets autant que possible en un seul tour pour réduire la part des frais généraux. Différemment de OFsoftswitch qui adopte un mode de polling simple dans user-space, OpenvSwitch utilise NAPI pour gérer les paquets arrivés plus intelligemment. Par conséquent, l'impact du modèle de trafic sur la performance de OpenvSwitch peut être atténué de manière significative, tant que la ressource du noyau est toujours disponible.

Chaque switch OpenFlow doit établir et maintenir une connexion TCP à son contrôleur. Il existe deux catégories de base sur la façon dont cette connexion traverse le réseau: il est soit en utilisant un réseau dédié qui est complètement différent de celui commandé par les switches, ou il partage le même réseau des switches. Le premier cas est traité comme "Out-of-band contrôle", alors que ce dernier cas est "In-band contrôle".

Dans SDN, le mode réactif à installer des règles a introduit latence supplémentaire dans le traitement de nouveaux flux par rapport au schéma de routage traditionnel. Sans un réseau

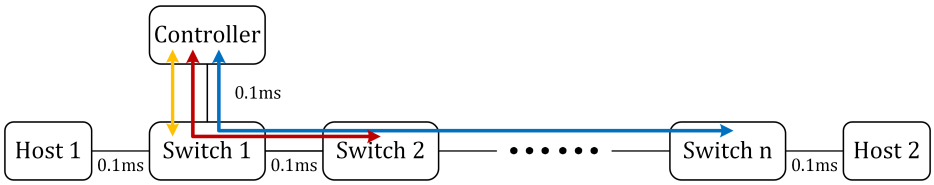


Fig. B.5 Exemple de in-band contrôle

de contrôle dédié dans le scénario de in-band contrôle, cette latence peut encore être élargie en raison de la topologie et l'état du réseau. Fig. B.5 affiche un exemple de scénario dans in-band contrôle: n switches sont linéairement connecté; deux hôte sont reliés à l'extrémité de la chaîne de switch; le contrôleur se connecte directement au switch 1. Pour switch n communiquer avec le contrôleur, les messages de contrôle doivent passer par tous les autres switches pour arriver au le contrôleur. Ce comportement résulte en grande latence grâce à la longue route de contrôle. Afin de remédier à ce problème, une solution d'installation proactif est proposé. Au lieu d'attendre les demandes de chaque switch, le contrôleur doit installer des règles sur tous les switches liés en même temps une fois qu'il reçoit la demande du premier switch. Le régime proactif peut réduire considérablement la latence à un niveau acceptable. Parce que ce système élimine non seulement l'effet cumulatif de la latence sur switch, mais installe également des règles plus efficacement en poussant de manière proactif.

Dans cette section, nous nous concentrons sur l'analyse des performances des deux switches logiciels OpenFlow, à savoir "OpenvSwitch " et "OFsoftswitch", dans un environnement entièrement virtualisé. Au cours de l'analyse, nous analysons d'abord le traitement des paquets par OpenFlow et le diviser en plusieurs facteurs principaux, puis évaluer ces facteurs sur deux switches séparément. Outre les facteurs de performance à l'intérieur de switch logiciel, nous étudions en outre les facteurs externes tels que veth paire et la fréquence du processeur. In-band contrôle est pratique pour le déploiement sans nécessiter un réseau de contrôle dédié. Toutefois, il est rarement mentionné dans les études précédentes. Mais in-band contrôle introduit plus grande latence dans le traitement de nouveaux flux à cause de longue route de contrôle. Un schéma d'installation de la règle proactif est proposée pour régler le problème de latence. Nos résultats peuvent être utilisés comme un guide pour la conception d'un réseau SDN sur la base du switches logiciels. Ils sont également utiles pour obtenir le contrôle des ressources à grains fins sur switch logiciel.

Table B.4 SDN contrôleur

Controller	Ver.	Lang.	OF	Release	Thread
Pox	0.2.0	Python	1.0	10/2013	Single
Ryu	3.19	Python	1.0∼1.4	03/2015	Single
Nox	0.9.2	C++	1.0&1.3	02/2014	Mult.
Floodlight	0.90	Java	1.0	11/2012	Mult.
Beacon	1.0.4	Java	1.0	09/2013	Mult.

4. L'évaluation des performances du contrôleur

Le contrôleur est un élément essentiel dans le paradigme SDN, car elle fournit un soutien important pour la mise en logiques de contrôle du réseau. SDN est principalement basée sur un plan de contrôle relativement centralisé dans les scénarios émergents. En raison de cette approche centralisée, les performances du contrôleur détermine directement le scalabilité de SDN dans de tels scénarios, qui devient crucial pour le succès de la SDN écosystème.

Nos objectifs d'évaluation sont: avoir une évaluation juste pour les contrôleurs, et fournir une indication dont l'un est approprié dans lequel le scénario. nous nous concentrons sur les contrôleurs centralisés, à savoir: Ryu [40], Pox [3], Nox [2], Floodlight [10], et Beacon [75]. Tableau B.4 résume les principales caractéristiques des contrôleurs sélectionnés.

Le banc de test est construit sur un serveur qui a seul processeur Intel Xeon E5-1620, avec 4 cœurs à 3.6GHz. Tous les contrôleurs ont été configurés en mode OpenFlow v1.0. Cbench [5] est utilisé comme outil d'évaluation. Pour plus de simplicité, chaque contrôleur testé uniquement exécute une application de L2 learning switch. Afin de libérer tout le potentiel de chaque contrôleur, leurs configurations ont été optimisés selon leurs sites officiels et les communautés de développement.

Nous explorons d'abord l'impact de l'interpréteur Python. Afin d'améliorer l'efficacité de l'interpréteur original CPython, un interpréteur de remplacement, PyPy, a été développé au cours des dernières années. Pour étudier l'impact de l'interpréteur pour les contrôleurs basés sur Python, deux d'entre eux ont été utilisés pour Ryu et Pox. En ce qui concerne la latence, PyPy atteint environ 3.8 fois l'amélioration pour les deux Pox et Ryu, comparativement à CPython. Encore une fois, PyPy contribue à améliorer les performances de débit de Pox et Ryu de 9.4× et 4.4× respectivement. Lorsque vous utilisez PyPy, Pox et Ryu atteindre le même débit à environ 105 milliers de réponses par seconde. Parce que PyPy améliore les performances des contrôleurs basés sur Python significative, dans le reste de l'évaluation des résultats des contrôleurs basés sur Python sont toutes effectuées en utilisant PyPy.

Nous commençons alors notre mesure avec la configuration la plus simple, qui est utilisé comme une référence. Chaque contrôleur fonctionne sur un thread unique, et le nombre de

Table B.5 La performance de base du contrôleur

Controller	Throughput (rps/ms)	Latency (ms)
Pox	105	0.0416
Ryu	106	0.037
Nox	687	0.0179
Floodlight	670	0.0222
Beacon	2302	0.0164

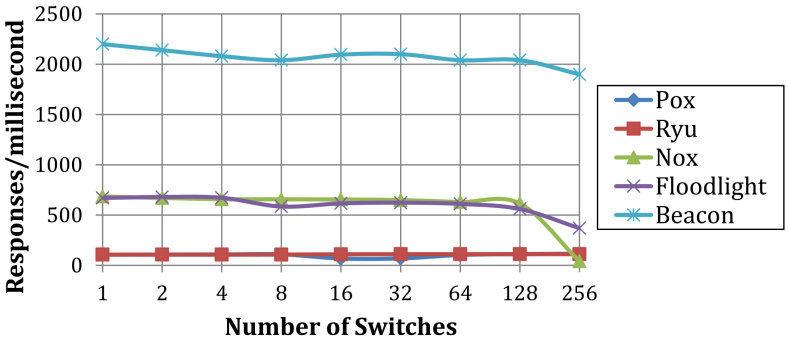
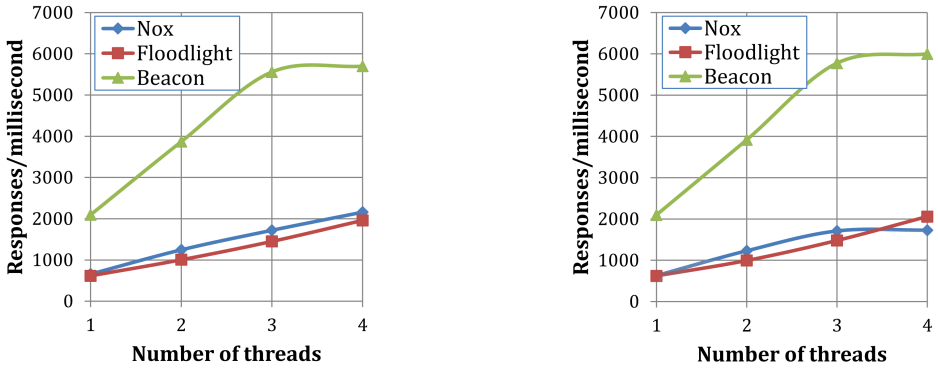


Fig. B.6 Le débit obtenu avec un nombre différent de switch (un thread).

switch émulé par Cbench est fixé à un. Tableau B.5 résume les résultats de l'évaluation, en notant que "rps/ms" représente *réponses par milliseconde*. Beacon surpasse de loin les autres. Il ne fait aucun doute que les contrôleurs basés sur Python offrent la plus faible performance. Floodlight et Nox sont proches les uns des autres, tandis que Nox a une latence plus petite. Ce classement reste inchangé dans la plupart des résultats des évaluations suivantes.

Dans cette partie, le nombre de switch émulés est modifiée et les autres paramètres fixes. Lorsque le nombre de switch augmente, le temps de latence augmente à peu près linéairement. Le temps de latence augmente de 0.01ms quand il n'y a qu'un seul switch, jusqu'à plus de 10ms avec 256 switches. Lorsque plusieurs switches sont connectés, une connexion TCP est maintenue pour chaque switch. Le contrôleur adopte généralement une méthode de Round-Robin, donc la relation linéaire. Le débit total (réponses/ms) obtenue avec des nombres différents de switches est représenté sur la Fig. B.6. Les performances sont relativement stables lorsque le nombre de switch est inférieure à 128. Lorsque le nombre atteint 256, une dégradation des performances apparaît sur tous les contrôleurs. Ceci implique que le maintien d'un grand nombre de connexions est coûteuse.

L'impact du nombre de thread utilisés par les contrôleurs est ensuite étudiée, avec un nombre fixe de switch. Deux numéros de switch est utilisé, 16 et 64. Lorsque Hyper-Threading est désactivé, le débit sont présentés dans la Fig. B.7. Pour l'intervalle entre 1 et 3, comme prévu, la performance augmente à peu près linéairement avec l'augmentation de



(a) 16 switches (b) 64 switches

Fig. B.7 Le débit obtenu avec un nombre différent de thread (HT-désactivé).

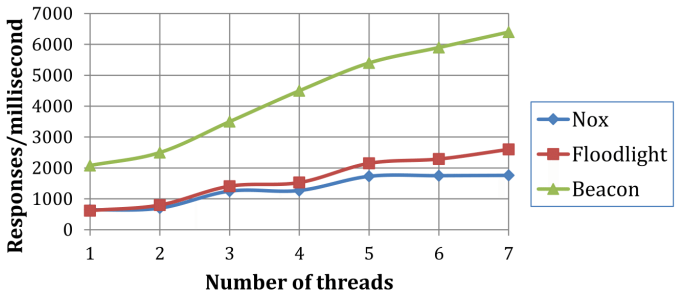


Fig. B.8 Le débit obtenu avec un nombre différent de thread (HT-activé).

thread. Dans les deux cas avec 16 et 64 switches, Nox et Floodlight se comportent à peu près le même. Beacon peut atteindre jusqu'à 6 millions de réponses par seconde avec 3 threads dans le cas avec 64 switches. Lorsque Hyper-Threading est activé, Fig. B.8 montre le résultat du débit total. Nox et Floodlight ont une courbe comme les escaliers, ce qui signifie que les résultats entre 1 et 2, 3 et 4, 5 et 6 sont assez proches. Cela indique que l'ajout d'un cœur logique du même cœur physique ne peut obtenir des performances limitées. Par rapport aux résultats de la Fig. B.7b, Beacon et Floodlight atteindre environ 10% gain de performance de en moyenne, tandis que Nox a une amélioration négligeable. Ainsi HT est réellement utile pour les contrôleurs basés sur Java, mais pas beaucoup pour les contrôleurs basés sur C++.

Dans l'évaluation précédente, la latence est mesurée avec un contrôleur libre. Cela signifie que la mémoire tampon de réception est vide, et tout paquets arrivé peut être traité immédiatement. Mais quand la charge de travail augmente, le tampon sera rempli progressivement. Il en résulte une latence supplémentaire en introduisant le temps d'attente dans le tampon. Donc, il y a un compromis entre la latence et le débit. Fig. B.9 affiche la corrélation entre le débit et la latence dans Beacon. Nous voyons que Beacon est capable de

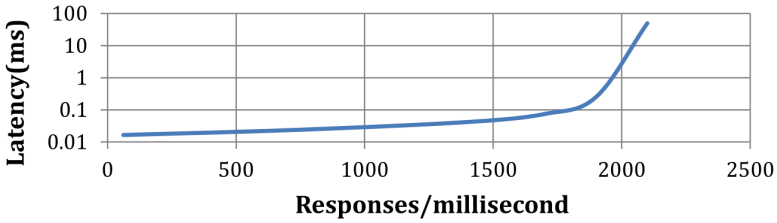


Fig. B.9 Corrélation entre le débit et la latence dans Beacon.

obtenir une faible latence (<0.1ms) ainsi que d'atteindre un débit élevé (vers 1750 rps/ms). Ceci est important pour les opérateurs de réseau pour équilibrer la latence et le débit en fonction des besoins pratiques.

Dans SDN, contrôleur distribué [93] est conçu pour améliorer la résilience et la scalabilité du plan de contrôle. Même si plusieurs contrôleurs sont distribué géographiquement, ils se comportent comme un plan de contrôle centralisé. Afin de fonctionner correctement, la synchronisation et la coordination entre plusieurs contrôleurs est critique pour l'ensemble du système. Dans cette section, nous nous concentrons principalement sur la synchronisation et la coordination entre les nœuds de multiples contrôleurs. Nous étudions les caractéristiques du trafic de synchronisation ainsi que la latence introduite par la coordination entre plusieurs nœuds. La mesure est basée sur un topologie linéaire avec 10 switches, et 10 hôtes sont connectés à chaque switch. Les switches sont uniformément distribués aux nœuds du contrôleur. Le contrôleur est ONOS (v1.1.0), et Hazelcast [43] est utilisé pour former le cluster dans ONOS.

Par défaut, ONOS crée un cluster logique full-mesh. Le cluster-head est le plus ancien contrôleur du cluster, qui envoie périodiquement la table de partition à d'autres contrôleurs. Lorsque les contrôleurs ne sont pas occupés, le taux de trafic entre contrôleur-contrôleur est en dessous de 5Mbps. Alors que dans un scénario lourd chargé où des milliers de *Packet_in* messages sont reçus par le contrôleur, le taux de trafic atteint à 90Mbps. Tableau B.6 montre le trafic moyen (avec contrôleur complet chargé) entre chaque paire de contrôleurs quand un contrôleur principal différent est choisi. Lorsque **c1** est primaire, le trafic global est beaucoup plus petite que les autres cas. Plus de recherches sur la sélection du contrôleur principal est nécessaire de comprendre et d'optimiser le trafic de synchronisation. Sur la base des résultats ci-dessus, le trafic de synchronisation totale (90Mps pour le pire des cas) est considéré comme négligeable par rapport à la bande d'aujourd'hui (plus de 10Gbps). Par conséquent, l'impact du trafic de contrôle sur le trafic de données est minime. D'un autre côté, l'impact du trafic de données sur le trafic de contrôle peut être atténué en mettant la priorité sur le trafic de contrôle.

Table B.6 Trafic vs. contrôleur principal (Mbps).

Primary	$c1 \leftrightarrow c2$	$c1 \leftrightarrow c3$	$c2 \leftrightarrow c3$	Total
c1	31.6	30.1	8.6	70.3
c2	89.6	12.3	27.0	128.9
c3	14.2	90.3	21.3	125.8

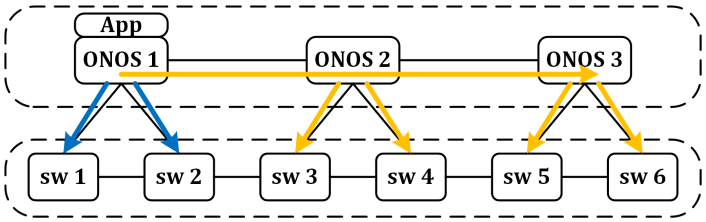


Fig. B.10 Le banc de test pour la installation de intent

La coordination entre les nœuds de contrôleurs multiples introduit une latence supplémentaire pour installer des règles de flux, en particulier lorsque les switches appartiennent à d'autres nœuds de contrôleurs. Fig. B.10 montre un banc de test de cluster de contrôleur avec 3 nœuds. 6 switches sont connectés linéairement, et chaque nœud contrôle 2 switches comme indiqué sur la figure. "Intention" est le concept défini par ONOS qui spécifie les désirs de contrôle de réseau sous la forme de la politique. ONOS peut traduire l'intention dans un ensemble de règles de flux pour être installé sur les switches connexes. Par exemple, dans notre cas de test, le script de test qui est utilisé pour générer des lots de "intentions" est en cours d'exécution sur ONOS1. Cette intention est de permettre que tous les hôtes sur sw1 et sw6 peut communiquer les uns avec les autres. Selon la définition de l'intention, ONOS1 devrait générer un ensemble de règles pour établir la connexion bidirectionnelle entre sw1 et sw6. Dans ONOS cluster, seul le nœud qui contrôle directement le switch peut installer des règles à ce switch. Par conséquent, même si les règles de SW3 et SW4 sont générés dans ONOS1, ils doivent être livrés à ONOS2 puis installé par ONOS2. Ce comportement agrandit

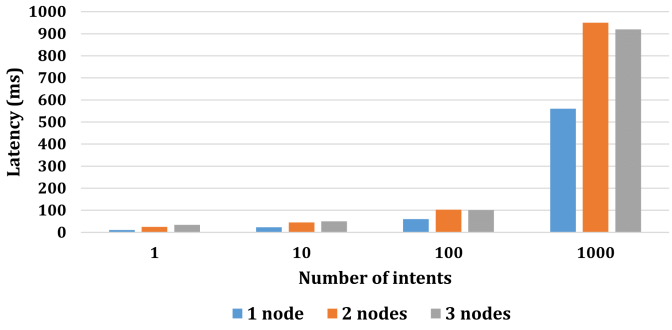


Fig. B.11 La latence d'installation de intent

le temps de latence pour installer des règles. Afin d'évaluer cette latence, nous comparons les résultats de latence avec un nombre différent de nœuds de contrôleur. Fig. B.11 montre les résultats de latence pour installer un nombre différent d'intention avec un nombre différent de nœuds. Comme prévu, plus grande latence existe dans un scénario de multiples nœuds par rapport au seul nœud en raison de la coordination entre les nœuds. Pour la grande taille d'intention (à savoir, 100 et 1000), lorsque la taille augmente, le temps de latence a tendance à diminuer. Parce que chaque nœud ne traite un plus petit nombre d'intentions en raison de la répartition de la charge de travail.

Dans cette section, une évaluation de la performance de cinq open-source SDN contrôleurs est effectuée. La performance n'est plus la seule dimension dans le choix de contrôleurs. La facilité d'utilisation, la reliabilité et la sécurité sont tout aussi importants. Étant donné que les contrôleurs sont de plus en plus complexe dans un rythme rapide, il est difficile pour les développeurs indépendant à suivre, et une grande communauté est généralement nécessaire. Pendant l'évaluation, nous voyons aussi une limitation claire du contrôleur centralisé dans un grand scénario, ce qui implique que le contrôleur distribué est nécessaire pour SDN. Différent de contrôleur centralisé, les contrôleurs distribués sont conçus pour un grand scénario ainsi que de fournir la tolérance de pannes. Les contrôleurs distribués peut augmenter le débit total en traitant simultanément sur plusieurs nœuds. Pendant ce temps, la latence due à la coordination et la synchronisation est également agrandie. Sur la base de ces faits, une est/ouest API plus efficace entre les nœuds du contrôleur doit encore une enquête plus approfondie.

5. Contrôle des ressources avec fine granularité

La performance du switches logiciel intégré sur le serveur ne soit pas aussi rapide et stable que le matériel dédié. Outre la surcharge introduite par la pile réseau dans le noyau Linux, il existe intensive conflits de ressources CPU entre les switches logiciels, le système d'exploitation et d'autres services. Alors que dans le scénario NFV, il est fréquent que plusieurs switches logiciels et fonctions réseau virtuelles coexistent sur le même serveur, et chaque switch et service sera donné d'une certaine quantité de ressources CPU pour remplir sa fonction. Pour les raisons précitées, le contrôle des ressources avec fine granularité est important pour le succès du déploiement de switch logiciel. Il permet non seulement d'offrir des performances stables et prévisibles pour répondre aux besoins de qualité de service définies par l'utilisateur, mais renforcer l'affectation des ressources afin de minimiser les interférences, ainsi que de maximiser la performance globale. Dans cette section, nous mettons en œuvre un prototype de Service Function Chaining (SFC) où plusieurs switches et les fonctions de

service sont nécessaires pour coordonner les uns avec les autres. Dans ce scénario de SFC, nous examinons l'efficacité de l'allocation des ressources à grains fins. Enfin, combinée avec des études antérieures sur l'évaluation des performances de switch logiciel, nous proposons une plateforme qui automatiser l'allocation des ressource par l'introduction de la théorie du contrôle classique.

Dans le cloud ou le centre de données où la technologie de virtualisation est largement déployée, plusieurs machines virtuelles (VM) partagent le même serveur sous-jacent pour exécuter des applications de calcul intensif, et les conflits de ressources entre les machines virtuelles ne peuvent pas être éviter. Les nouvelles technologies telles que la virtualisation du réseau et NFV sont confrontés au même problème de conflits de ressources. Afin de garantir la fonctionnalité de la performance du réseau, plusieurs switches logiciels et des fonctions de réseau virtuel sont consolidées et orchestrées.

CGroups (groupes de contrôle) [6] est la fonctionnalité du noyau Linux qui est utilisé pour contrôler l'utilisation des ressources de processus unique ou un ensemble de processus. Il est conçu pour fournir une interface unifiée pour réaliser un contrôle précis sur l'attribution, la priorité, en niant, la gestion et le suivi des différentes ressources du système (par exemple, CPU, mémoire, disque I/O, réseau, etc.).

Les processeurs actuels disposent de technologies pouvant influer sur la fréquence du CPU pour mieux réguler la consommation et le dégagement de chaleur selon l'utilisation. les fréquences du processeur peuvent être changé automatiquement en réponse à des événements ACPI, ou manuellement par les programmes de l'espace utilisateur. Le contrôle de fréquence du CPU est également mis en œuvre dans le noyau Linux, qui est appelé " CPUFreq ". CPUFreq permet de changer la vitesse d'horloge du CPU par les programmes de l'espace utilisateur. L'utilisation combinée de CGroups et CPUFreq peut réaliser le contrôle du processeur à grains fins sur OFsoftswitch.

Nous étendons notre vision de la performance du switch logiciel individuel à l'optimisation globale entre plusieurs switches basés sur l'allocation des ressources à grains fins. L'émergence de la SDN et NFV accélère le développement et le déploiement des switches logiciels et des fonctions de réseau virtuel pour satisfaire différentes exigences strictes de softwarization réseau. Service Function Chaining (SFC) est largement reconnu comme une application importante et prometteuse dans ce contexte. Une chaîne de service est défini comme un ensemble ordonné de fonctions de service abstraites qui doivent être appliquées aux paquets/flux sélectionné en tant que résultat de la classification. Le concept de Service Function Chaining (SFC) [78][95] permet d'orienter le trafic spécifique au service pour parcourir les fonctions de service de réseau dans l'ordre donné. SFC se réfère à découpler le déploiement des fonctions de réseau à partir d'infrastructures sous-jacentes. Fig. B.12 montre la vue

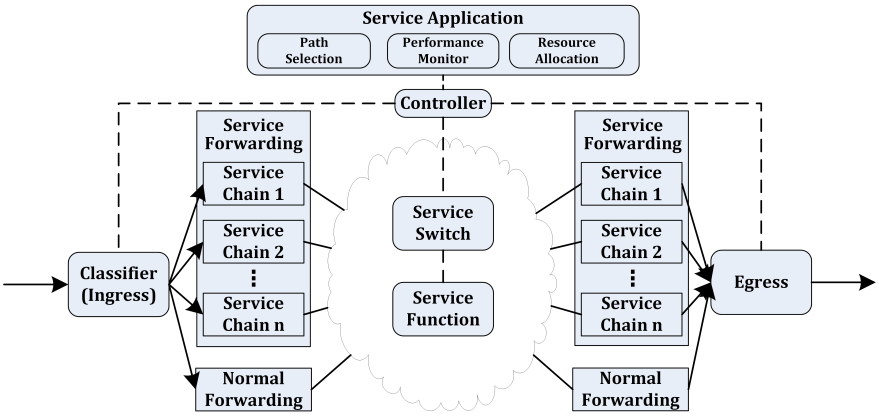


Fig. B.12 Le cadre de Service Function Chaining

d'ensemble du cadre SFC ainsi que le traitement des paquets en la couche de service à l'aide du Network Service Header (NSH). Les principaux composants sont Ingress/Egress, Switch de Service et Fonction de Service.

Afin de mettre en œuvre SFC, nous modifions OFsoftswitch et le contrôleur Ryu pour soutenir NSH. Nous suivons le projet de l'IETF [94] pour encapsuler NSH et paquet d'origine dans UDP. Dans OFsoftswitch, le tableau des flux est étendu pour supporter l'encapsulation/décapsulation de NSH ainsi que recherche/modification des champs de NSH correspondants. Par conséquent, OFsoftswitch peut être utilisé comme Ingress/Egress et le switch de service. En outre, OFsoftswitch peut également agir en tant que fonction de service, par exemple, Firewall, NAT, Gateway, etc. Pour le plan de contrôle, Ryu est sélectionné et modifié pour le soutien des opérations de NSH. Donc, le module de sélection de chemin est en mesure d'installer des règles NSH liées au switches de service. Le module de l'analyseur de performances est mis en œuvre par les messages standard de OpenFlow pour les statistiques de flux. Une API externe basée sur CGroup et CPUFreq est également prévu de réaliser un contrôle précis des ressources CPU.

Dans un environnement de cloud, le modèle pay-as-you-go a été largement adopté pour fournir Infrastructure-as-a-Service (IaaS). De même, NaaS est un modèle de gestion pour la prestation de services de réseau pratiquement dans cloud sur une base de pay-per-use. SFC peut être considéré comme l'une des applications typiques et importants de NaaS. SFC oblige les utilisateurs à spécifier explicitement le montant total des ressources à la réserve. Cependant, les utilisateurs ne peuvent grossièrement estimer la ressource nécessaire en raison de l'ignorance sur la performance de la plate-forme de fournisseurs de services. Par conséquent, les utilisateurs a généralement tendance à sursouscrire la ressource auprès du fournisseur de services, ce qui conduit à des coûts gonflé. Du point de vue du fournisseur de services, surbudgétisation réduit l'utilisation des ressources physiques

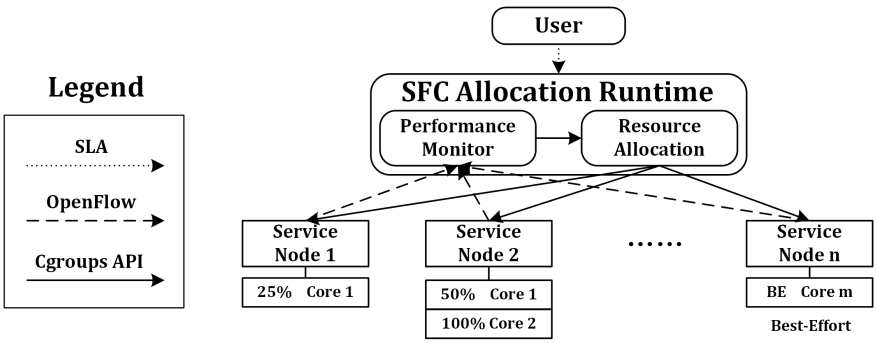


Fig. B.13 Le cadre de runtime d'allocation des ressources

sous-jacents. L'émergence de la SDN et NFV permet SFC d'être déployé d'une manière entièrement virtualisé. Cela apporte une grande flexibilité dans la gestion du réseau, en particulier dans l'allocation des ressources à grains fins. Sur la base de ces faits, nous nous efforçons de présenter un runtime d'allocation des ressources pour SFC qui peut satisfaire l'accord de Service-Level défini par l'utilisateur.

Fig. B.13 représente la vue d'ensemble du SFC runtime. Tout d'abord, l'utilisateur définit le SLA, à savoir, la bande passante d'entrée d'une chaîne de service prédéfinie. Nous supposons que la topologie de la chaîne de service est fixe. D'autre part, le runtime détermine la ressource nécessaire pour chaque nœud de service pour satisfaire le SLA, la bande passante de processeur est attribué à chaque nœud par l'API CGroups. Troisièmement, le runtime surveille périodiquement la bande passante en temps réel via l'API OpenFlow standard ou d'autres API externes. OpenFlow fournit des compteurs internes pour enregistrer le nombre de paquets qui correspondent à la table de flux spécifique, qui peuvent être utilisés pour calculer la bande passante en temps réel. L'erreur entre le SLA et la bande en temps réel est en outre utilisé pour ajuster la bande passante de CPU affectée. Étant donné que c'est un système en circuit fermé qui fournit des évaluations de l'état actuel, il est censé fournir une bande passante stable avec oscillation minimisée.

Deux types différents de SLA dans SFC, à savoir "Best-effort basé" et "Bande-Fixe basé" sont évalués séparément. Pour "Best-effort basé" SLA, nous fournissons 9 topologies différentes avec un nombre différent de nœuds (y compris les fonctions de service et switches de service). Dans chaque topologie, des nombres différents de cœurs de processeurs (2/3/4 cœurs) sont prévus pour l'allocation des ressources. Le résultat est représenté sur Fig. B.14. En appliquant l'allocation des ressources automatisée, le pourcentage d'amélioration obtenue est variée en fonction de différents nombres de nœuds et de cœurs. En observant la tendance générale d'amélioration, l'allocation effectue toujours supérieure ou égale à un défaut ordonnancement du système d'exploitation. Lorsque le nombre de nœuds de service

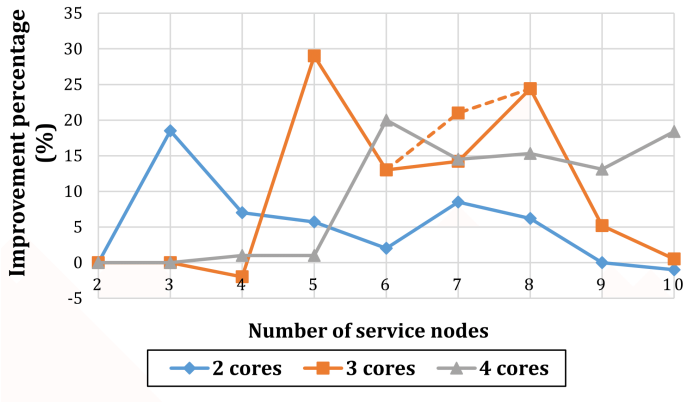


Fig. B.14 L'Amélioration réalisée avec l'allocation des ressources en Best-effort

(N_s) est plus petit que le nombre de cœurs (N_c), aucune amélioration est représentée, parce que chaque nœud peut occuper la ressource d'un cœur complet unique à la fois. Lorsque N_s est nettement plus grande que N_c, l'amélioration est également négligeable. En raison du grand nombre de nœuds, les conflit intensive de CPU et les interférences mutuelles sur les cœurs sont inévitables, peu importe quel type d'allocation est appliquée. La plupart des améliorations significatives qui sont plus de 15% se produisent généralement lorsque $N_c < N_s < 3 * N_c$.

L'évaluation sur l'allocation dynamique de runtime vise à valider sa capacité à fournir une "bande-fixe" SLA. Nous supposons que SLA définit seulement la chaîne de service spécifique et sa bande passante d'entrée. Afin de satisfaire aux exigences, le runtime fixe simplement la bande passante d'entrée au switch d'entrée selon la SLA et garantit que les ressources CPU affectés à d'autres nœuds de service sont suffisantes pour répondre aux besoins réels. La bande passante d'entrée est défini comme 400Mbps. Au début, le trafic ne contient que des paquets de 1500 octets qui appartiennent à Flow1, et le runtime fournit une bande passante en temps réel stable. Flow2 qui ne contient que des paquets de 512 octets rejoint le trafic au 100ème de seconde et quitte au 300ème de seconde. La bande passante en temps réel va changer en conséquence. Le runtime peut détecter ce changement, puis déclencher une réallocation de la bande passante du processeur. Les réactions dynamiques en temps réel l'allocation de bande passante et CPU bande passante sont présentés sur la Fig. B.15. Cela démontre que le runtime est capable de fournir une bande passante en temps réel stable, même quand il y a des variations sur les caractéristiques de la trafic. Le temps de réaction est d'environ 15 secondes.

Dans cette section, nous étendons notre vision de la performance du switch logiciel individuel à l'optimisation globale entre plusieurs switches. Service Function Chaining (SFC) est présentée comme un scénario typique où des switches logiciels multiples/fonctions

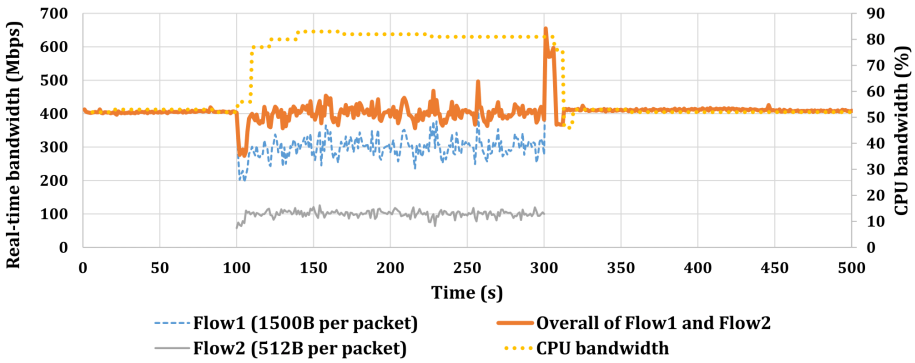


Fig. B.15 Le comportement dynamique d'allocation de runtime

de réseau virtuel coexistent partagent les mêmes ressources physiques sous-jacents. Nous suivons les projets de IETF pour mettre en œuvre un prototype de SFC en utilisant de Network Service Header (NSH). Sur la base de ce scénario de SFC, un runtime automatique est proposé de soutenir l'allocation dynamique des ressources. Il prend en charge deux types de SLA différents, à savoir "Best-effort basé" et "Bande-Fixe basé". Dans "Best-effort basé" SLA, l'utilisateur souscrit un montant fixe de ressources physiques/virtuels et les fournisseur de services mettent en œuvre la chaîne de service spécifique sur les ressources souscrites d'une manière best-effort. "Bande-Fixe basé" SLA vise à fournir une bande passante fixe en temps réel automatiquement et dynamiquement. Le runtime est proposé comme un cadre général qui peut être étendu pour supporter différents types de ressources et divers objectifs de niveau de service.

6. Conclusion

En raison de la tendance croissante de "Softwarization", la virtualisation devient la technologie dominante dans le centre de données et de l'environnement de cloud. SDN et NFV sont la technologie de réseau émergent pour faciliter le déploiement et la gestion du réseau par les API standards ouverts au lieu de manière spécifique au fournisseur traditionnel. Le switch logiciel est l'outil puissant pour mettre en œuvre des services de SDN et NFV. Cependant, la combinaison de switch logiciel et SDN est encore loin d'être bien étudié. Sur la base de ces faits, nous réalisons notre étude en suivant 3 directions: l'évaluation de switch logiciel, l'évaluation du contrôleur et le contrôle des ressources à grains fins.

Dans 3ème section, en raison de l'importance de la performance du switch logiciel, une évaluation systématique de la performance est réalisée. Nous choisissons deux représentants du switch logiciel OpenFlow, à savoir OpenvSwitch et OFsoftswitch. Nous tirons les conclu-

sions des différences et des similitudes entre les deux switches sélectionnés. Ces conclusions soulignent le scénario adapté à chaque switch logiciel. De plus, nous démontrons le principe de conception derrière le in-band contrôle. Nous discutons aussi la solution potentielle pour la latence supplémentaire introduite par in-band contrôle. Comme le contrôleur est la pierre angulaire de la réussite de l'architecture SDN, une évaluation juste et totalement reproductible du contrôleur est fourni dans 4ème section. Au-delà d'une évaluation simple, non seulement nous examinons les paramètres de système général tels que l'interpréteur de Python et Hyper-Threading pour évaluer leurs impacts, mais aussi concevoir des scénarios spécifiquement pour d'autres mesures. Sur la base des résultats de l'évaluation, nous constatons que la performance est plus la seule dimension dans le choix de contrôleurs.La facilité d'utilisation, la reliabilité et la sécurité sont tout aussi importants. On voit aussi la nécessité de contrôleur distribué dans de grands scénarios en raison de la limitation des performances du contrôleur centralisé. En raison de l'importance du trafic de synchronisation pour mettre en œuvre la fonctionnalité de contrôleur distribuée, une étude préliminaire est prévue pour examiner la caractéristique de trafic. La 5ème section examine la contention des ressources et l'allocation des ressources entre les switches logiciels multiples. Nous construisons un prototype de Service Function Chaining où plusieurs switches ou des fonctions de service partagent le même matériel sous-jacent. Un runtime automatique est proposé pour l'allocation des ressources à grains fins dans le scénario de SFC. Le runtime prend en charge deux types de SLA différents pour SFC, à savoir "Best-effort basé" et "Bande-Fixe basé". Le runtime est prouvé être capable de fournir une bande passante en temps réel stable selon SLA et de maximiser la chaîne de service spécifique d'une manière best-effort. Le runtime est présenté comme un cadre général qui peut être étendu pour supporter différents types de ressources et divers objectifs de niveau de service.

Dans cette thèse, nous essayons de résoudre un certain nombre de problèmes critiques dans le déploiement du switch logiciels dans un environnement de virtualisation de réseau, ce qui pose quelques problèmes ouverts. Par conséquent, notre étude peut être étendre et renforcée dans les directions suivantes.

Les évaluations doivent être effectuées sur un banc de test amélioré. Puisque les contrôleurs distribués se lient généralement à une grande quantité de ressources pour le calcul parallèle. Avec un banc de test puissant, nous pouvons en outre remplacer les outils d'évaluation émulés avec des tests de trafic réel. Par exemple, nous pouvons utiliser de nombreux switches logiciels au lieu de Cbench pour évaluer les contrôleurs, qui est plus proche de la réalité. De nombreux réseaux de recherche académique tels que Ofelia [24] ou PlanetLab [38] pourrait être pris en considération pour construire un banc de test pour le déploiement à grande échelle. Contrôleurs distribués conçus pour réseaux à grande échelle

gagnent malgré tout en popularité, avec le nombre de projets déjà existants. L'évaluation du contrôleur distribué est plus complexe que le contrôleur centralisé. Étant donné que la performance du contrôleur distribué repose non seulement dépend du ressources de calcul sous-jacent, mais dépend aussi de l'efficacité de la synchronisation, ainsi que l'emplacement des nœuds de contrôleur. Des efforts et des enquêtes pourraient suivre cette direction pour établir un écosystème pour développer et tester les est/ouest APIs efficacement. Le runtime est un cadre général qui peut être améliorée pour supporter différents types de ressources. Afin de contrôler la ressource pour le nœud de service plus précisément, nous pourrions construire un modèle de ressources performances pour chaque fonction de service. Outre le modèle de ressources performance, la prédiction de trafic est également utile pour améliorer l'allocation des ressources. Parce que la ressource nécessaire pour chaque fonction dépend directement des caractéristiques du trafic, et précise prédicat du trafic permet au système de réagir à l'avance pour réduire le temps de réponse.

Software Switch Deployment in SDN-enabled Network Virtualization Environment

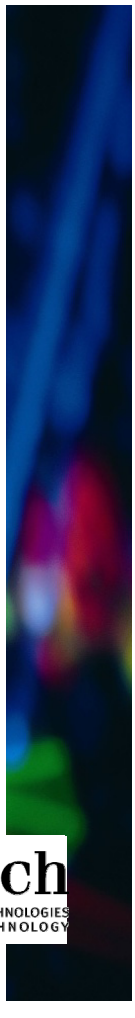
Yimeng ZHAO

RESUME : Avec la prévalence de logicielisation, virtualisation est devenue une technologie dominante dans des data-centres et clouds. Deux aspects principaux de la logicielisation de réseaux sont Software Defined Network (SDN) et Network Function Virtualization (NFV), dont un des outils essentiel sont les switches logiciels, à l'opposition des switches matériaux. Les switches logiciels sont également indispensables pour le succès de NFV. Cette thèse vise à relever des défis principaux dans la logicielisation de réseaux. Spécifiquement, elle porte sur le déploiement des switches logiciels dans un réseau virtuel avec SDN.

MOTS-CLEFS : Switch Logiciel, Software Defined Network, Network Function Virtualization, La Softwarization du Réseau

ABSTRACT : Due to the growing trend of "Softwarization", virtualization is becoming the dominating technology in data center and cloud environment. Software Defined Network (SDN) and Network Function Virtualization (NFV) are different expressions of "Network Softwarization". Software switch is exactly the suitable and powerful tool to support network softwarization, which is also indispensable to the success of network virtualization. Regarding the challenges and opportunities in network softwarization, this thesis aims to investigate the deployment of software switch in a SDN-enabled network virtualization environment.

KEY-WORDS : Software Switch, Software Defined Network, Network Function Virtualization, Network Softwarization



TELECOM ParisTech

EDITE

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY