PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Analyse de performances des réseaux programmables, à partir d'une trace d'exécution
Auteur: Author:	Adel Belkhiri
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Belkhiri, A. (2021). Analyse de performances des réseaux programmables, à partir d'une trace d'exécution [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/9988/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/9988/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

ANALYSE DE PERFORMANCES DES RÉSEAUX PROGRAMMABLES, À PARTIR D'UNE TRACE D'EXÉCUTION

ADEL BELKHIRI
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Décembre 2021

© Adel Belkhiri, 2021.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

ANALYSE DE PERFORMANCES DES RÉSEAUX PROGRAMMABLES, À PARTIR D'UNE TRACE D'EXÉCUTION

présentée par **Adel BELKHIRI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Alejandro QUINTERO, président
Michel DAGENAIS, membre et directeur de recherche
Soumaya CHERKAOUI, membre
Ferhat KHENDEK, membre externe

DÉDICACE

Je dédie ce travail à la mémoire de :
Ma chère défunte mère Zina,
la femme la plus courageuse et la plus généreuse qui puisse exister,
Mon cher défunt frère Mounir,
un homme exceptionnel au grand cœur,
. . .
Je vous aime,
et je ne vous oublierai jamais !

REMERCIEMENTS

En premier lieu, j'aimerais remercier mon directeur de recherche, Michel Dagenais, pour la confiance qu'il m'a accordée en acceptant d'encadrer cette thèse. Ses connaissances scientifiques, ses conseils constructifs, et ses qualités humaines d'écoute et de compréhension, m'ont permis de mener à bien ce travail.

Je tiens à exprimer ma gratitude à tous nos associés de recherche, et particulièrement Geneviève pour son aide tout au long de ce travail.

Aussi, je remercie vivement tous les membres du laboratoire de recherche DORSAL, et en particulier Houssem, Ahmad, Majid, Hani, Iman, Hervé, et Pierre.

Je remercie le CRSNG (Conseil de Recherches en Sciences Naturelles et en Génie du Canada), le Conseil Régional de Nord Pas de Calais-Picardie, le Ministère de l'Enseignement Supérieur et de la Recherche, et tous les partenaires du laboratoire, Ericsson, Ciena, EfficiOS, Google et Prompt, pour leur soutien financier.

Je présente mes remerciements, mon respect et ma gratitude à tous les enseignant(e)s et employé(e)s de Polytechnique Montréal.

Mes pensées vont aussi à mes amis Donna Grace, Meriem, Adnen, Hamdi, Nabouli, Sofiene, Makrem, Anis, et Tarek pour les bons moments passés ensemble.

Enfin, je remercie ma famille qui a été toujours là pour moi : mon père Bechir, mon frère Mohammed, et mes sœurs Houda, Besma, Kawther, et Wafa. Aussi, j'adresse mes remerciements à tous les membres de la famille Belkhiri.

RÉSUMÉ

De nos jours, plusieurs architectures réseau innovantes, telles que les réseaux définis en logiciel (SDN) et la virtualisation des fonctions de réseau (NFV), ont été proposées, afin de répondre aux besoins des applications modernes, en bande passante et flexibilité. L'objectif de ces architectures est d'augmenter la programmabilité des réseaux, afin de faciliter leur gestion, et rendre plus flexible le déploiement des nouveaux services. En effet, le SDN repose sur la centralisation de l'intelligence du réseau dans une entité appelée Contrôleur SDN, et la suppression des fonctions de contrôle dans les équipements de transfert de données. D'un autre côté, la NFV se base sur l'exploitation des fonctions réseau (p. ex., routage, filtrage, équilibrage de charge) sous la forme d'applications qui s'exécutent dans des machines virtuelles, au lieu d'utiliser du matériel spécifique, cher et peu flexible. Dans ces architectures, le logiciel devient de plus en plus important, en prenant en charge des fonctions qui étaient auparavant remplies par du matériel dédié. Toutefois, cette "softwarisation" favorise davantage l'occurrence des bogues de performance dans les réseaux, et complique énormément le diagnostic de leurs causes profondes.

Le travail de recherche que nous présentons dans cette thèse porte sur l'analyse de performance des solutions logicielles utilisées dans la mise en œuvre des infrastructures SDN et NFV. En effet, nous proposons une approche d'analyse qui se base sur le traçage dans la collecte des données de performance requises pour les analyses. Pour appliquer cette approche, nous avons instrumenté le code source de plusieurs logiciels, comme DPDK, Open vSwitch, et KVMGT. Les points de trace insérés permettent de mettre en lumière le fonctionnement du logiciel observé, et de décrire en détails les interactions entre ses différents mécanismes. Pour capturer les données exportées par ces points de trace, nous avons utilisé un outil qui s'appelle Linux Trace Toolkit Next Generation (LTTng). Ce traceur est particulièrement connu pour son faible surcoût et sa capacité à tracer le noyau Linux, ainsi que les applications qui s'exécutent dans l'espace utilisateur.

Afin de minimiser l'impact du traçage sur les performances du logiciel observé, nous avons fait des compromis entre le niveau de détails des données à exporter et le surcoût que nous estimons que l'utilisateur peut tolérer. Notre évaluation des surcoûts induits par la phase de collecte des données montre que l'impact du traceur sur le temps d'exécution du programme surveillé est très faible (<2%). Toutefois, le surcoût en termes d'espace de stockage est assez élevé. Pour atténuer ce surcoût, nous avons proposé une méthode qui permet de réduire la taille des traces générées. Cette méthode consiste à lancer en parallèle deux sessions de

traçage. La première session collecte des données de traçage, à partir d'un nombre limité des points de trace. Les données collectées sont analysées en ligne dans le but de détecter des anomalies de performance. D'un autre coté, la deuxième session collecte des données détaillées, à partir d'un nombre plus grand des points de trace. Les données de traçage générées par la deuxième session sont enregistrées dans la mémoire tampon du traceur, et ne sont sauvegardées sur disque que lorsque les analyses en ligne détectent un problème de performance. Les traces enregistrées sont analysées hors ligne afin d'identifier les causes de l'anomalie identifiée.

Par ailleurs, nous avons développé de nombreuses analyses qui sont capables de diagnostiquer des bogues de performance complexes, à partir des traces collectées. Ces analyses, qui se basent sur des techniques d'abstraction variées, calculent des métriques de performance adaptées, et déterminent les états des composants internes du logiciel cible. Nous avons également proposé une analyse pour dériver la topologie d'un réseau virtuel, à partir d'une trace d'exécution. Le but de cette analyse est d'identifier les chaînes de service dans l'infrastructure NFV, à travers la découverte des liens qui existent entre les fonctions réseau virtuelles (VNF). Étant donné que le réseau virtuel peut s'étendre sur plusieurs machines physiques, les traces collectées depuis ces machines sont synchronisées en se basant sur les relations de causalité qui existent entre les événements de traçage.

Dans cette recherche, nous avons relevé de nombreux défis techniques et conceptuels, tels que l'instrumentation du code source des logiciels à observer, la maîtrise du surcoût de traçage, la synchronisation des traces enregistrées sur des machines différentes, et le développement des analyses multiniveaux capables de couvrir plusieurs aspects de performance des logiciels surveillés. Les analyses élaborées sont implémentées sous la forme des modules d'extension à Trace Compass, un logiciel libre d'analyse de traces. En outre, les résultats de ces analyses sont affichés dans des interfaces graphiques interactives. L'utilisateur peut facilement utiliser ces interfaces pour identifier les causes profondes des bogues de performance qui se produisent dans les infrastructures SDN et NFV. Les différents cas d'utilisation que nous avons présentés démontrent l'efficacité de nos analyses dans le diagnostic des problèmes de performance variés. De plus, les nombreux tests que nous avons effectués prouvent que le surcoût induit par le traçage est très faible.

ABSTRACT

In recent years, several innovative network architectures, such as Software-Defined Networking (SDN) and Network Function Virtualization (NFV), have been proposed to meet the needs of modern applications for more bandwidth and flexibility. These architectures aim to make networks more programmable, in order to facilitate their management and to increase the flexibility of new services deployment. Hence, the SDN is based on moving the network intelligence from packet forwarding equipment to a central entity called SDN Controller. On the other hand, NFV is based on the use of applications running in virtual machines to execute network functions (e.g., routing, filtering, load balancing), instead of using expensive dedicated hardware. In these architectures, the software is becoming increasingly important, because it implements functions that were previously performed by hardware. Nevertheless, this "softwarization" causes the performance bugs in the network to become more frequent, and complicates the task of finding and analyzing their root causes.

The focus of the research work presented in this thesis is on the performance analysis of software used in SDN and NFV infrastructures. Hence, we propose an analysis approach that uses tracing to collect performance data from various sources, such as the kernel and userspace applications. To implement this approach, we instrumented the source code of several network applications, such as Open vSwitch, DPDK, and KVMGT. The inserted tracepoints enable shedding light on the operations of the observed software, and describing the interactions between its different mechanisms. To capture the data exported by these tracepoints, we used a tool called Linux Trace Toolkit Next Generation (LTTng). This tool is particularly known for its low overhead and its ability to trace the Linux kernel and userspace applications.

To minimize the impact of tracing on the performance of the observed software, we make a trade-off between the granularity of the tracing data and the overhead we believe the user can tolerate. The breakdown of the overhead induced by the data collection phase shows that the impact of the tracer on the execution time of the monitored application is minimal ($<2\%$). However, the tracing overhead in terms of storage space is significant. We proposed, therefore, an efficient technique to reduce the size of generated traces. This technique consists of configuring the tracer to start, in the same time, two tracing sessions. The first session collects tracing data from a limited number of tracepoints. We analyze the data collected from this session to detect potential performance anomalies using online analyses. On the other hand, the second session collects detailed data from a high number of tracepoints. The

tracing data generated by the second session are saved into the memory buffers of the tracer and stored on the disk only when the online analyses detect a performance problem. The result traces are analyzed offline to identify the root causes of the identified performance problem.

Moreover, we developed several analyses capable of diagnosing complex performance bugs from the collected traces. These analyses use various abstraction techniques to calculate adapted performance metrics and derive the states of the internal components of the traced software. We also proposed an analysis to recover the link-layer topology of a virtual network from an execution trace. This analysis aims to identify the service chains in the NFV infrastructure by discovering the links between the Virtual Network Functions (VNF). Since the virtual network can span over many physical machines, the traces that are collected from different sources are synchronized using the causal relationships between the trace events.

In this research, we overcame many technical and conceptual challenges, such as instrumenting the source code of the software to be observed, minimizing the tracing overhead, synchronizing the traces recorded on different machines, and implementing multilevel analyses capable of covering several aspects of the monitored software performance. We implemented our analyses as extension modules to Trace Compass, a free trace viewer and analyzer. The results of these analyses are displayed in interactive graphical views. The user can easily use these views to identify the root causes of performance bugs that may occur in SDN and NFV infrastructures. In short, the different use cases that we have presented demonstrate the effectiveness of our approach in diagnosing various performance issues. Furthermore, our numerous tests prove that the tracing overhead is extremely low.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xiii

LISTE DES FIGURES xiv

LISTE DES SIGLES ET ABRÉVIATIONS xvii

CHAPITRE 1 INTRODUCTION 1
1.1 Définitions et concepts de base 1
1.2 Éléments de la problématique 5
1.3 Objectifs de recherche 7
1.4 Contributions 7
1.5 Plan de la thèse 8

CHAPITRE 2 REVUE DE LITTÉRATURE 9
2.1 Les réseaux SDN 9
2.1.1 Architecture de SDN 10
2.1.2 Performances des réseaux définis par le logiciel 14
2.2 Virtualisation des fonctions réseau 18
2.2.1 Architecture des plateformes NFV 18
2.2.2 Approches d'accélération logicielle 21
2.2.3 Outils d'analyse de performance 22
2.3 Les coprocesseurs réseau 24
2.3.1 Analyse de performance des processeurs graphiques 25
2.3.2 Virtualisation des processeurs graphiques 28
2.4 Traçage 30
2.4.1 Outils de Traçage 30

2.4.2 Outils de lecture et d'analyse des traces 33
2.5 Synthèse 34

CHAPITRE 3 MÉTHODOLOGIE 37
3.1 Collecte de données 37
3.2 Analyse de données 38
3.2.1 Méthode d'analyse 38
3.2.2 Synchronisation des traces 39
3.3 Évaluation des coûts 40
3.4 Environnement de travail 41
3.4.1 Matériel 41
3.4.2 Logiciel 42
3.5 Axes de recherche 42

CHAPITRE 4 ARTICLE 1: DIAGNOSTIC AND TROUBLESHOOTING OF OPENFLOW–ENABLED SWITCHES USING KERNEL AND USERSPACE TRACES 46
4.1 Abstract 46
4.2 Introduction 47
4.3 Related work 49
4.4 Background 52
4.4.1 Open vSwitch Architecture 52
4.4.2 Kernel/userspace interface 54
4.5 Design of the proposed framework 55
4.5.1 Data collection 55
4.5.2 Analysis and Visualization 60
4.6 Use Cases 64
4.6.1 OVS performance tuning 65
4.6.2 Overcommitment of the datapath cache 68
4.6.3 Kernel/userspace flow key mismatch 69
4.6.4 Microflow and megaflow cache 70
4.7 Overhead Analysis 73
4.8 Conclusion 75

CHAPITRE 5 ARTICLE 2: PERFORMANCE ANALYSIS OF DPDK-BASED APPLICATIONS THROUGH TRACING 77
5.1 Abstract 77
5.2 Introduction 78

5.3 Related Work 81
5.4 DPDK Overview 83
5.4.1 Poll Mode Drivers (PMDs) 83
5.4.2 DPDK Libraries 85
5.5 Design of the Proposed Framework 88
5.5.1 Data Collection 89
5.5.2 Analysis and Visualization 92
5.6 Use Cases 100
5.6.1 The Pipeline sample application 101
5.6.2 The Eventdev pipeline sample application 104
5.6.3 Use Case Discussion 109
5.7 Overhead Analysis 109
5.8 Conclusion 112

CHAPITRE 6 ARTICLE 3: PERFORMANCE ANALYSIS OF VIRTUALIZED GPUS THROUGH HOST-BASED TRACING 115
6.1 Abstract 115
6.2 Introduction 116
6.3 Background and related work 118
6.3.1 GPU Virtualization Approaches 119
6.3.2 GPU Performance Analysis Tools 121
6.4 Architecture of the Proposed Framework 123
6.4.1 Data Collection Subsystem 124
6.4.2 Performance Analysis Subsystem 124
6.5 Use Cases 129
6.5.1 KVMGT Scheduling Algorithm 129
6.5.2 Contention Between vGPUs 132
6.5.3 Virtual Machine Placement 134
6.6 Tracing Cost Analysis 136
6.7 Conclusion 137

CHAPITRE 7 ARTICLE 4: VIRTUAL NETWORKS LINK-LAYER TOPOLOGIES DISCOVERY THROUGH HOST-BASED TRACING 139
7.1 Abstract 139
7.2 Introduction 139
7.3 Related Work 142
7.4 Design of the Proposed Framework 143

7.4.1 Data collection 144
7.4.2 Data Analysis 146
7.5 Cost Analysis 152
7.6 Conclusion 154

CHAPITRE 8 DISCUSSION GÉNÉRALE 155

CHAPITRE 9 CONCLUSION 158
9.1 Limitations de la solution proposée 159
9.2 Améliorations futures 160

RÉFÉRENCES 162

LISTE DES TABLEAUX

Table 4.1	Description of a subset of events that are fired during packet forwarding	59
Table 4.2	Description of a subset of events that are triggered during flows revalidation	61
Table 4.3	Hardware and software experimental configurations	65
Table 4.4	Trace size and analysis time according to transmission bit rate	73
Table 4.5	Impact of tracing overhead on OVS forwarding performance	74
Table 5.1	Description of a subset of tracepoints used to instrument the Pipeline library	91
Table 5.2	Description of a subset of tracepoints used to instrument the Eventdev library	93
Table 5.3	Hardware and software experimental configurations	101
Table 5.4	Overhead induced by the collection and analysis of the performance data	112
Table 6.1	Tracepoints Description	125
Table 6.2	Captions of Figure 2	126
Table 6.3	Hardware and software experimental configurations	130
Table 6.4	Tracing overhead	137
Table 7.1	Subset of the tracepoints leveraged by the first method	146
Table 7.2	A subset of the tracepoints leveraged by the second method	149
Table 7.3	Hardware and software experimental configurations	152
Table 7.4	Overhead incurred by the collection and analysis of the tracing data	154

LISTE DES FIGURES

Figure 2.1	Architecture simplifiée de SDN	11
Figure 2.2	Diagramme simplifié illustrant le traitement d'un paquet par un commutateur OpenFlow	13
Figure 2.3	Architecture simplifiée d'une plateforme NFV	19
Figure 2.4	Analyse de trace avec Trace Compass	34
Figure 3.1	Démarche adoptée pour l'analyse de performance des applications réseau cibles	38
Figure 3.2	Nos activités de recherche ont abouti à quatre articles scientifiques	43
Figure 4.1	Architecture of Open vSwitch	53
Figure 4.2	Architecture of the proposed framework	56
Figure 4.3	Packets processing paths within OVS and their corresponding tracepoints	58
Figure 4.4	Excerpt of the interval-time tree-based data structure on which our framework is based	63
Figure 4.5	SDN topology of our test environment	65
Figure 4.6	Rate of upcalls triggered by the port issuing the traffic	66
Figure 4.7	Time spent in revalidating the flows cached by the datapath	66
Figure 4.8	Flow-limit value calculated by revalidator threads based on the time spent in revalidation cycles	67
Figure 4.9	Number of flow rules cached by the datapath	67
Figure 4.10	Variation of the flow-limit value after fixing the issue	67
Figure 4.11	Difference between the packet rates of flow rules	68
Figure 4.12	Number of flow rules cached by the datapath	69
Figure 4.13	Caching duration of installed flow rules	69
Figure 4.14	Packet rate at the reception for the traffic generated with double MPLS headers	70
Figure 4.15	Waiting queue length of handler threads	71
Figure 4.16	User upcalls issuing rate per port	71
Figure 4.17	User upcalls are issued because of a flow key mismatch	71
Figure 4.18	EMC and megaflow cache usage	72
Figure 4.19	Adopted tracing architecture	75
Figure 5.1	Primary and secondary lcores in a DPDK application	86

Figure 5.2 A super-pipeline is composed of two or more pipelines interconnected by software queues 87
Figure 5.3 Architecture of the proposed framework 88
Figure 5.4 Excerpt of the tree-based data structure on which our framework is based. 97
Figure 5.5 Example of an XY view showing the rate at which packets matched with the flow rules of an ACL classification table. 100
Figure 5.6 Example of a time graph view showing the status of application lcores, alongside the status of the services attached to service lcores. 100
Figure 5.7 View generated by our tool showing the architecture of our super-pipeline and the flow of received traffic. 102
Figure 5.8 The rate of packet forwarding is lower than the rate of the packet reception. 103
Figure 5.9 Traffic generated by TRex did not overflow neither the RX buffer of the first NIC nor the TX buffer of the second NIC. 104
Figure 5.10 Latency of the software queues in the super-pipeline. 104
Figure 5.11 Occupancy of the software queues connecting the pipelines used by our application. 105
Figure 5.12 Architecture of the event device used by our monitored application. . 105
Figure 5.13 Occupancy of the RX buffer of the first vhost-user NIC. 106
Figure 5.14 The packet dequeue rate of Worker1 (red) is characterized by a considerable fluctuation while it is stable for Worker0 (blue). 107
Figure 5.15 The effective RX spins metric shows that the worker threads responsible for processing packets in the first stage were underloaded 108
Figure 5.16 According to the metric percentage of effective RX spins, Worker1 (second stage) looks more overloaded than Worker0 (first stage). . . . 108
Figure 5.17 Zooming in the view calculating the percentage of effective RX spins 108
Figure 5.18 During the second stage of the pipeline, all the flows were aggregated in a single elephant flow 109
Figure 5.19 Average overhead for recording one event in tracer's memory buffer . 113
Figure 6.1 Architecture of the proposed framework 123
Figure 6.2 States of a GPU request along with the corresponding tracing events 126
Figure 6.3 Excerpt of the tree-based data structure upon which our analyses are based 127
Figure 6.4 Evolution of the waiting queue length 128
Figure 6.5 GPU requests latency metric 128

Figure 6.6 Latencies of benchmarks when executed in multi-VMs and VM only contexts 130
Figure 6.7 GPU usage related to two executions of Gaussian benchmark in VM1 131
Figure 6.8 Wait latencies of GPU requests in KVMGT and host driver queues along with their execution times 132
Figure 6.9 The execution time of Gaussian benchmark (VM1) increased when it was concurrently executed with Hotspot benchmark (VM2) 133
Figure 6.10 GPU usage during a concurrent execution of Gaussian/VM1 (red) and Hotspot/VM2 (blue) benchmarks 133
Figure 6.11 When vGPU2 is sending big workloads (blue), the performance of vGPU1 (red) is affected 134
Figure 6.12 A VM alternating CPU and GPU computing phases 135
Figure 6.13 VM-532 and VM-533 are preempting each other because they are competing for the same CPU resources 136
Figure 6.14 After applying the optimized VM placement algorithm, VM-532 is rarely preempted by the GPU intensive VM-535 136
Figure 7.1 Architecture of the proposed framework 144
Figure 7.2 Virtual network of reference for our framework 145
Figure 7.3 Switches use their FDB to forward frames between network segments 147
Figure 7.4 Graphical representation of a VN composed of four containers connected through three switches 148
Figure 7.5 Graphical representation of a VN composed of dozens of containers connected through three switches 150
Figure 7.6 An excerpt of a synchronized trace indicating the path taken by a packet sent by a container in Host1 to another container in Host2 . . 152

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CTF	Common Trace Format
DPDK	Data Plane Development Kit
E/S	Entrée/Sortie
EAL	Environment Abstraction Layer
ETSI	European Telecommunications Standards Institute
FDB	Forwarding Database
Ftrace	Function tracer
GPGPU	General-purpose Processing on Graphics Processing Units
GPU	Graphics Processing Unit
HPC	High Performance Computing
I/O	Input/Output
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IoT	Internet of Things
LTTng	Linux Trace Toolkit next generation
MANO	MANagement and Orchestration
MIB	Management Information Base
NF	Network Function
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVO	NFV Orchestrator
NIC	Network Interface Card
NPU	Network Processing Unit
NTP	Network Time Protocol
OF	OpenFlow
OVS	Open vSwitch
OVSDB	Open vSwitch DataBase
PMD	Poll Mode Driver
PNF	Physical Network Function
POF	Protocol-Oblivious

QoS	Quality of Service
RAM	Random Access Memory
RCU	Read Copy Update
RDMA	Remote Direct Memory Access
SDN	Software-Defined Networking
SHT	State History Tree
SIMT	Single Instruction, Multiple Threads
SMP	Symmetric Multiprocessing
SNMP	Simple Network Management Protocol
SR-IOV	Single Root–Input Output Virtualization
VIM	Virtualized Infrastructure Manager
VNE	Virtual Network Embedding
VNF	Virtual Network Function
VNFM	VNF Manager
sFlow	sampled Flow
XDP	eXpress Data Path

CHAPITRE 1 INTRODUCTION

De nos jours, le réseau informatique est devenu un véritable pilier du succès de l'entreprise, et un facteur déterminant de sa croissance. Le passé démontre que les entreprises sont susceptibles de subir des pertes financières lourdes, ou au moins un dérèglement de leurs activités, en raison des pannes dans leurs réseaux.

D'une manière générale, le réseau peut être affecté par des bogues de performance et des bogues fonctionnels. Un bogue de performance, contrairement à un bogue fonctionnel, préserve la connectivité du réseau, mais réduit ses performances. Souvent, l'impact de ce type de bogues sur le fonctionnement du réseau est non négligeable (p. ex., diminution du débit, augmentation des latences, et gaspillage des ressources). Par ailleurs, l'identification des bogues de performance dans le réseau est particulièrement difficile à cause de leur sémantique, puisque ces bogues n'induisent pas un échec des opérations réseau. Elle est, en outre, compliquée en raison de la nature fondamentale de ces opérations, qui sont asynchrones, sensibles au temps, et sujettes à des erreurs.

Ces dernières années, plusieurs nouvelles architectures réseau ont été proposées, dans le but de rendre les réseaux modernes plus flexibles, et plus faciles à gérer. Parmi ces architectures, le réseau défini par le logiciel (Software-Defined Networking ou SDN) et la virtualisation des fonctions réseau (Network Functions Virtualization ou NFV) sont de plus en plus populaires. Avec ces deux architectures, le logiciel a acquis plus d'importance, ce qui a accroît la programmabilité et l'agilité des réseaux, mais également la probabilité d'occurrence des bogues de performance, ainsi que leur complexité.

L'objectif de cette thèse est d'analyser l'efficacité de certains logiciels sur lesquels reposent ces deux architectures, et proposer des méthodes et des analyses capables de diagnostiquer leurs bogues de performance.

1.1 Définitions et concepts de base

Le SDN est une architecture réseau innovante qui se base sur la centralisation de l'intelligence du réseau dans une entité logique appelée contrôleur SDN, et la suppression des fonctions de contrôle au niveau des équipements de transfert de données. Dans cette architecture, le logiciel gagne de plus en plus d'importance, par la prise en charge des fonctions qui étaient auparavant remplies par le matériel, créant ainsi des réseaux plus agiles. Ces derniers sont capables de mieux répondre aux besoins des applications actuelles en bande passante, flexibilité, et

résilience. Toutefois, cette nouvelle organisation des fonctions réseau augmente davantage le risque de panne. En effet, les logiciels sont plus susceptibles aux bogues qui, désormais, peuvent se produire à plusieurs endroits, tels que le plan de transfert, le contrôleur SDN, les services réseau, et le système d'exploitation.

La littérature rapporte quelques outils qui sont dédiés pour le diagnostic des problèmes de performance en SDN. Souvent, ces outils basent leur fonctionnement sur la collecte des statistiques à partir des équipements réseau, et le calcul des paramètres dénotant leur consommation de ressources (p. ex., pourcentage d'utilisation de la mémoire centrale). L'inconvénient majeur de ces outils est qu'ils sont incapables de déterminer les causes profondes des problèmes de performance détectés. La raison derrière cet échec se rapporte généralement soit à la granularité des données collectées, soit au surcoût induit par les méthodes utilisées pour ce faire. À titre d'exemple, certains outils utilisent des protocoles de surveillance, tels que SNMP et Netflow, pour collecter des données de haut niveau. Malheureusement, la granularité de ces données ne permet pas la détermination des causes des bogues qui affectent les performances du réseau.

D'un autre côté, la NFV désigne une approche réseau qui vise à remplacer du matériel dédié par des logiciels mettant en œuvre des fonctions réseau diverses. Ces logiciels sont exécutés par des machines virtuelles (ou des conteneurs), sous le contrôle d'un hyperviseur. Les plateformes NFV ont souvent recours à différents types d'accélérateurs pour réduire l'écart de performance entre les fonctions virtualisées et leurs équivalents physiques. Bien que les solutions logicielles dédiées à l'accélération du traitement des paquets réseau soient nombreuses, la plupart d'entre elles sont développées sur la base de techniques de contournement du noyau Linux. Ces techniques visent à traiter les paquets en dehors de la pile réseau du noyau, afin d'éviter les goulots d'étranglement qui se trouvent dedans. À titre d'exemple, DPDK [1] est un accélérateur logiciel qui implémente le contournement du noyau, et qui est exploité par de nombreuses applications réseau de haute performance.

Quoi qu'il ait été prouvé que les accélérateurs logiciels permettent d'améliorer les performances de traitement des paquets, leur utilisation par les applications pose des défis majeurs. Ces derniers sont principalement en lien avec la surveillance des performances des applications accélérées et le diagnostic de leurs bogues. En effet, les outils de surveillance et de débogage traditionnels, dont le fonctionnement se base sur le noyau, cessent d'être opérationnels lorsque l'application surveillée contourne ce dernier [1]. En outre, ces accélérateurs proposent souvent des choix de conception et d'implémentation très variés. Cela va sans dire

1. Par exemple, on ne peut pas utiliser la commande *ifconfig* pour accéder aux statistiques des cartes réseau lorsqu'elles ne sont pas gérées par le noyau.

qu'une mauvaise conception ou une configuration inadéquate peuvent causer une réduction des performances de l'application accélérée.

De même, les VNF peuvent solliciter une forme d'accélération matérielle afin d'atteindre leurs objectifs de performance. Dans ce contexte, le processeur graphique représente sans aucun doute un accélérateur idéal, étant donné sa disponibilité et l'existence de plusieurs bibliothèques facilitant sa programmation. En effet, le processeur graphique est très connu pour son architecture hautement parallèle, et sa capacité à exécuter efficacement certains types de traitement. Par ailleurs, il existe de nombreuses architectures haute performance qui se basent sur le processeur graphique pour délester certaines charges de calcul, qui sont habituellement traitées par le processeur central.

Heureusement, il existe des plateformes de virtualisation capables de mettre à la disposition des VNF, des instances de processeurs graphiques virtuels. Toutefois, surveiller les performances de ces équipements virtuels et diagnostiquer les problèmes liés à leur utilisation sont un véritable défi. En effet, les outils d'analyse existants ne sont pas en mesure de fournir des informations détaillées sur le fonctionnement des processeurs graphiques virtuels. À titre d'exemple, ils sont incapables de montrer comment les ressources du processeur graphique physique sont consommées par les machines virtuelles, et de détecter certaines anomalies qui découlent de cette consommation. En effet, la couche de virtualisation mise en œuvre par l'hyperviseur, et les abstractions aux ressources du processeur graphique, limitent la visibilité de ces outils, et entravent leur capacité à diagnostiquer des problèmes de performance.

De manière générale, les administrateurs utilisent divers types d'outils pour diagnostiquer les bogues qui se produisent dans un réseau. Parmi ces outils, les débogueurs sont très populaires. Un débogueur est un outil d'analyse qui permet d'exécuter un programme pas-à-pas, d'afficher le contenu de ses variables, et de mettre des points d'arrêt au niveau de ses instructions. De nombreux outils, comme NDB (Network Debugger) [2], implémentent diverses primitives de débogage, dans le but d'analyser les bogues réseau. Ces outils sont efficaces dans la détermination des causes de certains bogues fonctionnels, tels que ceux se rapportant à la logique de transfert des paquets (p. ex., boucles, trous noirs, etc.). Cependant, ils sont peu utiles lorsque ces bogues sont imprévisibles ou difficiles à reproduire. En outre, ils sont incapables de diagnostiquer les bogues qui affectent les performances des programmes impliqués dans le traitement des paquets réseau.

Les testeurs de logiciels ont souvent recours à des outils d'étalonnage pour mettre en œuvre divers types de tests non fonctionnels (p. ex., les tests de performance et de résistance). L'étalonnage est une technique qui consiste à soumettre le système sous test à des charges spécifiques, dans le but d'obtenir une évaluation objective de ses performances et de sa

robustesse. Les outils d'étalonnage présentent au moins deux limitations. D'abord, ces outils peuvent aider à découvrir certains problèmes de performance dans le logiciel sous test, mais ne sont pas capables de déterminer leurs causes fondamentales. Aussi, ces outils sont incapables de découvrir les bogues de performance qui peuvent apparaître lorsque le système est utilisé dans un environnement de production, puisque leur fonctionnement se base sur l'emploi de charges synthétiques.

Le traçage est un procédé logiciel permettant la journalisation des évènements qui se produisent dans un système informatique. Les outils responsables de la mise en œuvre de ce procédé sont appelés des traceurs. En effet, le traçage se distingue de la journalisation ordinaire (logging) par la granularité des détails avec laquelle les informations concernant les évènements générés sont enregistrés. Il est à noter que le traçage requiert l'insertion des points de trace dans le flot d'exécution de l'application à surveiller. Un point de trace est un appel à une fonction chargée de l'enregistrement de certaines données dans la mémoire tampon du traceur. Le surcoût induit par cet appel est presque nul (quelques cycles seulement pour être précis), lorsque le point de trace n'est pas activé.

Il existe deux techniques permettant l'ajout des points de trace au code de l'application, qui sont à savoir l'instrumentation statique et l'instrumentation dynamique. La première technique consiste à insérer des points de trace dans des emplacements clés du code source de l'application. Bien qu'un traçage basé sur cette technique présente un surcoût faible, l'accès au code source de l'application et la nécessité d'une étape de recompilation limitent son utilisation à des contextes bien précis. Quant à la deuxième technique, elle consiste à insérer des points de trace dans le fichier binaire de l'application, ou directement dans son espace mémoire, lorsque celle-ci est en cours d'exécution.

Dans le but d'élaborer des analyses capables de déceler des problèmes de performance dans les infrastructures SDN et NFV, nous avons fait le choix d'utiliser le traçage comme méthode de collecte des données. En effet, le traçage présente de nombreux avantages par rapport aux autres méthodes de collecte de données. Premièrement, il permet de cueillir des données de performance d'une granularité fine, ce qui est essentiel pour pouvoir dépister les bogues de performance difficiles à identifier par les autres techniques. Deuxièmement, le traçage permet de collecter des données à partir de toutes les couches de la pile logicielle de l'application cible, rendant ainsi possible l'obtention des informations nécessaires à l'identification des anomalies, et à la compréhension de leurs causes, que ce soit au niveau du noyau ou de l'application. Troisièmement, la plupart des outils de traçage sont conçus pour générer un surcoût minimal, ce qui permet d'étudier le comportement de l'application cible dans un environnement de production, avec des charges réelles.

1.2 Éléments de la problématique

Comme mentionné ci-dessus, notre approche pour évaluer l'efficacité des applications réseau cibles et diagnostiquer leurs bogues de performance repose sur l'analyse des traces d'exécution. Toutefois, l'emploi du traçage dans notre recherche pose de nombreux défis, à la fois d'ordre scientifique et technique.

Le premier défi à relever consiste à instrumenter manuellement le code des applications réseau, qui sont impliquées dans les infrastructures SDN et NFV. En effet, l'instrumentation automatique du code (source ou binaire) est très restrictive, parce qu'elle n'offre pas la flexibilité nécessaire à la collecte des données de performance adaptées. D'autre part, le noyau Linux présente plusieurs points de trace qui peuvent nous être utiles dans l'élaboration des analyses de performance. Cependant, nous estimons que ces points de trace ne sont pas suffisants pour réaliser une analyse détaillée de l'application cible. À notre avis, une instrumentation additionnelle, probablement au niveau de l'espace utilisateur, pourrait être requise, afin de couvrir certains aspects de performance cachés de l'application.

La mise en place d'un système de mesure basé sur l'instrumentation statique du code source de l'application cible est la technique que nous privilégions pour collecter des données de performance. Bien que cette technique implique la recompilation de l'application, comme nous l'avons expliqué précédemment, elle est de loin la plus performante et la plus flexible. Toutefois, la mise en œuvre de cette technique n'est pas aussi simple qu'il n'y paraît, car elle nécessite une connaissance approfondie de l'architecture et du fonctionnement de l'application. Elle requiert, en outre, une maîtrise des langages de programmation utilisés dans le code source. La lecture de grandes portions du code source est, la plupart du temps, inévitable avant l'insertion d'un point de trace. De plus, c'est une pratique courante de raffiner l'instrumentation effectuée, afin d'adapter les données exportées aux besoins des analyses, ou réduire le surcoût induit.

Il est parfois nécessaire de tracer en même temps plusieurs machines connectées à travers un réseau, afin d'étudier le fonctionnement d'une application distribuée, et découvrir ainsi ses goulots d'étranglement. Dans ce cas, on a besoin de combiner les traces collectées à partir de ces machines en une seule trace unifiée, pour pouvoir analyser les interactions entre les composantes de l'application en question. Cette procédure pose, toutefois, le problème du maintien de l'ordre entre les évènements issus des traces différentes, puisque les machines se basent sur des horloges différentes, et n'utilisent pas une même référence temporelle. Le choix d'un algorithme de synchronisation des traces, qui soit précis et efficace, est essentiel pour le développement des analyses pertinentes et, de ce fait, constitue l'un des enjeux de notre

recherche.

Il convient de noter que la maîtrise des surcoûts liés au procédé du traçage est vitale pour la viabilité et l'extensibilité de la solution d'analyse à proposer. Cela signifie que nous devons, d'un côté, évaluer rigoureusement ces surcoûts et, d'un autre côté, proposer des techniques permettant de les garder au minimum. De manière générale, nous pouvons évaluer le surcoût de traçage sur la base de la quantité des ressources (i.e., processeurs centraux, mémoire centrale, et les disques de stockage) requise pour produire la trace d'exécution. En effet, il est clair que le temps d'exécution de l'application tracée tend à augmenter avec l'exécution du code associé aux points de trace actifs. Par ailleurs, les fils d'exécution du traceur sont susceptibles d'entrer en concurrence pour les ressources avec ceux de l'application tracée, ce qui peut potentiellement causer une augmentation du temps d'exécution de cette dernière.

La première étape vers la maîtrise du surcoût de traçage est sans doute l'élaboration d'une instrumentation efficace. En effet, pour réduire la fréquence de génération des évènements de traçage, nous avons besoin de choisir soigneusement les emplacements des points de trace. À titre d'illustration, il est plus probable que la fréquence de génération d'événements soit élevée lorsqu'un point de trace est inséré dans une boucle ou une fonction en ligne, plutôt que lorsqu'il est inséré dans une structure conditionnelle. De plus, un autre facteur à considérer lors de notre démarche pour minimiser le surcoût de traçage est le choix du nombre de points de trace à activer et la quantité de données à exporter à travers ces derniers. De toute évidence, le développement de notre solution d'analyse va nous contraindre à faire des compromis entre le niveau de détail des données à analyser, et le surcoût de traçage que nous estimons que l'utilisateur peut tolérer.

Fournir des techniques d'abstraction permettant de produire des informations cohérentes et de haut niveau, à partir des évènements de traçage, bas niveau et ponctuels, est un autre défi de taille. En effet, les évènements de traçage sont généralement difficiles à comprendre, surtout lorsqu'ils ne sont pas associés à des contextes ou corrélés avec d'autres évènements. Les informations abstraites produites, en revanche, doivent être faciles à assimiler et à interpréter. En d'autres mots, elles doivent permettre d'analyser le comportement de l'application tracée sous plusieurs angles, repérer des problèmes de performance, et identifier leurs causes profondes. Il va sans dire que la taille des traces peut augmenter considérablement, dépendamment de la fréquence de génération des évènements et de la durée de la session de traçage. Il est donc de notre responsabilité de proposer des algorithmes et des structures de données efficaces permettant à notre solution d'analyse de passer à l'échelle.

1.3 Objectifs de recherche

Les travaux entrepris dans le cadre de cette thèse visent à élaborer des outils et des méthodes efficaces, qui permettent d'analyser les performances des réseaux programmables, et ce en se basant sur les techniques de traçage. De façon plus précise, notre objectif de recherche est de fournir des outils capables de diagnostiquer, à partir d'une trace d'exécution, les bogues de performance (p. ex., goulots d'étranglement) des logiciels impliqués dans la mise en œuvre des architectures SDN et NFV.

À partir de cet objectif principal, nous pouvons formuler les objectifs spécifiques suivants :

— Utiliser le procédé d'instrumentation pour collecter des données de performance, à partir des composants logiciels cibles.

— Modéliser les données de performance collectées, de manière à optimiser leur traitement par les analyses.

— Fournir des analyses pertinentes couvrant plusieurs aspects de performance des réseaux SDN et NFV.

— Développer des interfaces graphiques intuitives permettant de guider l'utilisateur dans l'identification des anomalies et des bogues de performance des logiciels réseau.

— Évaluer les surcoûts induits par la collecte et l'analyse des données, et proposer une approche pour les réduire.

1.4 Contributions

Conformément aux objectifs de recherche énoncés ci-dessus, cette thèse présente les contributions originales suivantes, dans le domaine d'étude des performances des applications SDN/NFV :

— Une instrumentation du code source des applications dont nous avons analysé les performances. De plus, une méthode permettant de réduire le surcoût de traçage en termes d'espace de stockage. Cette méthode consiste à lancer en parallèle deux sessions de traçage, et ne sauvegarder les évènements générés que lorsqu'un problème de performance est soupçonné d'avoir lieu.

— Des analyses pertinentes permettant d'évaluer l'efficacité des applications cibles, et de diagnostiquer leurs bogues de performance. Ces analyses reposent sur l'emploi d'algorithmes et de structures de données efficaces, pour calculer des métriques de

performance adaptées à partir d'une trace d'exécution.

— Des interfaces graphiques interactives développées dans un cadriciel d'analyse de performance à source ouverte. Ces interfaces présentent d'une façon intuitive les résultats des analyses effectuées, afin de faciliter à l'utilisateur la tâche d'identification et d'analyse des anomalies de performance.

1.5 Plan de la thèse

Dans le chapitre 2, nous présentons la revue de la littérature. Nous y introduisons les concepts nécessaires à la compréhension de ce manuscrit, et passons en revue les différents travaux de recherche en relation avec notre thématique de recherche. Dans le chapitre 3, nous détaillons la méthodologie que nous avons suivie pour atteindre nos objectifs de recherche. Nous présentons les articles que nous avons rédigés dans le cadre de cette recherche, dans les chapitres 4, 5, 6, et 7.

— L'article "Diagnostic and Troubleshooting of OpenFlow-enabled Switches using Kernel and Userspace Traces" au Chapitre 4, porte sur l'analyse de performance des commutateurs logiciels qui supportent le protocole OpenFlow. Cet article a été publié dans le journal *International Journal of Communication Systems*.

— L'article "Performance Analysis of DPDK-based Applications Through Tracing" au Chapitre 5, porte sur l'analyse de performance de DPDK, un cadriciel pour l'accélération des performances des applications réseau. Cet article a été soumis au journal *Journal of Parallel and Distributed Computing*.

— L'article "Performance Analysis of Virtual GPUs Through Host-based Tracing" au Chapitre 6, porte sur l'analyse de performance de GVT-g, la plateforme d'Intel pour la virtualisation processeurs graphiques. Cet article a été soumis au journal *International Journal of High Performance Computing Applications*.

— L'article "Virtual Networks Link-Layer Topologies Discovery through Host-based Tracing" au Chapitre 7, porte sur la découverte des liens logiques entre les fonctions réseau virtuelles. Cet article a été soumis à la conférence *International Conference on Computer Networks and Communications*.

Par la suite, nous discutons, dans le chapitre 8, les résultats obtenus et nous les contrastons avec les objectifs de recherche précédemment définis. Enfin, nous terminons cette thèse avec une conclusion, au chapitre 9.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente les concepts les plus importants qui relèvent de notre thématique de recherche. Il fait également le bilan sur les travaux de recherche en lien avec notre problématique. Son but est de situer notre travail de recherche dans son contexte afin de démontrer son originalité et sa pertinence.

2.1 Les réseaux SDN

Un réseau informatique est un ensemble d'équipements de différents types qui sont connectés les uns aux autres à l'aide d'un support de communication, dans le but d'échanger des informations. De nos jours, les réseaux sont très hétérogènes du fait qu'ils sont gérés par plusieurs opérateurs, et les équipements matériels qui les composent tels que les commutateurs, les routeurs, les *middleboxes* (p. ex., pare-feu, chiffrement du trafic, équilibrage de charge, etc.) et les hôtes, sont manufacturés par différents fabricants. Souvent, ces équipements sont pilotés par des logiciels propriétaires et fermés qui implémentent des algorithmes complexes et des protocoles distribués.

Dans les réseaux traditionnels, tels que les réseaux IP classiques, les équipements matériels sont caractérisés par un couplage fort entre le plan de contrôle (qui indique comment transférer, filtrer ou modifier le trafic) et le plan de données (qui s'occupe de l'expédition de données). Par conséquent, pour configurer le réseau, les opérateurs sont la plupart du temps contraints à configurer le fonctionnement de chaque équipement séparément, à travers une interface de commande (textuelle ou graphique). Il est à noter que ces interfaces de commandes, qui sont souvent de bas niveau, varient selon le constructeur et le type d'équipement. En effet, la configuration manuelle et individuelle des équipements réseau complique énormément la gestion de ces réseaux, et rend très difficile l'évolution de leurs architectures, matériel et protocoles [3]. Par exemple, plusieurs chercheurs attribuent le taux faible d'adoption du protocole IPv6 à la rigidité de l'architecture du réseau Internet [4].

Le paradigme *Software-Defined Networking* (SDN) a été proposé dans ce contexte pour résoudre le problème de rigidité des réseaux traditionnels. Ce paradigme désigne une nouvelle architecture qui vise à simplifier la gestion des réseaux et la rendre plus flexible. Cette architecture est basée sur une séparation nette entre le plan de contrôle et le plan de données au niveau du matériel. En effet, le plan de contrôle est détaché du matériel et implanté dans une entité logiquement centralisée appelée contrôleur SDN. Cette entité est programmable,

ce qui veut dire qu'elle présente une API ouverte permettant aux applications de l'opérateur d'automatiser le contrôle, la configuration, et le suivi de l'infrastructure réseau sous-jacente. Ainsi, il est plus facile de développer des applications réseau (moyennant des langages de programmation de haut niveau) qui permettent de définir le comportement du réseau que de configurer chaque équipement séparément. En rendant le réseau programmable, les opérateurs peuvent gérer les équipements de l'infrastructure d'une façon cohérente, indépendamment de la taille du réseau et de la complexité de sa topologie.

2.1.1 Architecture de SDN

L'architecture SDN repose sur l'existence d'une entité logiquement centralisée qui détermine le comportement de tous les équipements de commutation de paquets. En effet, cette architecture se distingue de toute autre architecture réseau par ces quatre points [3] :

1. Le plan de contrôle est complètement dissocié du plan de données. Selon cette architecture, les équipements réseau sont de simples équipements de transfert de données, qui sont contrôlés à distance par une entité logiquement centrale. Cette dernière est communément appelée contrôleur SDN.
2. Toute l'intelligence du réseau est placée au niveau du contrôleur SDN. Ce dernier peut être vu comme un système d'exploitation réseau qui gère les équipements de transfert de données. Le contrôleur SDN et les équipements du plan de données maintiennent des liens de communication à travers une interface ouverte et standard, connue sous le nom d'API sud (*Southbound API*).
3. Le réseau est programmable à travers les applications qui s'exécutent au-dessus du contrôleur SDN. Ce dernier offre aux applications les ressources et les abstractions nécessaires pour leur permettre de configurer à distance les équipements du plan de données. De ce fait, le contrôleur expose une API connue sous le nom d'API nord (*Northbound API*). Cette API est exploitée par les applications réseau pour expliciter leurs besoins.
4. La transmission des paquets au niveau du plan de données se fait par flux. Dans la terminologie de SDN, un flux est une séquence des paquets entre une entité source et une entité destination. Ces paquets satisfont un ou plusieurs critères qui sont souvent définis sur la base de leurs champs d'entête. Les équipements du plan de données appliquent toujours le même traitement sur les paquets appartenant à un même flux.

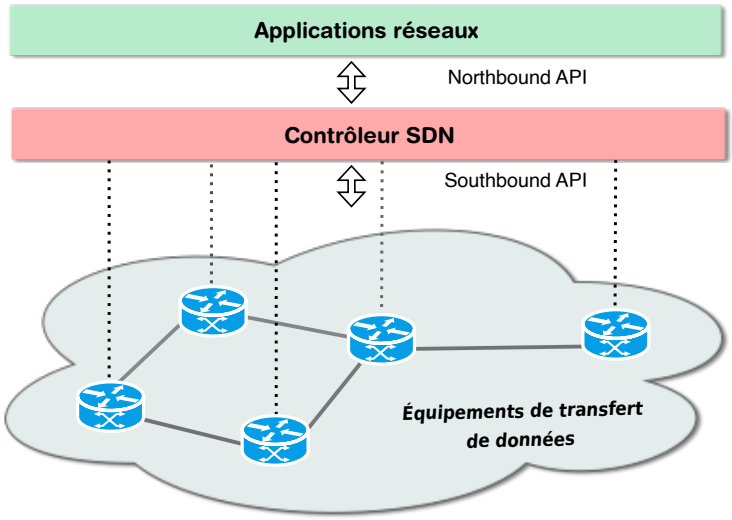

Figure 2.1 Architecture simplifiée de SDN

Le plan de données

Le plan de données représente l'infrastructure physique du SDN. Cette dernière est essentiellement composée par des équipements de transfert de données interconnectés entre eux à travers un canal de communication (filaire ou sans-fil). Un équipement du plan de données peut être un élément matériel, tel que les commutateurs et les routeurs classiques, ou un élément logiciel qui implémente des fonctionnalités de transfert de données (p. ex., Open vSwich [5]). À la différence des réseaux conventionnels, les équipements de transfert de données dans SDN ne disposent d'aucune fonctionnalité de contrôle. Par ailleurs, ils n'intègrent pas des programmes capables de prendre, d'une façon autonome, des décisions concernant les actions à appliquer sur le trafic.

En effet, comme nous l'avons mentionné dans les paragraphes précédents, tous les modules de contrôle qui implémentent l'intelligence du réseau sont placés dans le contrôleur SDN. Ce dernier utilise les interfaces sud pour configurer les équipements du plan de données et communiquer avec eux. Étant donné que ces interfaces sont ouvertes, l'interopérabilité entre le plan de contrôle et le plan de données est assurée indépendamment des choix de conception et d'implémentation de chacun. De nos jours, les interfaces sud sont implémentées par plusieurs protocoles comme OpenFlow (OF) [6], qui est une norme très répandue.

Les équipements du plan de données qui prennent en charge le protocole OpenFlow sont communément appelés commutateurs OpenFlow. Le contrôleur SDN utilise OpenFlow pour installer des règles de flux dans les commutateurs, afin de leur permettre de gérer le trafic entrant. Une règle de flux peut être décomposée en trois parties essentielles :

1. Un filtre défini par un ensemble des valeurs représentant les conditions à satisfaire par un paquet pour qu'il soit traité par la règle. Ces valeurs correspondent à des champs spécifiques dans les en-têtes du paquet (p. ex., IP, TCP, UDP, et MPLS) et à certaines métadonnées telles que le port d'entrée, par exemple.
2. Une liste d'actions à appliquer sur les paquets vérifiant les conditions indiquées dans le filtre de la règle. Les principales actions sont le transfert du paquet par un port spécifique, le rejet du paquet, la modification des champs d'en-tête du paquet, le transfert du paquet vers le contrôleur, l'ajout d'un en-tête au paquet, etc.
3. Les compteurs du flux qui maintiennent des statistiques sur les paquets traités par la règle.

Une règle de flux peut aussi inclure d'autres informations comme la priorité et la durée de vie de la règle. La priorité indique quelle règle doit être appliquée par le commutateur dans le cas où un paquet entrant corresponde à plusieurs règles de flux. Quant à la durée de vie, elle définit le temps après lequel la règle de flux expire et devient obsolète.

Il est à noter qu'un commutateur OpenFlow organise les règles de flux dans une ou plusieurs tables, appelées tables de flux. En effet, à la réception d'un paquet, le commutateur parcourt l'ensemble des règles sauvegardées dans ses tables de flux et tente de le correspondre avec l'une d'entre elles. En cas de succès, il applique les actions de la règle correspondante sur le paquet, et met à jour les compteurs de flux de cette règle par la suite. Le commutateur rejette le paquet si aucune de ses règles ne lui correspond et qu'une règle par défaut n'était pas définie. Les tables de flux peuvent être enchaînées afin de répartir sur plusieurs étapes le traitement à appliquer sur les flux (voir figure 2.2).

Le plan de contrôle

Le plan de contrôle est l'entité qui commande, à travers un protocole de communication, les équipements du plan de données et qui définit la manière par laquelle ils gèrent le trafic. Principalement, cette entité est formée par les contrôleurs SDN et les applications qui s'exécutent dessus. Un contrôleur SDN fournit plusieurs abstractions et services permettant l'implémentation des diverses fonctions réseau (p. ex., routage, pare-feu, équilibrage de charge, surveillance, etc.). En effet, le contrôleur se sert de ses interfaces nord pour faire abstraction de la complexité et l'hétérogénéité des équipements du plan de données, ainsi que des détails de leur fonctionnement. Il offre aussi de nombreux services tels que le service de découverte de topologie qui permet aux applications d'avoir une vue globale de l'infrastructure SDN. Ce service est essentiel pour plusieurs types d'application tels que les applications de routage et de détection d'intrusion.

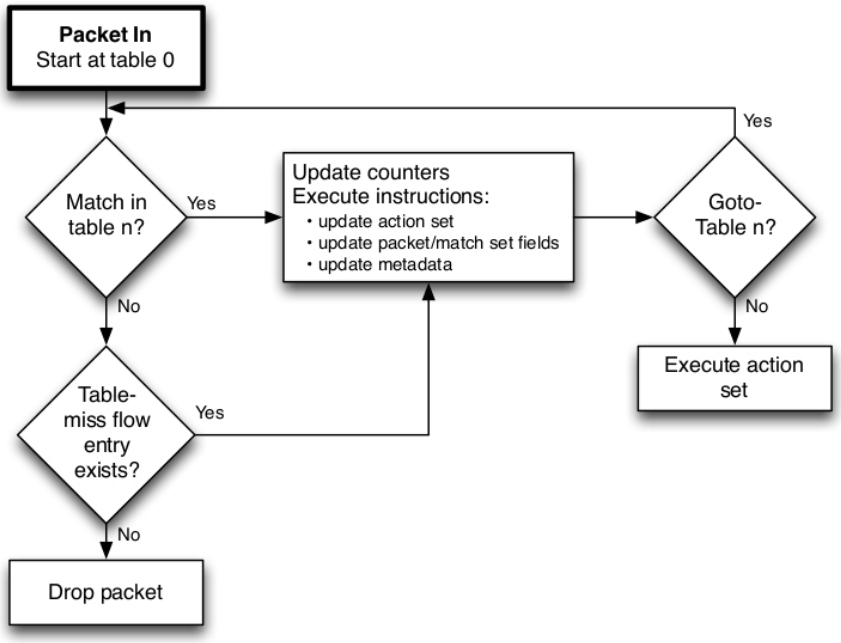

Figure 2.2 Diagramme simplifié illustrant le traitement d'un paquet par un commutateur OpenFlow [7]

Du point de vue conceptuel, la plupart des contrôleurs SDN sont conçus selon une architecture physiquement centralisée. Ces contrôleurs profitent du modèle multifil pour tirer avantage des ressources de calcul des machines multicœurs et augmenter ainsi leurs performances. À titre d'exemples, Floodlight [8], NOX [9] et Beacon [10] sont des contrôleurs bien connus qui font partie de cette catégorie.

La centralisation des fonctions du plan de contrôle dans un seul contrôleur SDN présente deux inconvénients majeurs. D'abord, une telle conception pose un problème d'extensibilité, étant donné qu'un seul contrôleur peut être incapable de gérer la charge d'un réseau formé par un grand nombre d'équipements de transfert de données. Le contrôleur dans ce cas-ci crée un goulot d'étranglement qui pourrait impacter négativement les performances du réseau. De même, cette conception pose un problème de sécurité du fait qu'un réseau SDN qui est géré par un seul contrôleur serait une cible facile aux attaques, telles que celles de type déni de services. En effet, le contrôleur constitue un point de défaillance unique qui menace la disponibilité des services du réseau. Il suffit de le surcharger (avec des requêtes superflues par exemple) ou de compromettre sa sécurité pour entraîner une dégradation considérable des performances de tout le réseau ou mettre à l'échec ses opérations.

Pour résoudre les problèmes de passage à l'échelle, de fiabilité et de disponibilité, des nombreux contrôleurs sont conçus selon une architecture physiquement distribuée. Il existe plu-

sieurs approches pour mettre en œuvre une telle architecture [11], telles que la répartition de la charge de contrôle sur plusieurs contrôleurs, la mise en place des copies passives du contrôleur qui deviennent actives quand le contrôleur principal tombe en panne, la distribution horizontale des fonctions de contrôle sur une grappe de contrôleurs, etc. Souvent, les architectures multicontrôleurs nécessitent des APIs supplémentaires pour assurer la communication entre plusieurs contrôleurs (les APIs Eastbound et Westbound). Les contrôleurs Orion [12] et ONOS [13] sont des exemples des contrôleurs distribués.

Les APIs nord et sud

Les interfaces nord et sud représentent des abstractions clés de l'écosystème SDN. En effet, l'interface nord est une interface logicielle qui est définie par un ensemble d'API ouverts. Le contrôleur SDN fournit cette interface pour permettre aux développeurs de logiciels d'implémenter des applications de gestion, sans se soucier des détails de bas niveau de l'infrastructure physique. Typiquement, le développement de ces applications se fait à l'aide des langages de programmation de haut niveau.

Jusqu'à présent, il n'existe aucun standard ouvert qui soit largement accepté pour l'interface nord [3]. Tous les contrôleurs SDN existants, tels que Floodlight et NOX, définissent et offrent leurs propres implémentations de l'interface nord. L'existence des interfaces nord ouvertes et standardisées assure la portabilité des applications, ainsi que l'interopérabilité entre les plateformes de contrôle.

Contrairement à l'interface nord, l'interface sud dispose d'un standard largement accepté qui est le protocole OpenFlow. En effet, l'interface sud définit le protocole de communication par lequel le plan de contrôle interagit avec le plan de données. Elle définit également le jeu d'instructions utilisé par les équipements de transfert de paquets pour gérer le trafic réseau. Pour implémenter l'interface sud, de nombreuses APIs ont été proposées, tels que le POF (Protocol-Oblivious Forwarding) [14], OVSDB (Open vSwitch DataBase) [15], et OpenFlow [6].

2.1.2 Performances des réseaux définis par le logiciel

Plusieurs outils et travaux de recherche ont été proposés dans la littérature pour évaluer et analyser les performances des réseaux définis par le logiciel. Ces travaux peuvent être classifiés selon plusieurs critères tels que le plan des éléments cibles (plan de données ou de contrôle), les techniques utilisées (p. ex., étalonnage, profilage, analyse de trafic, etc.), et les métriques calculées.

En effet, il existe de nombreux outils qui utilisent la technique d'étalonnage pour émuler la charge des réseaux à grande échelle et évaluer les performances des contrôleurs et des commutateurs SDN [16]. Bien que cette technique soit utile pour mesurer et comparer les performances des différents composants de SDN, elle ne permet pas de diagnostiquer les causes des problèmes de performance.

À titre d'exemples, nous citons les outils CBench [17], OFCBenchmark [18], OFCProbe [19] et OFCP [20] qui sont très utilisés dans l'évaluation des divers aspects de performance des contrôleurs SDN comme l'évolutivité, la robustesse et l'efficacité. La majorité de ces outils calculent des métriques de performance comme le débit et la latence du contrôleur SDN, pour évaluer l'efficacité de son modèle d'enfilage (threading). Le fonctionnement de OFCP, par exemple, se base sur l'utilisation d'un commutateur virtuel pour envoyer un grand nombre des messages de type *packet-in*, recevoir les réponses du contrôleur, et calculer sa latence en se basant sur le temps d'aller-retour. De même, il existe des outils d'étalonnage pour évaluer les performances des commutateurs SDN comme OFLOPS [21], OFLOPS-turbo [22] et ORTF [23].

D'autres outils exploitent la technique de profilage pour détecter et diagnostiquer divers types de problèmes de performance. Le profilage est une technique d'ingénierie logicielle qui consiste à collecter des données de performance lors de l'exécution d'un programme, dans le but d'optimiser son code source et éliminer ses goulots d'étranglement. En se basant sur cette technique, Kang et al. [24] ont proposé SPIRIT, un outil permettant d'analyser les performances des applications SDN. Cet outil se connecte à l'interface nord du contrôleur SDN pour collecter les données de profilage, en plus d'autres données comme les latences et l'utilisation CPU des applications SDN supervisées. Les données collectées sont analysées afin de reconstruire les flots d'exécution de ces applications et déterminer leurs chemins critiques ainsi que leurs fonctions les plus utilisées (hotspots). Les analyses effectuées par SPIRIT permettent de découvrir des problèmes de performance dans les applications SDN comme certains goulots d'étranglement (p. ex., accès en concurrence aux ressources du contrôleur). L'inconvénient majeur de cet outil est que les analyses qu'il fournit sont seulement limitées au plan de contrôle. Aussi, cet outil génère un surcoût considérable qui s'élève à 1200%, ce qui entrave son utilisation dans un environnement de production.

Handigol et al. [25] ont proposé *nprof*, un outil de profilage réseau de la plateforme de supervision NetSight. Cet outil combine les informations sur la topologie du SDN ainsi que les historiques de paquets pour mesurer la charge sur les liens réseau et évaluer les décisions de routage au niveau des équipements réseau. Les administrateurs peuvent utiliser *nprof* pour équilibrer la charge entre les ressources du réseau et optimiser ses performances.

Les protocoles de surveillance réseau traditionnels peuvent être utilisés pour mesurer les performances des commutateurs SDN. La plupart des commutateurs OF, tels que les commutateurs conventionnels, prennent en charge ces protocoles. Dans les paragraphes suivants, nous présentons quelques-uns de ces protocoles.

Le SNMP (Simple Network Management Protocol) [26] est un protocole élaboré par le groupe de travail IETF (Internet Engineering Task Force) pour la gestion et la supervision des réseaux IP. Le fonctionnement de ce protocole se base sur l'interrogation périodique, par une application de supervision, des agents SNMP installés sur les unités supervisées (p. ex., commutateur, routeurs, ponts, etc.). Le comportement de chaque unité supervisée est défini par des attributs sauvegardés dans une base d'information de gestion appelée MIB (Management Information Base). En effet, le protocole SNMP permet à l'application de supervision de lire et d'écrire dans les MIBs des unités supervisées, ce qui permet de les gérer, mais aussi de collecter des mesures de performance de haut niveau telles que la charge du processeur et le débit de transmission/réception par interface réseau.

Netflow [27] est un protocole propriétaire développé par Cisco pour collecter des statistiques sur les flux IP depuis les équipements de transfert des paquets. Ce protocole représente un standard en matière de surveillance des réseaux et d'analyse du trafic IP. Netflow présente une architecture simple qui comprend essentiellement trois types de composants : les Exportateurs, les Collecteurs et les Analyseurs. Un Exportateur est un agent logiciel qui s'exécute sur un équipement réseau pour établir des statistiques sur les flux IP (p. ex., nombre de paquets et d'octets par flux) et les transmettre périodiquement à un Collecteur. Ce dernier prend en charge le prétraitement et la sauvegarde des données transmises par un groupe de composants exportateurs. Quant aux composants de type Analyseur, ils prennent en charge l'analyse des données sauvegardées et la dérivation des métriques de performance du réseau.

Le sFlow (sampled Flow) [28] est un autre protocole de surveillance réseau qui est largement pris en charge dans les infrastructures réseau. Le fonctionnement de sFlow se base sur un échantillonnage statistique des paquets, ce qui permet de réduire le surcoût de traitement au niveau des équipements réseau. En effet, l'agent sFlow effectue une copie tronquée de chaque $n^{ième}$ paquet, reçu ou transmis par l'équipement réseau, et l'envoie à un nœud collecteur. Le choix du taux d'échantillonnage dépend principalement de la précision de mesure visée et de la charge à vouloir imposer sur les périphériques et le réseau. L'agent sFlow exporte aussi périodiquement les compteurs des interfaces réseau de l'équipement vers le nœud collecteur.

L'état de l'art reporte plusieurs travaux qui exploitent les protocoles susmentionnés pour détecter des problèmes de performance dans SDN [29–32]. Afaq et al. [29], par exemple, ont proposé un cadriciel de surveillance pour SDN qui utilise le protocole sFlow pour détecter les

flux gourmands en bande passante (connus sous le nom de flux éléphants). L'ensemble des flux à surveiller est déterminé par le contrôleur SDN. En effet, le cadriciel collecte en temps réel des statistiques sur les flux et identifie ceux qui sont gourmands en bande passante, en se basant sur des seuils prédéterminés. Pour réduire le surcoût de surveillance, les auteurs proposent simplement d'ajuster la fréquence d'échantillonnage de sFlow.

Van Adrichem et al. [31] ont proposé OpenNetMon, un outil qui améliore l'ingénierie du trafic par la surveillance de plusieurs paramètres de qualité de service (Quality of Service ou QoS) du réseau. Cet outil exploite le protocole OF pour extraire, à une fréquence adaptative, les compteurs de flux à partir des commutateurs OF. Il dérive ensuite des mesures de performance comme le débit de RX/TX des interfaces réseau, et le taux de perte de paquets. L'outil proposé est incapable de diagnostiquer des problèmes de surveillance, étant donné la granularité des données collectées.

Les auteurs dans [33] proposent une approche pour détecter certaines anomalies qui peuvent affecter le modèle du trafic dans SDN comme les *heavy hitters* et les *superspreaders*. Selon cette approche, les équipements de transfert de données utilisent une structure de données hiérarchique pour affiner de manière itérative les préfixes IP et déterminer les nœuds qui sont responsables de l'anomalie. Pour éviter les communications contrôleur-plan de données inutiles, un mécanisme *push* est utilisé pour envoyer des notifications au contrôleur, lorsque des conditions spécifiques sont remplies. Bien que cette approche se concentre sur la détection des anomalies du trafic réseau, notre proposition vise plutôt le diagnostic des problèmes de performance de SDN.

Phan et al. [34] ont proposé SDN-Mon, un cadriciel pour la supervision des flux dans SDN avec une granularité fine. Ce cadriciel comprend deux modules : un agent qui s'exécute sur les commutateurs SDN et une application de contrôle qui s'exécute sur la machine du contrôleur. Le premier module implémente les fonctionnalités de surveillance et traite les messages qui sont envoyés par le module de contrôle. Le deuxième module détermine les flux à superviser dans le réseau selon les besoins des applications. Le cadriciel proposé minimise l'impact de la supervision sur les performances du commutateur à travers le maintien des tables contenant les statistiques des flux séparées des tables OpenFlow de transfert de paquets. Les auteurs proposent aussi dans [35] d'étendre les fonctionnalités de SDN-Mon pour lui permettre de prendre en charge la surveillance distribuée. En effet, cette extension introduit un module additionnel qui est responsable de l'affectation dynamique des tâches de surveillance aux agents et de l'équilibrage de la charge entre les commutateurs.

2.2 Virtualisation des fonctions réseau

Le concept de la virtualisation des fonctions réseau (Network Functions Virtualization ou VNF) repose sur la dissociation des fonctions réseau (transfert, filtrage, classification, et transformation du trafic) du matériel dédié sur lequel elles s'exécutaient. L'objectif de cette dissociation est de pouvoir utiliser du matériel générique et standard (Commercial Off-The-Shelf ou COTS) dans l'approvisionnement de ces fonctions. En effet, l'approche traditionnelle qui s'appuie sur l'utilisation du matériel spécialisé s'est montrée inefficace, surtout face à l'augmentation rapide du trafic Internet, en volume et en diversité [36].

De manière générale, les caractéristiques inhérentes au matériel dédié et propriétaire entravent la flexibilité et l'évolutivité du réseau, et n'assurent pas le déploiement rapide et à la demande de ses services. Puisque ce type de matériel est assez cher et nécessite un certain niveau d'expertise pour pouvoir le configurer, les coûts d'acquisition et de maintenance sont généralement onéreux. De même, l'implémentation des fonctions réseau au niveau matériel rend très difficile le déploiement des nouvelles fonctions sur le même équipement. Au surplus, un matériel dédié pose souvent un problème d'extensibilité, vu qu'il ne permet pas aux services du réseau de s'adapter à une demande croissante.

La NFV est une architecture réseau évolutive qui est axée sur l'exploitation des fonctions réseau sous la forme de modules logiciels, plutôt que d'utiliser un matériel spécifique qui est cher et peu flexible. Ces fonctions sont habituellement déployées dans des machines virtuelles ou des conteneurs qui sont hébergés par des serveurs COTS. Elles sont, de ce fait, appelées fonctions réseau virtuelles (Virtual Network Functions ou VNF).

Cette nouvelle approche réseau est basée sur deux piliers importants, qui sont la dissociation du cycle de développement du matériel de celui des fonctions réseau et l'exploitation des technologies de virtualisation. Le premier pilier permet aux opérateurs de service de développer et déployer de nouveaux services et des applications réseau à la demande, sans pourtant avoir besoin des ressources matérielles supplémentaires. Le deuxième pilier permet d'optimiser l'utilisation des ressources réseau physiques et d'améliorer l'évolutivité et l'agilité du réseau. En résumé, la NFV simplifie énormément l'approvisionnement des services réseau, parce qu'elle s'appuie sur l'utilisation des logiciels pour automatiser le déploiement, la surveillance, et la mise à l'échelle des fonctions réseau.

2.2.1 Architecture des plateformes NFV

En novembre 2012, L'ETSI (European Telecommunications Standards Institute) a créé le premier groupe de travail ayant pour objectif la standardisation des technologies régissant le

développement de VNF. Les travaux de ce groupe de travail ont abouti à la publication des spécifications d'un cadre commun permettant la mise en œuvre de la VNF. Ces spécifications proposent une architecture de référence qui inclut trois composants essentiels qui sont : le plan de service, l'infrastructure NFV, et le plan MANO (Management, Automation, and Network Orchestration). Nous présentons ces composants dans les paragraphes suivants.

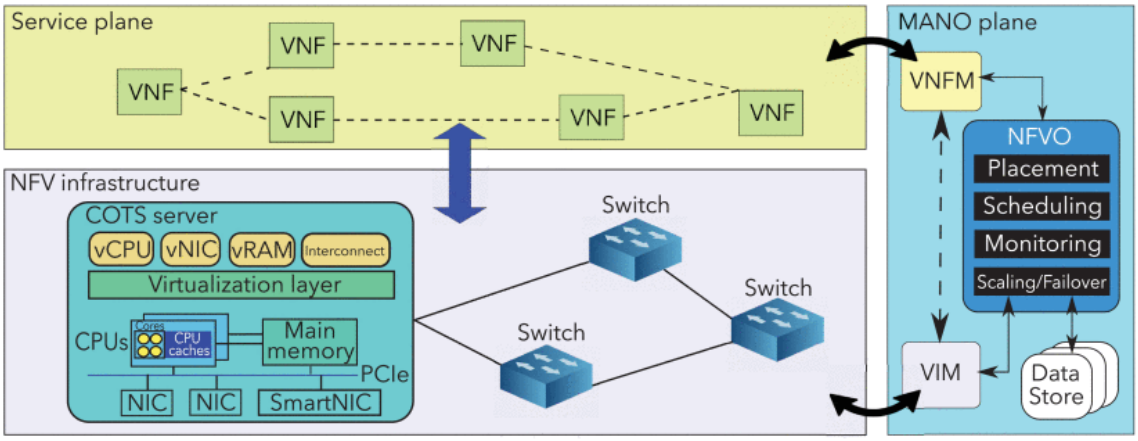

Figure 2.3 Architecture simplifiée d'une plateforme NFV [37]

Plan de service

Le plan de service est essentiellement formé par une collection des VNF. Une VNF est une fonction qui effectue un certain traitement sur du trafic réseau, et qui est réduite à son simple aspect logiciel. Elle peut être hébergée et exécutée dans un nœud de l'infrastructure NFV sous la forme d'une application logicielle, d'une machine virtuelle, ou d'un conteneur (p. ex., Docker).

Une VNF peut être agencée avec d'autres VNF pour former une chaîne de services dans le but d'accomplir une fonction réseau particulière. La distribution des VNF sur les nœuds de l'infrastructure est extrêmement flexible. Il est possible, par exemple, de déployer une VNF ou toute une chaîne de service sur une seule machine virtuelle. Il est également courant de diviser une VNF en des unités de traitement plus fines, et les déployer sur plusieurs machines virtuelles.

Infrastructure NFV

L'infrastructure NFV est constituée de l'ensemble des équipements matériels et logiciels nécessaire pour l'exécution des VNF. L'équipement matériel inclut principalement les serveurs

COTS, mais aussi du matériel réseau tel que les commutateurs et les routeurs. Ces serveurs sont des plateformes de calcul à usage général, qui intègrent des processeurs centraux multi-cœurs, des interfaces réseau, et possiblement différents types d'accélérateurs matériels pour augmenter leur capacité de traitement (p. ex., les processeurs graphiques).

Sur les serveurs COTS sont installés plusieurs logiciels qui collaborent et interagissent afin de créer un environnement d'exécution pour les VNF. À titre d'exemple de ces logiciels, on peut citer ceux qui mettent en œuvre la couche de virtualisation, tels que le système d'exploitation, l'hyperviseur des machines virtuelles, et le moteur d'exécution des conteneurs. Les logiciels qui instancient des commutateurs virtuels (exemples : Open vSwitch et VPP) pour interconnecter les machines virtuelles sont aussi requis.

Plan MANO

Le plan MANO (Management, Automation, and Network Orchestration) est le système en charge de la gestion des VNF, des services réseau, et de l'infrastructure physique et virtuelle sous-jacente. Typiquement, ce système est composé de trois modules : le VIM (Virtualized Infrastructure Manager), le NFVO (NFV Orchestrator), et le VNFM (VNF Manager) [37].

Le VIM a pour rôle de gérer les ressources de calcul, de stockage, et de réseau de l'infrastructure NFV. Il est aussi responsable de l'optimisation de l'utilisation des ressources virtuelles à travers l'orchestration de leur allocation, mise à jour, et libération. Le VIM gère en outre la correspondance entre les ressources virtuelles et physiques.

Le NFVO est le composant le plus important de toute la plateforme NFV. En effet, sa mission consiste à assurer le bon fonctionnement des services réseau, et ce à travers la mise en œuvre des fonctions suivantes :

- La mise en place d'une stratégie de placement permettant de déterminer les nœuds sur lesquels les VNF seront déployées, ainsi que l'ordre de leur enchaînement.
- Ordonnancer les opérations d'allocation des ressources de l'infrastructure NFV (à travers le module VIM) qui sont requises pour l'exécution des VNF.
- La collecte des statistiques depuis le plan de service et l'infrastructure NFV (p. ex., état de la consommation de ressources virtuelles et physiques), et la mise de ces statistiques à disposition des autres modules MANO.
- La mise à l'échelle des VNF et des chaînes de service pour s'adapter à une fluctuation du trafic ou à l'échec de certaines VNF.

De son côté, le VNFM s'occupe de la gestion du cycle de vie des VNF et des chaînes de service. Il est ainsi responsable de leur instanciation, leur mise à jour, leur passage à l'échelle, et leur terminaison.

2.2.2 Approches d'accélération logicielle

La substitution des fonctions réseau codées en dur avec leurs équivalents logiciels a motivé la recherche des techniques efficaces permettant l'accélération des fonctions réseau logicielles [38]. L'objectif de ces techniques est de réduire l'écart entre les performances des VNF et celles du matériel dédié. Les techniques d'accélération développées incluent plusieurs techniques d'optimisation du code, ainsi que des conceptions innovantes pour le logiciel réseau (p. ex., applications, pilotes, pile protocolaire, etc.).

Au niveau des pilotes des cartes réseau, la migration vers un mode de fonctionnement basé sur la scrutation pour traiter les paquets entrants, est un exemple de ces techniques d'accélération logicielle. En effet, le mode de fonctionnement classique qui est basé sur les interruptions présente l'inconvénient d'avoir un surcoût considérable. Ce surcoût est essentiellement dû à la fréquence élevée des changements du contexte du processeur central, lorsque les charges réseau sont importantes.

Dans un mode de fonctionnement basé sur la scrutation, le processeur central vérifie en permanence la présence des paquets entrant dans les files d'attente associées aux cartes réseau et n'attend pas à ce que ces dernières soulèvent des interruptions. Ce mode permet de réduire l'attente des paquets dans les files d'attente et augmente le débit réseau. Toutefois, ce mode nécessite un taux d'utilisation du processeur central estimé à 100%, même lorsqu'il n'y aucun paquet entrant.

La technique appelée copie-zéro permet au processeur central d'éviter la copie des données d'une mémoire tampon à une autre. Cette technique permet d'éliminer le coût associé à la copie des paquets reçus dans l'espace d'adressage de l'application destinataire. La mise en œuvre de cette technique nécessite l'emploi du mécanisme DMA (Direct Memory Access) pour sauvegarder les paquets reçus directement dans l'espace mémoire de l'application. Grâce à cette technique, le noyau et l'application échangent seulement des pointeurs vers les paquets, au lieu de faire des copies entières d'eux.

Une autre technique consiste à regrouper les opérations d'E/S dans des lots. En effet, la réception ou la transmission de paquets en rafale, au lieu d'un seul à la fois, permet de réduire l'utilisation du processeur central et d'amortir le coût de transfert mémoire. Une technique similaire, appelée traitement en lot, consiste à regrouper les paquets en des lots pour augmenter l'efficacité de leur traitement par les VNF. Cette technique optimise l'utilisation du cache d'instructions du processeur central, ainsi que ses pipelines du traitement.

Plusieurs études dans la littérature soulignent les défauts de la pile réseau du noyau Linux et démontrent son incapacité à gérer les performances élevées des cartes réseau modernes. Un

des défauts majeurs associés à cette pile se présente au niveau des changements du contexte fréquents, qui sont déclenchés par les interruptions et les appels aux primitives de gestion des sockets (exemples : *bind*, *accept*, *read*, *write*, et *close*). La technique d'accélération appelée contournement du noyau (*kernel bypass*) a été inventée dans ce contexte pour contourner les goulots d'étranglement présents dans la pile réseau du noyau. Cette technique consiste à contourner le noyau et à traiter les paquets à l'aide d'une pile réseau personnalisée.

Ces dernières années, différentes solutions d'accélération des fonctions réseau ont vu le jour. Certaines de ces solutions, telles que XDP (eXpress Data Path) [39], ont été proposées pour améliorer les performances de la pile réseau du noyau. D'autres solutions, en revanche, ont été proposées pour contourner cette pile réseau (p. ex., DPDK [1] et Netmap [40]). De toute manière, la plupart de solutions proposées implémentent plusieurs techniques, parmi celles que nous avons évoquées dans cette section. Par exemple, les solutions DPDK et Netmap implémentent le regroupement des opérations d'E/S, le traitement en lot, et la copie-zéro.

2.2.3 Outils d'analyse de performance

Par comparaison avec les fonctions réseau physiques, les VNF sont plus susceptibles à des bogues de performance puisqu'elles s'appuient sur beaucoup plus de composants logiciels indépendants (hyperviseurs, conteneurs/machines virtuelles, commutateurs virtuels, etc.). De plus, plusieurs recherches dans la littérature affirment que ce type de bogues peut considérablement réduire l'efficacité des VNF. Par exemple, les auteurs dans [41] ont étudié les problèmes de performance causés par la collocation des VNFs dans une même machine. Ils notent que la compétition effectuée par ces fonctions sur les ressources pourrait engendrer une dégradation de performance qui peut atteindre 50.3%.

Pour déceler et diagnostiquer les problèmes de performance des VNF, plusieurs travaux de recherche ont été proposés [42–47]. Ces travaux varient en fonction des techniques qu'ils utilisent et de leur exhaustivité. Par exemple, pour diagnostiquer des problèmes de performance dans les infrastructures NFV, les outils *NFV-Inspector* [45] et *Probius* [46] utilisent le profilage alors que d'autres, comme *Env2Vec* [47], exploitent des techniques d'apprentissage machine.

Probius est un système d'analyse de performance qui se base sur l'extraction des caractéristiques de performance appropriées à chaque élément de l'architecture NFV. Ce système collecte les données de performance relatives aux éléments NFV en se basant sur plusieurs méthodes comme le traçage et les APIs de virtualisation. Pour découvrir les raisons fondamentales des bogues de performance, il analyse le comportement des VNF dans les chaînes de service à l'aide d'une technique d'analyse basée sur la régression statistique. L'avantage prin-

cipal de ce système est son exhaustivité : il peut diagnostiquer des problèmes de performance à tous les niveaux de la pile de VNF.

Env2Vec est une architecture d'apprentissage machine permettant d'automatiser les tests dans le flux de production des VNF. Le fonctionnement de cette architecture se base sur l'identification d'une dégradation de performance au niveau des VNF pour détecter des anomalies et des bogues. Pour repérer un problème de performance dans les VNF, *Env2Vec* se base sur la caractérisation de leur consommation de ressources à l'aide des réseaux de neurones. La principale limite de cette architecture est que, comme la plupart des solutions d'apprentissage machine, elle nécessite une phase d'apprentissage coûteuse en termes de temps et de ressources.

NFVPerf [42] est un outil capable d'identifier les goulots d'étranglement dans les infrastructures de NVF. Cet outil évalue d'une façon continue les charges des éléments de l'infrastructure à travers le calcul de certaines métriques de performance, telles que le débit et la latence de transfert des paquets. En effet, un élément est considéré contenir un goulot d'étranglement si sa charge actuelle dépasse d'une certaine valeur sa charge calculée précédemment. L'outil proposé présente une limite majeure qui est l'incapacité de diagnostiquer les goulots d'étranglement qui se trouvent à l'intérieur des VNF.

Wu et al. [43] ont proposé *PerfSight*, un outil pour diagnostiquer les problèmes de performance qui peuvent affecter les plans de données logiciels dans les infrastructures NFV. Le fonctionnement de cet outil se base sur l'analyse des données de bas niveau collectées depuis les éléments qui se trouvent dans le chemin de données des paquets (les pilotes des interfaces réseau physiques et virtuelles, les commutateurs virtuels, etc.). *PerfSight* se base sur des métriques de performance, comme le taux d'abandon des paquets et le débit, pour repérer des goulots d'étranglement au niveau des éléments du réseau. La principale limite de cet outil est qu'il propose des analyses seulement pour les chemins de données situés au niveau du noyau.

Moradi et al. [44] ont proposé *ConMon*, une plateforme distribuée pour surveiller les performances des VNF déployées dans des conteneurs. Le fonctionnement de cette plateforme se base sur l'instanciation et le déploiement à la demande des conteneurs exécutant des fonctions de surveillance. Ces conteneurs spéciaux utilisent les APIs fournies par le moteur d'exécution de l'hyperviseur et l'écoute du trafic réseau pour collecter certains indicateurs de performance depuis les conteneurs adjacents (p. ex., taux d'utilisation du processeur central/mémoire, débits et latences de transfert de paquets, etc.). La principale limite de la plateforme proposée est son incapacité à découvrir les causes fondamentales des problèmes de performance détectés.

VTune Amplifier est un outil de profilage proposé par Intel pour analyser les performances

des applications multifils. Cet outil offre certaines analyses permettant de diagnostiquer les goulots d'étranglement des applications qui nécessitent des opérations d'E/S intensives. À titre d'exemple, VTune fournit des analyses permettant de calculer certaines métriques de performance pour les applications réseau qui se basent sur DPDK. Malheureusement, ses analyses sont très limitées et ne couvrent que quelques bibliothèques DPDK. De plus, VTune est un outil commercial à source fermée.

2.3 Les coprocesseurs réseau

Étant donné que, souvent, les fonctions réseau incluent des opérations d'E/S intenses, les plateformes de calcul à usage général, qui sont essentiellement basées sur des processeurs centraux multicœurs, ne sont pas conçues pour les exécuter efficacement. Les lacunes de ces plateformes ont motivé la fabrication de plusieurs types de co-processeurs, et du matériel dédié, qui peuvent augmenter les performances des processeurs centraux et accélérer ainsi le traitement des paquets réseau.

En effet, parmi les équipements matériels dédiés qui sont couramment exploités dans la réseautique, on trouve les ASIC (Application Specific Integrated Circuits), les FPGA (Field Programmable Gate Arrays) et les processeurs graphiques (Graphics Processing Units ou GPU). Bien que ces équipements présentent chacun des caractéristiques très différentes, ils sont tous capables d'améliorer les performances des processeurs centraux en déchargeant certaines tâches des fonctions réseau, comme la commutation des paquets, le chiffrement des données, et la recherche dans les tables de flux [36, 48]. Les processeurs de type ASIC et FPGA présentent l'avantage d'être capables de répondre aux besoins de performance des applications réseau. Toutefois, leurs architectures figées ne leur permettent pas de mettre en œuvre des modèles de traitement de paquets (*pipeline*) flexibles.

Un processeur graphique est une unité de traitement destinée à l'exécution des applications adoptant le modèle d'exécution une seule instruction, plusieurs fils d'exécution (Single Instruction, Multiple Threads, ou SIMT). Ce dernier désigne un modèle d'exécution selon lequel un groupe de fils d'exécution (appelé *warp* ou *wavefront*) appliquent simultanément le même flux d'instructions sur plusieurs éléments de données. En effet, le processeur graphique est très connu pour son architecture hautement parallèle, vu qu'il incorpore des centaines à des milliers de cœurs de calcul et dispose d'une bande passante mémoire très large, par opposition à celle du processeur central.

Traditionnellement, ce dispositif est utilisé pour accélérer les applications responsables du rendu graphique ou du traitement de la vidéo (p. ex., les applications basées sur les biblio-

thèques DirectX et OpenGL). Il est aussi couramment utilisé pour accélérer des traitements qui sont habituellement effectués par le processeur central et dont l'intérêt n'est pas graphique en premier lieu. Ce type de traitement, qui s'appuie sur des modèles de programmation favorisant l'emploi du processeur graphique pour effectuer du calcul haute performance, est connu sous le terme GPGPU (General-Purpose computation on Graphic Processing Units). La plupart des fabricants des processeurs graphiques proposent des bibliothèques qui implémentent ces modèles de programmation, dont les plus connues sont CUDA [49] et OpenCL [50].

Heureusement, la plupart des applications réseau sont développées selon le modèle SIMT et, de ce fait, peuvent être accélérées par les processeurs graphiques [38]. En fait, la littérature inclut plusieurs travaux de recherche affirmant que ce coprocesseur peut justement aider à améliorer les performances de ce type d'applications [48, 51–53]. À titre d'illustration, Tseng et al. [51] ont montré que le débit réseau peut être augmenté par un facteur allant jusqu'à 2.5 lorsque un processeur graphique intégré est utilisé pour décharger certains traitements sur les paquets entrant (p. ex., commutation des datagrammes, routage des paquets IPv4/IPv6, recherche par calcul d'adresse).

De même, plusieurs auteurs [52, 54] ont proposé des plateformes qui se basent sur les processeurs graphiques discrets (appelés aussi cartes graphiques) pour accélérer une variété des applications réseau. Une carte graphique, contrairement à un processeur graphique intégré, dispose de sa propre mémoire dédiée, ce qui lui permet d'offrir des performances très élevées. Toutefois, ces auteurs ont mis en garde contre le surcoût de communication considérable entre le processeur central et la carte graphique à travers le bus PCI-e. Comme solution pour atténuer ce surcoût, ils suggèrent de traiter les paquets en lots avec ce type de processeurs graphiques.

2.3.1 Analyse de performance des processeurs graphiques

Contrairement aux outils dédiés à l'analyse de l'activité du processeur central, les outils spécialisés conçus pour l'analyse de performance des processeurs graphiques sont peu nombreux. Nvidia, AMD et Intel, qui comptent parmi les principaux acteurs du marché des solutions graphiques, ont proposé des environnements d'analyse de performance qui sont compatibles seulement avec leur propre matériel. Souvent, ces environnements d'analyse sont à code source fermé. Heureusement, quelques solutions ouvertes, qui sont généralement développées par des universités et des laboratoires de recherche, ont vu le jour. Nous présentons ces outils dans les paragraphes suivants.

Plusieurs constructeurs des processeurs graphiques offrent des interfaces de programmation qui donnent accès à des informations permettant de caractériser les performances de leurs

matériels et plateformes de calcul [55–57]. À titre d'illustration, Nvidia offre *CUPTI (CUDA Performance Tools Interface)* [57], une infrastructure de traçage et de profilage qui est intégrée dans plusieurs outils dédiés à l'analyse de performance des applications basées sur CUDA [58–60]. Cette infrastructure offre une API de rappel (*Callback API*) permettant à ces outils d'injecter du code d'analyse dans les entrées et sorties des fonctions de l'API du moteur d'exécution de CUDA. Elle offre également un mécanisme qui permet d'interroger les compteurs d'évènements matériels du processeur graphique ainsi que les compteurs d'évènements logiciels du pilote de CUDA. La plupart des outils d'analyse incorporent cette infrastructure afin de collecter des informations décrivant les interactions de l'application surveillée avec le moteur d'exécution de CUDA.

Nsight Systems [61] et *VTune Amplifier* [62] sont deux environnements d'analyse pour les applications s'exécutant dans les environnements hétérogènes manufacturés respectivement par Intel et Nvidia. Un système est dit hétérogène s'il comporte plusieurs unités de calcul de différents types, tels que les processeurs centraux, les processeurs graphiques et les circuits FPGA. Essentiellement, les fonctionnalités que proposent ces deux environnements pour profiler l'activité du GPU permettent d'analyser l'exécution des applications GPGPU et multimédias, vérifier la présence des goulots d'étranglement, et faire des optimisations au niveau de leur code source.

En effet, *Nsight Systems* est utilisé pour identifier et diagnostiquer les problèmes de performance des applications qui exploitent les processeurs graphiques, telles que celles basées sur les plateformes CUDA et OpenGL. De même, *VTune* intègre de nombreux outils qui permettent d'analyser l'exécution des programmes accélérés avec les plateformes OpenCL et OpenGL. Le fonctionnement de ces deux environnements d'analyse repose sur la collecte des données de performance depuis les compteurs d'évènements matériels et les traces générées par les moteurs d'exécution des plateformes.

CodeXL [63] est un environnement d'analyse qui intègre plusieurs outils de profilage et de débogage pour les systèmes hétérogènes manufacturés par AMD. En effet, le profileur du GPU de *CodeXL* dispose de plusieurs fonctionnalités d'analyse pour les applications utilisant les bibliothèques OpenCL, HSA, et DirectX. À titre d'illustration, ce profileur permet d'effectuer des analyses sur les trames des applications multimédias, les fonctions les plus exécutées (*hotspot analysis*), et l'occupation des noyaux de calcul. Le profileur de *CodeXL* collecte les données de performance à partir de plusieurs sources, telles que les compteurs de performance matériels et les traces générées par les moteurs d'exécution des bibliothèques.

TAU Performance System [59] est une suite d'outils open source pour le profilage et le traçage des applications multifils et parallèles. Cette suite est surtout connue pour sa capacité

à combiner plusieurs techniques pour la collecte des données de performance comme l'instrumentation statique du code source, l'échantillonnage d'évènements, et le traçage. L'instrumentation statique du code source de l'application se fait grâce à une intégration avec le compilateur *gcc*. De ce fait, elle est capable de fournir des mesures de performance pour plusieurs plateformes d'accélération de calcul, telles que CUDA, OpenACC et OpenCL. L'intégration et l'analyse des données collectées permettent à cette solution de fournir une vue unifiée permettant de détecter des problèmes de performance variés.

HPCToolkit (High-Performance Computing Toolkit) [64] est une suite d'outils de mesure et d'analyse de performance pour les applications de calcul haute performance. Cette suite incorpore des extensions [60, 65] qui permettent d'analyser les applications accélérées par les plateformes CUDA et OpenMP. Grâce à ces extensions, HPCToolkit peut calculer des métriques de performance, telles que le temps d'exécution des noyaux de calcul et le taux d'utilisation du GPU, et attribuer des indicateurs de performance aux lignes du code source, boucles et code incorporé. Cette suite se base sur l'échantillonnage des compteurs d'évènements logiciels et matériels et l'instrumentation du code pour collecter des données de performance.

Certains outils, comme *LTTng-HSA* [66] et *CLUST* (OpenCL User Space Tracepoint) [67], exploitent la technique de bibliothèques enveloppantes afin d'intercepter les appels aux fonctions des API fournies par les plateformes de calcul (p. ex., CUDA, OpenCL et HSA) et collecter ainsi des données de performance. L'avantage de cette technique est qu'elle ne nécessite pas l'instrumentation des sources des bibliothèques et des applications cibles. Cependant, les outils basés sur cette technique deviennent étroitement liés à des plateformes spécifiques et incapables d'analyser les performances des applications qui utilisent d'autres plateformes.

CLUST est un outil qui analyse les performances des applications accélérées par la plateforme OpenCL à travers un traçage au niveau de l'espace utilisateur. Cet outil précharge des bibliothèques enveloppantes pour intercepter les appels aux fonctions de l'API fournie par OpenCL et générer une trace au format CTF. L'analyse de cette trace permet de construire la pile d'appels de l'application exécutée et d'en dériver quelques métriques de performance comme les durées d'exécution et d'attente des noyaux de calcul d'OpenCL.

LTTng-HSA est un autre outil qui utilise la technique des bibliothèques enveloppantes pour générer des traces unifiées qui dénotent l'activité des processeurs centraux et des processeurs graphiques pour les programmes s'appuyant sur l'interface de programmation HSA (Heterogeneous System Architecture). HSA est une spécification multiplateforme qui facilite la communication entre divers types d'unités de calcul, tels que les processeurs centraux et graphiques, tout en minimisant le transfert des données entre les mémoires. Essentiellement,

l'analyse des données de traçage générées par cet outil permet d'obtenir des informations détaillées sur les appels aux fonctions de l'API de HSA ainsi que sur les durées d'exécution des noyaux de calcul. En combinant cette analyse à celle d'une trace au niveau du noyau du système d'exploitation, on peut obtenir une vue globale de l'utilisation des ressources CPU et GPU.

2.3.2 Virtualisation des processeurs graphiques

Pour compenser le surcoût induit par la couche de virtualisation et l'inefficacité des infrastructures de calcul générales, les VNF peuvent nécessiter une forme d'accélération matérielle pour atteindre leurs objectifs de performance. En effet, le processeur graphique représente, dans ce contexte, un accélérateur idéal pour ce type de fonctions réseau, étant donné sa disponibilité et la facilité avec laquelle il peut être programmé. Heureusement, la plupart des fournisseurs infonuagiques proposent des machines virtuelles équipées de processeurs graphiques virtuels (*Virtual GPUs* ou vGPUs) [1]. Dans cette section, nous présentons les différentes approches de virtualisation de ce coprocesseur qui ont été rapportées dans la littérature.

Virtualisation au niveau des bibliothèques

Cette approche de virtualisation consiste à installer dans la machine virtuelle un module frontend pour intercepter les appels aux fonctions des bibliothèques de calcul basées sur le processeur graphique (p. ex., CUDA). Les appels interceptés sont transmis à un module backend du côté de la machine hôte qui les exécute à l'aide des processeurs graphiques à bord. Le résultat de l'exécution des fonctions de la bibliothèque est retourné à l'application émettrice en suivant le chemin inverse. Plusieurs solutions de virtualisation ont adopté cette approche pour virtualiser le processeur graphique au niveau le plus haut [68–72]. L'inconvénient majeur de cette approche est que le module frontend pourrait devenir obsolète si les bibliothèques de calcul sont mises à jour.

Virtualisation au niveau du pilote

Il existe des solutions qui émulent les processeurs graphiques à travers l'implémentation des modules additionnels au niveau du pilote du processeur graphique [73–75]. La plupart du temps, ces solutions sont développées pour des pilotes à code source ouvert. À titre d'illustration des solutions basées sur cette approche, on peut citer GVT-G (Graphics Virtualization

1. À titre d'illustration, AWS offre des machines virtuelles appelées *GPU Elastic Amazon EC2*, qui intègrent des processeurs graphiques virtuels

Technology – Grid generation), la solution de virtualisation d'Intel pour ses processeurs graphiques [73]. Étant à l'origine compatible uniquement avec Xen, cette solution fut portée vers KVM sous le nom de KVMGT [76].

GVT-G utilise une technique appelée *Mediated Passthrough* qui permet d'offrir aux machines virtuelles un accès direct aux ressources critiques du processeur graphique (les mémoires tampons de commandes et de trames), en plus d'intercepter et émuler les opérations privilégiées. GVT-G partage la mémoire graphique globale entre les processeurs graphiques virtuels, pour renforcer l'isolation des machines virtuelles qui sont rattachées à un même processeur graphique physique.

Contrairement aux autres solutions de virtualisation, cette solution présente l'avantage de prendre en charge la migration en direct des machines virtuelles [77]. En plus, elle permet aux processeurs graphiques virtuels de bénéficier des performances qui avoisinent celles d'un processeur graphique physique. Elle présente, toutefois, deux inconvénients majeurs qui sont l'incapacité de prendre en charge une densité élevée de processeurs graphiques virtuels et le surcoût induit par les accès mémoire intensifs [74, 75].

Virtualisation assistée par le matériel

Plusieurs fabricants de matériel informatique se sont appuyés sur les spécifications de SR-IOV (Single Root–Input Output Virtualization) pour proposer des solutions de virtualisation pour leurs processeurs graphiques. Le SR-IOV est une extension aux spécifications de PCI Express (PCI-e) qui permet de partager les ressources matérielles d'un périphérique PCI entre plusieurs machines virtuelles. En effet, cette extension présente le périphérique PCI à l'hyperviseur sous la forme d'un ensemble de périphériques PCI distincts.

En se basant sur les spécifications de SR-IOV, AMD a développé MxGPU (Multi-users GPU), une solution de virtualisation qui permet de répartir les ressources du processeur graphique sur 16 processeurs graphiques virtuels [78]. De son côté, Nvidia a proposé sa technologie nommée Nvidia Grid [79] qui s'appuie sur les mêmes spécifications. L'architecture de cette technologie sépare, au niveau matériel, les espaces d'adressage des machines virtuelles moyennant une ré-implémentation de l'unité de gestion de mémoire MMU (*Memory Management Unit*). Cette architecture prévoit également des espaces mémoire tampons et des canaux de soumission de commandes dédiés pour chaque machine virtuelle.

Les solutions de virtualisation du processeur graphique mentionnées plus haut présentent l'avantage d'avoir un faible surcoût, ce qui permet au processeur graphique virtuel de bénéficier de performances très élevées. En plus, ces solutions assurent plus d'isolation et de

sécurité pour les machines virtuelles, étant donné qu'elles partagent les ressources du processeur graphique physique au niveau du matériel.

2.4 Traçage

La collecte des données est une tâche essentielle dans tout processus de monitorage ou d'analyse de performance d'un système informatique. Cette tâche consiste à collecter les données décrivant les changements des états internes du système surveillé et aussi des systèmes avec qui il interagit.

Principalement, il existe deux techniques permettant la collecte des données de performance à partir d'un programme en exécution, qui sont à savoir la journalisation et le traçage. La journalisation consiste à émettre des messages qui dénotent soit la progression de l'exécution du programme, soit l'occurrence d'un problème quelconque ou d'une situation imprévue (p. ex., messages d'information, d'avertissement et d'erreur). Les informations produites par la journalisation sont habituellement destinées à aider l'utilisateur à détecter et à déboguer rapidement certains types d'erreurs. Elles sont, de ce fait, de haut-niveau et très peu abondantes.

D'un autre coté, la deuxième technique, appelée traçage, est particulièrement efficace pour la collecte des données de bas niveau avec une granularité de détails assez fine. Souvent, et contrairement à la journalisation, le traçage nécessite l'utilisation d'un outil externe à l'application qui s'appelle traceur. En effet, de nos jours il existe une multitude des traceurs qui varient notamment aux niveaux des fonctionnalités offertes et du surcoût induit par l'opération de traçage. Pour les systèmes Linux, on distingue plusieurs traceurs parmi lesquels strace [80], LTTng [81], Ftrace [82], et SystemTap [83] sont très populaires.

2.4.1 Outils de Traçage

Dans cette section, nous présentons les outils de traçage les plus utilisés qui fonctionnent sous Linux.

LTTng

LTTng (Linux Trace Toolkit next generation) est un outil de traçage libre et compatible avec Linux [81]. Ce traceur prend en charge deux modes de traçage : le mode noyau grâce aux modules *lttng_modules* et le mode utilisateur grâce aux bibliothèques *lttng-ust*. Le traçage du noyau peut se faire statiquement en instrumentant son code source à l'aide de la macro

TRACE_EVENT ou dynamiquement à l'aide des *kprobes* et des *kretprobes*. Le noyau Linux contient plusieurs points de trace qui permettent de rendre transparents tous ses mécanismes internes (ordonnancement des processus, systèmes de fichiers, pile réseau, etc.).

LTTng est conçu et implémenté de manière à réduire au maximum le surcoût de traçage. En effet, plusieurs structures de données et des algorithmes sans verrous, issus de la bibliothèque *liburcu* (Userspace Read-Copy-Update) [84], ont été exploités afin d'éviter l'utilisation des mécanismes de cohérence de cache au niveau des processeurs multicœurs. Par exemple, pour assurer un enregistrement non bloquant des évènements émis par les processus tracés, LTTng utilise une mémoire tampon circulaire pour chaque cœur de processeur. Pour gérer la perte des évènements dans le cas où cette mémoire tampon serait saturée, LTTng propose à l'utilisateur le choix entre écraser les évènements les plus anciens (*overwrite mode*) ou abandonner l'enregistrement des évènements nouvellement émis (*discard mode*).

Contrairement à plusieurs autres traceurs, LTTng permet la sérialisation des évènements enregistrés dans la mémoire centrale sur disque. Les évènements peuvent alors être sauvegardés dans des fichiers de trace dans un format binaire ouvert appelé CTF (Common Trace Format). La sauvegarde dans une même trace des évènements noyau et des évènements de l'espace utilisateur permet d'analyser les performances de l'application surveillée à plusieurs niveaux de sa pile logicielle.

Ftrace

Ftrace (Function Tracer) est un cadriciel de traçage qui est intégré au noyau Linux. Le contrôle et la configuration de ce cadriciel peuvent se faire soit à travers le système de fichier virtuel *tracefs*, soit en utilisant l'utilitaire *trace-cmd*. En effet, Ftrace comprend plusieurs modules de traçage et d'analyse de trace. Toutefois, il a été initialement développé pour tracer uniquement les appels aux fonctions du noyau à l'aide des traceurs *function* et *function_graph*. La sortie produite par ces deux traceurs permet, entre autres, aux modules d'analyse de calculer les durées d'exécution des fonctions noyau et de représenter graphiquement la pile des appels.

Au fil du temps, Ftrace a été enrichi avec de nouvelles fonctionnalités plus sophistiquées. En guise d'exemples, on peut citer le traceur *blktrace* qui trace les requêtes d'E/S au niveau des périphériques de stockage, le traceur *hwlat* qui mesure les délais de latence du matériel, et le traceur *wakeup* qui trace les latences maximales entre l'instant où les processus sont réveillés par l'ordonnanceur et l'instant où ils deviennent prêts pour reprendre leur exécution. Ftrace prend en charge également le traçage statique des évènements à travers la macro *TRACE_EVENT*, qui donne accès à l'infrastructure de traçage statique du noyau, et le

traçage dynamique à travers les mécanismes *Kprobes* et *Kretprobes*.

Ftrace est considéré comme l'un des traceurs noyau les plus stables, puisqu'il est maintenu par une communauté très active, qui est la communauté du noyau Linux. Toutefois, il présente certaines limites comme l'incapacité de tracer dans l'espace utilisateur. Aussi, ce cadriciel ne peut pas tracer certaines fonctions du noyau, comme les fonctions de type *inline*.

SystemTap

SystemTap est un cadriciel de traçage pour Linux qui permet de tracer aussi bien le noyau que les applications des utilisateurs. Ce cadriciel prend en charge plusieurs types d'évènements tels que les entrées et sorties des appels système et des fonctions, les points de trace statiques du noyau, ainsi que les points de trace dynamiques implémentés via *Kprobe*. SystemTap offre aussi la possibilité de tracer l'exécution des applications dans l'espace utilisateur via les modules *uprobe*. Les évènements asynchrones sont aussi pris en charge, ce qui permet de définir des évènements qui se déclenchent à l'expiration d'un temporisateur ou lorsqu'un compteur atteint une valeur seuil.

L'instrumentation avec SystemTap se fait à l'aide des scripts écrits dans un langage ayant une syntaxe similaire à celle du langage C. Le langage de script de SystemTap est un langage complet, puisqu'il définit plusieurs structures de contrôle telles que les fonctions, les boucles, et les structures conditionnelles. Le traceur exploite ces structures pour prendre en charge le traçage conditionnel et offrir à l'utilisateur une grande flexibilité quant au traitement des données de traçage à exporter. Aussi, ce langage est typé et il prend en charge deux types de données : les entiers et les chaînes de caractères. En effet, définir un point de trace avec ce langage revient à écrire le code à exécuter lorsque celui-ci est déclenché, puis à y associer un type d'évènement. Par exemple, le fragment de script suivant permet d'émettre un évènement à chaque fois qu'un processus reçoit un paquet UDP.

```
probe udp.recvmsg.return {
    bytes = $return
    if(bytes > 0) {
        printf("UDP - Thread (%s:%d) received %d bytes\n",
        execname(), pid(), bytes)
    }
}
```

SystemTap est un cadriciel de traçage qui intègre une multitude de fonctionnalités qui simplifient l'opération de traçage. Toutefois, il présente certaines limites qui concernent notamment

le surcoût du traçage et le format adopté pour le stockage des traces sur le disque. Les auteurs dans [85] ont utilisé l'étalonnage pour comparer les performances de strace, LTTng et SystemTap et ils ont mis en évidence les performances réduites de ce dernier. Leur étude souligne que la détérioration de performance de SystemTap est essentiellement due à l'exécution des *handlers* associés aux évènements dans l'espace noyau. Également, la vérification des scripts, leur compilation et le chargement des modules résultats dans le noyau ajoutent un surcoût supplémentaire.

2.4.2 Outils de lecture et d'analyse des traces

Babeltrace

Babeltrace [86] est l'implémentation de référence pour le format de traces CTF. Il s'agit d'un format binaire compact et très optimisé pour l'écriture des évènements sur le disque. Une trace au format CTF est structurée sous la forme de plusieurs flux d'événements, où les événements sont classés par ordre chronologique d'estampilles de temps.

Chaque évènement dans une trace CTF est constitué d'un entête, un contexte, et une liste variable des arguments. L'entête contient essentiellement l'identifiant et l'estampille de temps de l'évènement. Le contexte contient des informations facultatives qui peuvent être associées à un évènement, telles que l'identifiant du processus qu'il l'a généré (*pid*) et le numéro du processeur (*cpu*) qui l'a sauvegardé en mémoire. La liste des arguments dénote l'ensemble des données de l'utilisateur (chaînes de caractères, entiers, etc.) qu'on veut exporter à travers l'évènement.

Babeltrace est aussi le nom d'un outil qui lit les traces CTF et les affiche sur terminal dans un format textuel compréhensible par l'utilisateur. Cet outil peut aussi convertir les traces CTF vers d'autres formats de trace. Babeltrace présente une bibliothèque écrite en C qui rend possible la manipulation des traces par la programmation. Il fournit également une interface de programmation permettant de lire et de modifier les évènements de la trace à l'aide des scripts en langage Python.

Trace Compass

Trace Compass [87] est un logiciel d'analyse et de visualisation des traces à source ouverte. Ce logiciel offre un ensemble des vues, des graphes, et des métriques permettant d'analyser la trace sous des angles différents, ce qui aide à dévoiler les aspects cachés du système surveillé et découvrir ses problèmes de performance. Trace Compass présente l'avantage d'être très optimisé pour l'analyse des traces de grandes tailles, en plus de prendre en charge plusieurs

formats de trace, tels que les formats CTF, BTF (Best Trace Format), et PCAP.

Ce logiciel fournit de nombreuses vues interactives qui incorporent des représentations graphiques diverses (Figure 2.4). Ces vues permettent de naviguer dans la trace d'une façon interactive et de varier le niveau de détails avec un surcoût négligeable. De plus, elles sont synchronisées, ce qui veut dire que la sélection d'un évènement, d'une estampille de temps ou d'un intervalle de temps dans une vue entraîne la mise à jour automatique des autres vues.

Trace Compass implémente plusieurs techniques d'abstraction, que ce soit au niveau de l'analyse ou de la visualisation des traces. Par exemple, une de ses techniques d'analyse intéressantes vise à étudier le comportement du système surveillé par la reconstitution de ses états d'exécution à partir des évènements de la trace. En effet, cette technique modélise le système comme un ensemble fini des états, et tout changement d'un état à un autre ne peut se produire que lorsque des événements sont déclenchés. De ce fait, analyser les performances du système revient à vérifier la cohérence de ses états, ainsi que les états de ses ressources.

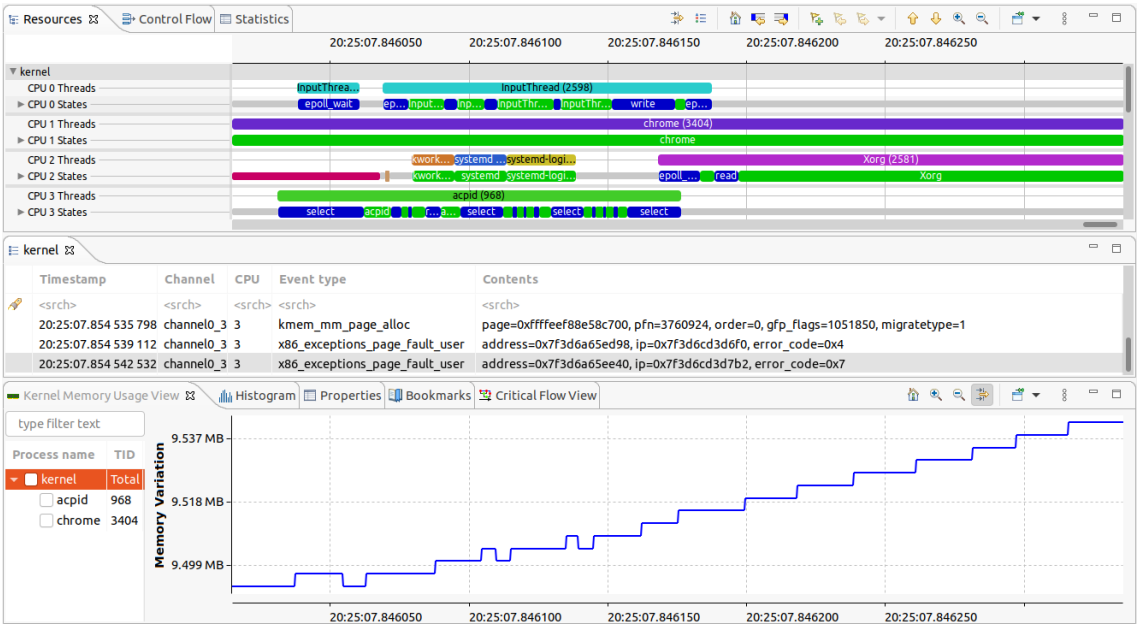

Figure 2.4 Analyse de trace avec Trace Compass

2.5 Synthèse

Dans ce chapitre, nous avons étudié deux technologies réseau émergentes, qui sont le SDN et la NFV. Ces technologies sont le fruit de plusieurs initiatives qui ont cherché à résoudre le problème de rigidité des réseaux traditionnels. D'après plusieurs experts dans le domaine, ces

deux technologies sont susceptibles de changer profondément le fonctionnement des réseaux actuels, ainsi que les méthodes et les pratiques par lesquelles ils sont gérés.

Bien que le SDN et NFV soient deux approches indépendantes, rien n'empêche de les mettre en œuvre en simultané dans un même réseau. En effet, le SDN rend le réseau plus programmable et améliore, de ce fait, sa flexibilité, son agilité et son évolutivité. De son côté, la NFV augmente la polyvalence du matériel réseau et accélère, en conséquence, le développement et le déploiement des applications, des services, et des produits innovants.

Principalement, le SDN et la NFV reposent sur la virtualisation et l'abstraction du réseau, ce qui ajoute un surcoût supplémentaire et réduit les performances des systèmes basés sur ces approches. Donc, nous avons examiné plusieurs techniques permettant l'accélération de ces systèmes, tant au niveau matériel que logiciel.

Plusieurs travaux de recherche ont été proposés dans la littérature pour évaluer, surveiller, et diagnostiquer les performances des systèmes basées sur le SDN et la NFV. Ces travaux se basent sur des approches et des techniques très variées. Certains travaux utilisent l'étalonnage pour tester le comportement du système et évaluer ses performances sous des charges spécifiques. Néanmoins, cette technique, bien qu'utile pour comprendre les limites du système et avoir une estimation globale de ses performances, ne permet pas d'identifier les goulots d'étranglement ou les autres bogues de performances.

Par ailleurs, d'autres travaux s'appuient sur des protocoles réseaux, tels que SNMP et sFlow, pour collecter des données de performance depuis les équipements du réseau. Toutefois, l'utilisation de ces protocoles présente des inconvénients majeurs. D'abord, ces protocoles ne sont pas capables de collecter des données de performance de bas niveau, qui sont souvent nécessaires pour pouvoir diagnostiquer les problèmes de performance complexes. Aussi, dans le but de réduire leur surcoût, la plupart de ces protocoles s'appuient sur des techniques d'échantillonnage variées qui entraînent une perte d'information. Enfin, la plupart des équipements virtuels (p. ex., les commutateurs logiciels comme Open vSwich), ne prennent pas en charge ces protocoles.

D'un autre coté, la revue de la littérature inclut aussi des travaux qui offrent des analyses basées sur le traçage pour diagnostiquer des problèmes de performance complexes. Ces travaux démontrent la pertinence de cette technique et sa capacité de collecter des données de performance détaillées et de très bas niveau. Malheureusement, la plupart de ces travaux n'accordent pas beaucoup d'importance au surcoût du traçage, et utilisent de ce fait des algorithmes et des structures de données qui ne sont pas efficaces. En effet, le traçage est très susceptible de causer un surcoût considérable. Ce surcoût pourrait influencer le comportement du système surveillé et, par voie de fait, affecter la précision de l'analyse. Par

conséquent, il est essentiel que les analyses de performance incluent une évaluation précise du surcoût induit par cette technique.

De plus, la plupart des analyses fournies sont basées sur des données collectées à l'aide d'un traçage au niveau utilisateur. En fait, très rares sont les analyses qui s'appuient sur des traces unifiées incorporant des évènements noyau et utilisateur. En effet, un traçage au niveau du noyau permet de révéler des informations pertinentes sur les mécanismes internes du noyau, ainsi que leur influence sur les performances de l'application surveillée : états des fils d'exécution (en cours d'exécution, en attente de CPU, bloqué sur les E/S), les interférences entre les processus, les conflits sur les ressources, etc. En combinant les événements au niveau du noyau et au niveau de l'application, on obtient toutes les informations nécessaires pour identifier les problèmes et diagnostiquer leurs causes fondamentales.

CHAPITRE 3 MÉTHODOLOGIE

Dans ce chapitre, nous présentons la méthodologie que nous avons adoptée pour mener à bien notre travail de recherche. De manière générale, l'étude de performance d'un système donné fait appel à deux types de tâches : la collecte de données et l'analyse des données collectées. Dans les sections suivantes, nous détaillons la démarche que nous avons entreprise pour mettre en œuvre ces tâches. Nous décrivons également notre environnement de travail et présentons nos axes de recherche.

3.1 Collecte de données

Le diagnostic des problèmes de performance complexes requiert souvent la collecte des données détaillées permettant de décrire le comportement et les états du système surveillé. De ce fait, puisque le traçage est une technique efficace qui répond parfaitement à ce besoin (voir chapitre 2, section 2.4), nous avons décidé de l'utiliser. De plus, nous avons choisi LTTng comme notre traceur principal. Cet outil est bien connu pour sa capacité à tracer avec un très faible surcoût, ce qui le rend très pratique dans les environnements de production. En outre, LTTng est capable d'effectuer un traçage en simultané, au niveau du noyau et de l'espace utilisateur. Ceci élimine le besoin de réordonner les événements, quand l'analyse doit porter sur une trace composée par des évènements de l'espace utilisateur et ceux de l'espace noyau.

Le traçage de l'exécution d'une application donnée nécessite, la plupart du temps, l'instrumentation de son code source. Cette technique, appelée instrumentation statique, consiste à insérer des sondes à des emplacements stratégiques de son code source. Lorsque le chemin d'exécution de l'application passe à travers ces sondes, les événements sont déclenchés et les données qu'ils exportent sont sauvegardées dans la mémoire tampon du traceur.

Il est important de souligner que si l'instrumentation n'est pas bien conçue, l'impact sur le comportement et les performances de l'application peut être non négligeable. Par conséquent, notre approche pour minimiser cet impact repose sur l'ajustement de la granularité de l'instrumentation requise par nos analyses. Cela revient essentiellement à faire un compromis entre le type et la quantité de données à exporter dans la trace, et l'impact sur les performances de l'application tracée (voir Figure 3.1).

Parfois, l'accès au code source de l'application n'est pas autorisé pour des raisons de confidentialité. Dans ce cas-ci, nous proposons de faire un recours à l'instrumentation dynamique,

bien que son surcoût dépasse celui de l'instrumentation statique. L'instrumentation dynamique consiste à insérer des points de trace dans les fichiers binaires de l'application (p. ex., lors de la compilation ou de l'édition des liens), ou directement dans son espace mémoire, lorsqu'elle est en cours d'exécution.

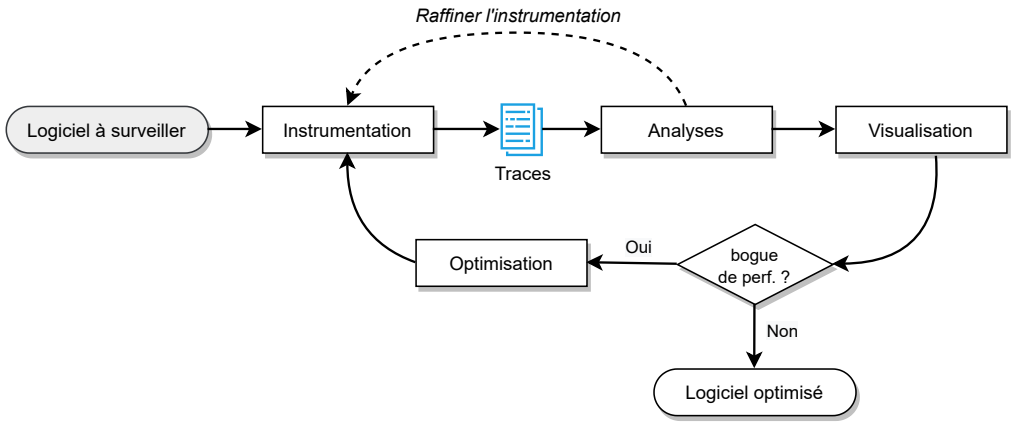

Figure 3.1 Démarche adoptée pour l'analyse de performance des applications réseau cibles

3.2 Analyse de données

Le traçage souvent donne lieu à des fichiers de trace volumineux contenant des millions, voire des milliards, d'évènements de bas niveau. En effet, la taille des fichiers de trace peut facilement atteindre plusieurs giga-octets dans l'espace de quelques minutes seulement. Les causes qui sont derrière l'augmentation rapide de la taille des fichiers de trace sont nombreuses. On peut citer, à titre d'exemples, la taille de l'application tracée, la granularité de l'instrumentation effectuée, le niveau de détail requis par les analyses, et la durée du traçage. Par ailleurs, la taille énorme des traces rend toute analyse manuelle des données impossible, surtout que plusieurs évènements de traçage ne sont compréhensibles que lorsqu'ils sont corrélés avec d'autres évènements. Par conséquent, le développement des analyses efficaces pour interpréter les données de traçage et en extraire des informations pertinentes est essentiel.

3.2.1 Méthode d'analyse

Pour créer des modèles permettant l'analyse de performance de l'application tracée, nous avons adopté une approche efficace qui se base sur l'emploi des techniques d'abstraction de données. Le but de ces techniques est de réduire le niveau de détails de la trace, à un niveau

qui permet à l'utilisateur d'analyser le comportement de l'application et de découvrir des bogues de performance.

À titre d'illustration, une des techniques d'abstraction utilisées considère l'application comme un automate à états finis, et tout changement d'état ne peut se produire que lorsqu'un ou plusieurs évènements sont déclenchés. Diagnostiquer des bogues avec cette technique revient donc à vérifier la cohérence des états de l'application, ainsi que ceux de ses ressources et des systèmes avec lesquels elle interagit. Une autre technique intéressante consiste à calculer des métriques caractérisant les performances de l'application, à partir des données de traçage. En effet, une dégradation de performance est détectée, si la valeur d'une métrique devient inférieure ou supérieure à une valeur seuil.

Pour implémenter ces techniques, nous avons besoin de sauvegarder les états et les attributs de l'application tracée dans une structure de données. Le but est de ne pas avoir à relire la trace à chaque fois que l'utilisateur veut varier le niveau de détail de l'analyse ou sélectionner une autre analyse. Cette structure de données doit satisfaire plusieurs exigences, telles que la capacité à gérer une très grande quantité d'information et la garantie des accès en lecture rapides. En effet, nous nous sommes basés sur la structure de données appelée SHT (State History Tree) [88] dans le développement de nos analyses de performance. Cette structure de données est évolutive et prend en charge de nombreux formats de trace. De plus, elle offre un accès rapide en lecture aux données enregistrées, dont la durée est estimée à $O(logn)$.

3.2.2 Synchronisation des traces

Étant donné que notre recherche porte sur l'analyse de performance des réseaux émergents, on est parfois amené à lancer, en simultané, des sessions de traçage au niveau de nombreuses machines interconnectées. Le recours à une telle procédure est inévitable dans certains cas d'utilisation, comme le traçage de l'exécution d'une application distribuée, dont les composantes sont réparties sur plusieurs machines, ou le traçage des machines virtuelles, ainsi que de la machine hôte qui l'héberge.

Par ailleurs, la combinaison des traces collectées à partir de ces machines en une seule trace unifiée est essentielle pour développer des analyses pertinentes. Toutefois, cette procédure pose un défi quant au maintien de l'ordre entre les évènements issus des traces différentes, puisque chaque machine utilise une horloge différente. Autrement dit, on ne peut pas ordonner les estampilles de temps des évènements, parce que ces derniers sont enregistrés sur des machines qui n'utilisent pas la même référence temporelle.

LTTng se base sur l'horloge monotone, fournie par le noyau Linux, dans la génération des

estampilles de temps nécessaires pour horodater les évènements de traçage. Cette horloge ne peut pas être ajustée à l'aide des protocoles de synchronisation de temps, puisqu'elle n'est pas conçue pour reculer. Néanmoins, LTTng peut être configuré pour prendre, comme source de temps, une autre horloge ajustable, telle que l'horloge murale. Dans ce cas, il est possible d'utiliser un protocole de synchronisation de temps, tel que le protocole NTP (Network Time Protocol), pour synchroniser les horloges des différentes machines du réseau. Malheureusement, les protocoles de synchronisation de temps qui sont actuellement disponibles sont incapables de fournir une précision suffisante (p. ex., à la microseconde ou mieux), qui est nécessaire pour la synchronisation des évènements de LTTng.

Pour résoudre ce problème de synchronisation des traces issues des machines différentes, nous nous proposons d'utiliser l'algorithme *The fully incremental convex hull synchronization algorithm* [89]. Cet algorithme se base sur le lien causal entre les évènements d'envoi et de réception des messages entre les machines communicantes, pour définir un ordre partiel entre tous les évènements de leurs traces. En d'autres termes, cet algorithme utilise la paire d'évènements $<x,y>$ comme ancre de synchronisation, si ces évènements vérifient les deux conditions suivantes :

1. Il doit y avoir un lien de causalité (relation de cause à effet) entre l'évènement x enregistré dans la trace de la machine M1 et l'évènement y enregistré dans la trace de la machine M2 ;
2. Les deux évènements x et y doivent partager un identifiant unique permettant de les distinguer en tant que paire (p. ex., les valeurs des champs port et numéro de séquence dans l'en-tête TCP, ainsi que les adresses IP source/destination).

3.3 Évaluation des coûts

Dans un environnement de production, l'analyse de performance d'une application donnée doit répondre à des exigences très strictes, notamment en terme du surcoût. En effet, un surcoût trop élevé peut impacter négativement les performances de l'application surveillée, ainsi que celles du système sur lequel elle s'exécute. Bien pire encore, si cette application est une application temps réel, l'impact sur les performances pourrait causer la violation de ses contraintes temporelles, et par conséquent, la modification de son comportement.

Dans cette optique, il est essentiel de mesurer le surcoût ajouté par la phase de collecte de données. Aussi, l'évaluation du surcoût de la phase d'analyse de données est importante afin de s'assurer de l'extensibilité de la solution d'analyse proposée. De ce fait, nous nous proposons d'évaluer le surcoût du traçage à l'aide de deux métriques. La première métrique

mesure l'augmentation du temps d'exécution de l'application tracée, alors que la deuxième mesure la taille des traces générées.

En effet, nous calculons l'augmentation du temps d'exécution de l'application par la mesure du temps qu'il lui faut pour traiter une charge spécifique, quand les points de trace sont activés et désactivés. Pour créer la charge à imposer à l'application cible, nous nous proposons d'utiliser un générateur de trafic logiciel (p. ex., *Trex* [90]). En ce qui concerne la phase d'analyse, nous évaluons le surcoût de l'exécution de nos analyses par la mesure du temps nécessaire à Trace Compass pour charger les traces, et créer les structures de données requises.

Afin d'avoir des mesures de performance exactes, il est important de créer un environnement d'exécution aussi déterministe que possible. Pour ce faire, nous utilisons une technique permettant l'isolation des processeurs. Le but est de réduire les interférences entre les processus, à travers l'affectation de chaque fil d'exécution à un cœur différent (p. ex., assigner les fils d'exécution de LTTng et de l'application tracée à des cœurs différents). Nous désactivons également l'*Hyperthreading*, la technologie d'Intel permettant au processeur physique de se présenter comme deux processeurs logiques. Bien que cette technologie soit connue pour sa capacité à améliorer les performances de traitement du processeur, plusieurs travaux ont démontré qu'elle réduit beaucoup le déterminisme.

3.4 Environnement de travail

Dans cette section, nous présentons la configuration matérielle ainsi que l'environnement logiciel que nous avons utilisés pour développer nos analyses et évaluer les performances des systèmes cibles. Le but est d'aider les autres chercheurs à répliquer les résultats issus de notre recherche, et à les comparer avec d'autres résultats, possiblement obtenus avec une configuration différente de la nôtre.

3.4.1 Matériel

Pour réaliser ce travail, nous avons utilisé trois machines ayant des configurations matérielles différentes, mais toutes équipées des processeurs centraux de type Intel x86 SMP (Symmetric Multi Processor). De toute manière, il est à noter que la majorité des analyses et outils développés dans le cadre de cette thèse sont écrits en java, et sont, par conséquent, indépendants de l'architecture.

Pour ce qui est de la collecte des données, il est important de remarquer que LTTng, notre principal outil de traçage, est seulement compatible avec Linux. De ce fait, les versions instrumentées des applications que nous avons analysées au niveau de leurs performances,

peuvent s'exécuter sur n'importe quelle architecture supportée par Linux. Néanmoins, nous signalons que GVT-g, la solution de virtualisation que nous avons instrumentée, supporte uniquement les processeurs graphiques d'Intel (Broadwell et les modèles plus récents).

3.4.2 Logiciel

Étant donné que notre travail de recherche repose en grande partie sur le procédé d'instrumentation statique, notre choix pour les logiciels à utiliser s'est orienté vers ceux qui sont à code source ouvert. Ce choix présente plusieurs avantages, dont le plus important est la possibilité d'étudier le fonctionnement de l'application cible, à travers l'analyse de son code source. De plus, il est facile de corriger les bogues de performance de l'application en modifiant simplement son code source.

Sous cette optique, nous avons choisi Linux, un système d'exploitation libre et à source ouverte. Ce choix est en grande partie justifié par la popularité de Linux dans les environnements infonuagiques. De plus, ce système d'exploitation incorpore en natif plusieurs outils de profilage et de traçage. En outre, Linux présente l'avantage d'être instrumenté et expose des centaines des points de trace, prêts pour être utilisés directement dans nos analyses.

Dans cette même logique, nous avons choisi les outils *KVM* et *QEMU* pour gérer notre environnement de virtualisation. *KVM* est un hyperviseur qui permet aux machines virtuelles de bénéficier de l'accélération matérielle offerte par la machine hôte. De son côté, *QEMU* émule le matériel sur lequel s'exécute le système d'exploitation invité.

Enfin, nous nous sommes servis de Trace Compass, un outil d'analyse des traces, dans l'élaboration de nos analyses de performance. En effet, afin de faciliter l'intégration de nos analyses, nous les avons développées sous la forme de plug-ins Eclipse, dans le projet *incubateur* de Trace Compass. Il est important de noter que le code source des outils développés dans le cadre de cette thèse est disponible dans le référentiel Git de l'auteur [1].

3.5 Axes de recherche

Sur la base de notre étude de la revue de littérature, nous avons identifié deux volets de recherche potentiels, auxquels nous pouvons faire des contributions. Le premier volet de recherche concerne l'analyse de performance des composants logiciels qui constituent l'écosystème SDN (p. ex., contrôleurs SDN, commutateurs logiciels, etc.). Le deuxième volet de recherche concerne l'analyse de performance des composants logiciels impliqués dans une infrastructure NFV. Dans ce dernier volet, nous avons manifesté un intérêt pour les solutions

1. https ://github.com/adel-belkhiri

d'accélération logicielles et matérielles qui sont dédiées à améliorer les performances des VNF. Nous nous sommes également intéressés à l'étude des chaînes de service, ainsi que les liens logiques qui existent entre les VNF.

Le travail que nous avons mené dans le cadre des volets de recherche susmentionnés est sanctionné par la rédaction de quatre articles scientifiques (voir Figure 3.2). Nous décrivons brièvement ces articles dans les paragraphes suivants. Par la suite, nous consacrons les quatre prochains chapitres pour les présenter en détail, chacun dans un chapitre à part.

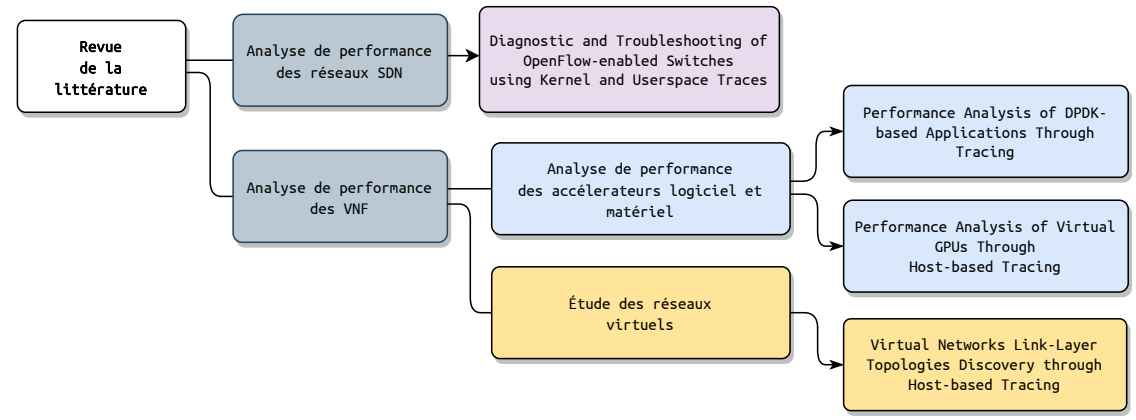

Figure 3.2 Nos activités de recherche ont abouti à quatre articles scientifiques

Diagnostic and Troubleshooting of OpenFlow-enabled Switches using Kernel and Userspace Traces — Cet article s'attaque au problème de diagnostic des bogues de performance dans les réseaux définis par le logiciel (SDN). Bien que cette nouvelle architecture améliore la flexibilité et l'agilité du réseau, elle rend le diagnostic de ses bogues de performance très difficile. Essentiellement, la difficulté vient du fait que le SDN répartit les fonctions réseau sur plusieurs entités. En effet, les bogues de performance peuvent désormais se trouver à plusieurs endroits dans le réseau, tels que les équipements du plan de données, les contrôleurs SDN, les applications réseau qui s'exécutent au-dessus des contrôleurs, et le système d'exploitation.

Cet article propose une approche pour identifier les bogues qui sont susceptibles d'affecter les performances des équipements du plan de données. Cette approche est basée sur l'utilisation du traçage pour la collecte des données de bas niveau, depuis les commutateurs logiciels surveillés. Dans cet article, nous relevons deux défis majeurs. Le premier défi est la corrélation des évènements issus de l'espace noyau et de l'espace utilisateur, afin de déterminer les causes fondamentales des bogues de performance détectés. Le deuxième défi consiste à proposer une méthode permettant de réduire la taille des traces obtenues. La taille considérable des fichiers

de trace, due à la fréquence élevée de génération des évènements, est un inconvénient partagé par plusieurs traceurs.

En effet, notre solution pour réduire le coût de stockage des traces consiste à lancer en parallèle deux sessions de traçage. La première session active un nombre restreint des points de trace, alors que la deuxième session active tous les points de trace. Les évènements générés par la première session sont analysés à la volée, à l'aide des méthodes heuristiques, pour détecter toute dégradation de performance. Le traceur est configuré pour garder en mémoire tous les évènements générés, et sauvegarder sur disque ceux de la deuxième session, lorsqu'une dégradation de performance est détectée à l'aide de la première session. Les traces produites sont analysées hors-ligne pour diagnostiquer les causes fondamentales des problèmes de performance.

Performance Analysis of DPDK-based Applications Through Tracing — Cet article aborde l'analyse de performance des applications réseau qui sont basées sur DPDK, une solution logicielle qui est très utilisée dans l'accélération des fonctions réseaux virtuelles. Cette solution implémente la technique de contournement du noyau, afin d'éviter les goulots d'étranglement qui sont présents dans sa pile réseau. Malheureusement, lorsqu'on contourne le noyau du système d'exploitation, les outils de surveillance et de débogage réseau traditionnels deviennent obsolètes. Par ailleurs, étant donné la complexité des logiciels réseau, le besoin pour des outils de diagnostic et d'analyse des problèmes de performance devient une nécessité.

Dans cet article, nous proposons une approche permettant de découvrir les goulots d'étranglement des applications basées sur DPDK. Cette approche se base sur l'instrumentation des bibliothèques de DPDK pour collecter des données de performance qui caractérisent ces applications. Le principal défi relevé dans cet article est la génération des modèles basés sur l'historique des états qui illustrent le fonctionnement des mécanismes de contrôle implémentés dans DPDK. Les modèles générés sont utilisés pour calculer des métriques de performance adaptées, et les afficher dans des vues graphiques synchronisées.

Performance Analysis of Virtual GPUs Through Host-based Tracing — Cet article étudie la solution de virtualisation des processeurs graphiques appelée GVT-g, et propose une approche permettant d'analyser ses performances. Principalement, cette approche se base sur le suivi des requêtes qui sont envoyées au processeur graphique, à travers plusieurs stades de traitement : le pilote du processeur graphique virtuel de la machine virtuelle, où l'application accélérée est exécutée, la solution de virtualisation GVT-g, et le pilote du processeur graphique de la machine hôte.

Dans cet article, nous relevons deux défis majeurs. Le premier défi consiste à proposer une méthode de collecte de données qui ne nécessite pas un accès à l'intérieur de la machine virtuelle. En effet, toutes les analyses fournies portent sur des traces collectées à partir du noyau de la machine hôte. Le deuxième défi consiste en la génération des métriques permettant de refléter le comportement et les performances des processeurs graphiques virtuels, à partir des évènements ponctuels de la trace.

Virtual Networks Link-Layer Topologies Discovery through Host-based Tracing — L'objectif de cet article est de proposer une méthode permettant de découvrir les topologies des réseaux virtuels, au niveau de la couche liaison de données. En effet, un réseau virtuel est un réseau logique déployé au-dessus d'une infrastructure réseau physique. Ce type de réseaux est composé par des nœuds virtuels (p. ex., machines virtuelles, commutateurs virtuels, et cartes réseau virtuelles) interconnectés à l'aide des liens logiques. Il est important de mentionner que le fonctionnement de plusieurs applications, dans le domaine des réseaux, repose sur la connaissance des topologies des réseaux virtuels. À titre d'exemple, l'objectif du VNE (Virtual Network Embedding) est de déterminer les meilleures façons pour déployer plusieurs réseaux virtuels sur un même réseau physique. Malheureusement, les méthodes permettant d'atteindre cet objectif dans un réseau traditionnel ne sont pas adaptées aux particularités des réseaux virtuels.

Ainsi, cet article propose deux méthodes permettant de déduire la topologie d'un réseau virtuel à partir d'une trace d'exécution. La première méthode se base sur la collecte des données de traçage qui dénotent les opérations appliquées sur les tables de transfert des commutateurs virtuels. La deuxième méthode repose sur la détermination des chemins empruntés par les paquets pour atteindre leurs destinations. Un des défis posés par cette problématique est la synchronisation des traces qui sont issues des machines différentes. Le lancement de plusieurs sessions de traçage en parallèle est requis lorsque les nœuds du réseau virtuel sont déployés sur de nombreuses machines hôtes.

CHAPITRE 4 ARTICLE 1: DIAGNOSTIC AND TROUBLESHOOTING OF OPENFLOW–ENABLED SWITCHES USING KERNEL AND USERSPACE TRACES

Authors

Adel Belkhiri and Michel Dagenais
Department of Computer and Software Engineering, École Polytechnique de Montréal, Montreal, H3T 1J4, Canada

E-mail : {adel.belkhiri, michel.dagenais}@polymtl.ca

Published in International Journal of Communication Systems, 7 June 2021

https://doi.org/10.1002/dac.4920

4.1 Abstract

Although Software-Defined Networking (SDN) provides a flexible way to provision and control networks, it also makes network debugging and troubleshooting more complex. In SDN, the network is fully managed by software programs that increase flexibility and sophistication but are prone to bugs. Pinpointing those bugs is challenging because they can occur at multiple locations, such as the forwarding plane, the controller OS, and the network services running on top of the controller. Compared to functional bugs, performance bugs are particularly irritating due to their non-failure semantics. They can lead to performance loss (reduced throughput, increased latency, and wasted resources) while maintaining the network connectivity. The literature reports several tools and techniques to diagnose SDN bugs, but, unfortunately, they are mostly ineffective against performance bugs.

In this paper, we propose a novel monitoring and diagnostic framework capable of diagnosing performance bugs in the SDN data plane. The proposed tool works within Open vSwitch (OVS), a popular software switch, albeit it can easily be adapted to any OpenFlow switch. Tracing techniques are used to collect low-level performance data from monitored switches. Our tool derives adapted performance metrics from kernel and userspace traces and then displays them in time-synchronized graphical views. These views provide valuable insights into OVS operation. They also enable practitioners to discover performance-related issues and analyze their root causes. A few use cases are presented to demonstrate the efficiency of

our tool in optimizing OVS performance and diagnosing its performance bugs.

4.2 Introduction

Over the last decade, the emergence of SDN has led to a paradigm shift in the way networks are deployed, managed, and controlled. The most important key concept brought by SDN is the physical separation of the control plane from the data forwarding plane in network switches. All the network intelligence and the control logic are thus moved to a central entity called SDN controller, while switches and routers became just simple forwarding devices. By decoupling the networking services and applications from the underlying hardware and enabling more network abstractions, SDN promises to (1) provide more agility and flexibility to network provisioning; (2) ensure better control of the network infrastructure including cross-vendor networking elements; and (3) make network configuration and management less complex through automation and high-level abstractions.

Despite all the benefits mentioned above, the complexity of debugging SDN might significantly hinder its adoption. Hence, distributing network functions over many components and layers, alongside the high-level abstractions made by SDN, complicate the debugging and troubleshooting process. Besides, leveraging a central controller to program SDN switches increases the probability of large-scale configuration errors. Locating those errors and discovering their root causes are usually painful and time-consuming.

Broadly speaking, SDN might be affected by two categories of bugs, which are functional bugs and performance bugs. A functional bug is often related to inconsistencies in the logic of SDN applications. It is typically noticed when a solicited network functionality becomes unavailable or leads to an unexpected result. A performance bug, on the other hand, preserves the correctness of the network while affecting its performance metrics. The impact on the network performance may indeed be substantial: latencies are increased, throughput is reduced, and valuable computation cycles are wasted. Performance bugs can occur for various reasons, such as buggy implementation, poor application design, workload mismatch, and misconfiguration. Unfortunately, it is hard to discover and diagnose them since they are not fail-stop faults.

Several approaches were proposed to meet the need for debugging SDN networks. Among them, runtime debugging (e.g., gdb-like tools) [2, 25, 91] and formal verification [92–94] are very popular. Based on these two approaches, many existing tools have proven their usefulness in diagnosing SDN bugs, particularly those affecting the forwarding logic (loops, black holes). Collecting performance statistics and monitoring the usage of SDN resources are an-

other effective approach to identify failing subsystems and network performance bottlenecks. Based on this approach, several monitoring frameworks were proposed [29–31, 33–35, 95–98], most of which are limited to the extraction of traffic statistics, such as per-flow packet and byte counts. The techniques they use to collect performance metrics vary, especially on the granularity of collected data and the incurred overhead. For example, to collect fine-grained monitoring data, many frameworks proposed in the literature [96, 98] integrate a data acquisition module into the switch itself. Other proposals [29–31] place such module in the controller node. Coarse-grained monitoring data is thus pulled from the forwarding devices using monitoring protocols such as sFlow [28] and SNMP (Simple Network Management Protocol). Often, techniques like sampling, aggregation, and load balancing are used to reduce the overhead caused by enabling these monitoring protocols.

We argue that the most significant limitation of the proposed frameworks is their inability to detect and diagnose network performance bugs. Tools based on runtime debugging and formal verification are more focused on checking the network correctness and the consistency of the forwarding logic of SDN applications. They are thereby ineffective in diagnosing performance bugs. Monitoring tools can help detect performance degradation, but it is unlikely that they can help to find the cause of the issue. The main reason behind this incapacity is that the collected data is high-level and, more importantly, incomplete due to the utilization of sampling and aggregation techniques.

To address the aforementioned shortcomings, this paper proposes a new monitoring and diagnostic tool for SDN networks. This tool provides full visibility into OpenFlow (OF) switches, hence giving the network operator the ability to diagnose performance bugs in the SDN data plane. The tool can also be leveraged to identify potential performance optimization opportunities. We chose OVS as a representative of OF-enabled switches, mainly because of its popularity. OVS is a product-level software switch that is widely deployed within many popular cloud platforms like Open Stack [99]. The forwarding decisions made by this switch are based on a set of rules, usually set up by a remote SDN controller, via the OF protocol. Though the implementation of our tool is bound to OVS, its design principles remain generic and could easily apply to any OF switch. Through userspace and kernel tracing, our tool gathers and analyzes OVS low-level performance data, to diagnose network bottlenecks and unexpected packet processing latencies. To sum up, the main contributions brought by this paper are the following:

1. Multi-level efficient tracing of OVS, and implementation of modules within a low-overhead tracer, LTTng, to collect kernel and userspace tracing data.
2. A new approach to build a unified stateful model from the collected event-based tracing

data. This model can easily be used by trace analyzers to evaluate the performance of OVS and to shed light on its internal mechanisms.

3. The implementation of several analyses, within an open-source performance analyzer, along with a set of interactive and ergonomic views. The main goal of these analyses is to pinpoint the causes of dataplane performance pathologies.
4. Validating the usefulness of our technique with many industrial-level use cases, showing how to leverage our views to troubleshoot SDN real-world performance issues.

The rest of this paper is organized as follows. Section 4.3 provides an overview of related work. Section 4.4 presents OVS and describes its architecture. Section 4.5 introduces the design and the implementation of our framework. Section 4.6 introduces four use cases to illustrate the usefulness of our tool and its ability to debug OVS performance issues. Section 4.7 evaluates and discusses the overhead induced by our framework. Finally, section 4.8 concludes the paper and provides directions for future investigations.

4.3 Related work

In this section, we look at debugging and troubleshooting tools that are dedicated to SDN.

The predominant approaches to diagnose bugs in SDN are runtime debugging and formal verification [3]. Hence, most SDN compatible debugging tools are capable of detecting various networking problems such as inconsistent flow rules, lack of reachability, and faulty device configuration. Ndb (network debugger) [2], for instance, is a network debugging tool built atop the NetSight framework [25]. This tool implements basic debugging primitives such as breakpoints and backtraces. When a packet matches with a breakpoint (i.e., a filter on the packet header), ndb constructs a backtrace which unveils the network path taken by this packet, header changes undergone along the path, the list of flow rules it has matched with, and the states of switches at the moment of forwarding it. A backtrace is constructed using a set of postcards, a unique packet identifier composed of a truncated copy of its header, the matching flow entry, the switch, and the output port. The information in a backtrace can easily be leveraged to isolate a faulty OF switch and diagnose bugs that affect the forwarding correctness. The main downside of ndb is its incapacity to diagnose performance bugs.

Besides runtime debugging tools, it is also possible to use formal verification techniques to detect bugs via the verification of the correctness of SDN components. Using these techniques, tools like Vericon [92], NICE [93], and Verificare [94] have proven their usefulness in checking inconsistencies in the forwarding logic of SDN applications (e.g., loops, black holes). For example, Vericon [92] takes as input the controller program, encoded in an imperative

language called Core SDN (CSDN), and a set of first-order formulas that express the network invariants along with the constraints on admissible topologies. Based on this input, the tool generates a first-order formula called Verification Condition (VC). This latter is then checked by an automated theorem prover allowing thus to either verify the correctness of the program or to pinpoint the events responsible for violating the safety or the topology invariants. The incapacity to verify liveness properties and to debug performance errors are the main limitation of Vericon. Besides, this tool presents the shortcomings inherited from the formal verification approach, notably the complexity of modeling large-scale networks.

The literature also reports a few profiling frameworks that are dedicated to SDN, such as nproof [25] and SPIRIT [24]. For instance, the latter uses a java profiling tool to profile the SDN controller and the applications running on top of it. Collected profiling data is analyzed to build the SDN applications' execution flows and find critical paths and hotspots that are the most critical in terms of performance. We believe that limiting the scope of detection to the performance bottlenecks that only affect the control plane (e.g., latency due to contention on access to controller resources) is the main weakness of this tool. Besides, the overhead it induces is considerable (up to 1200%), thus restricting its usage exclusively to test environments.

Diagnosing network performance problems can also be accomplished using monitoring tools. Most OF switches, like conventional ones, include support for network traffic monitoring protocols (i.e. Netflow [27], sFlow [28], IPFIX [100]). Several tools leverage those protocols to implement monitoring in SDN [29–32]. For instance, based on traffic statistics collected through sFlow, the authors in [29] propose a monitoring framework capable of detecting long-lived large flows (aka elephant flows). To reduce the overhead of stats collection, the framework simply relies on the adjustment of sampling frequency. Van Adrichem et al. [31] propose OpenNetMon, a system that monitors QoS parameters in OF networks to improve its traffic engineering. This system uses the OF protocol to pull per-flow counters from edge switches at an adaptive frequency. It then derives performance metrics such as throughput, packet loss, and delay. The coarse-grained monitoring performed by the proposed system makes it hard to debug performance issues and represents, therefore, its main downside.

Phan et al. [34] proposed SDN-Mon, a scalable monitoring framework for SDN. This framework allows management applications to define a variety of matching fields for the flows to be monitored. Several optimizations were introduced to consolidate further the scalability of SDN-Mon design, such as leveraging the computing capabilities of SDN switches to run most of the monitoring processes and to balance the monitoring workload between them [35] [97]. Limiting the collected data to only monitored flows reduces the chances of pinpointing the

root cause of potential performance issues.

Kučera et al. [33] proposed an approach to detect changes and anomalies (heavy hitters and superspreaders) in network traffic, entirely in the dataplane. Using a tree-like data structure and an algorithm called *Elastic Trie*, dataplane devices iteratively refine the IP prefixes that are the source of the anomaly/traffic pattern change. To avoid unnecessary controller-dataplane communications, a push-based mechanism is used to send notifications to the controller when specific conditions are met. While this approach focuses on the detection of abnormal network traffic, the focus of our proposal is more on diagnosing network performance issues.

As we have seen in this section, the state-of-the-art reports several debuggers and troubleshooting tools that are SDN compatible. However, most of the tools we reviewed so far present a common limitation: the incapacity to diagnose performance bugs. Hence, tools based on runtime debugging and formal verification approaches are focused on network correctness and thereby can not analyze SDN performance. SDN-based monitoring frameworks are very helpful in monitoring various network performance metrics but are not intended for use in real performance debugging. Most of these frameworks rely on existing monitoring protocols to query the switches and collect their traffic counters. While the collected data is useful in computing basic performance metrics, it is insufficient to conduct a detailed performance analysis or to diagnose performance bugs. Hence, the collected data is often high-level and inaccurate due to the utilization of sampling and aggregation techniques. Collecting detailed information about the OS, the hardware, and the running threads is essential to unveil performance pathologies and identify their root causes. Moreover, depending on the sampling frequency, enabling traffic monitoring protocols could be costly in terms of CPU and bandwidth overhead. This would reduce the chances of using SDN-based monitoring frameworks in production environments.

To the best of our knowledge, this is the first attempt to propose a troubleshooting tool that focuses on targeting performance bugs in the data plane. Our tool uses tracing techniques to gather low-level data, which is crucial for conducting an in-depth analysis capable of detecting unforeseen latencies and diagnosing their root causes. Hence, in this article, we present the design of our tool and propose an efficient multi-level tracing strategy to collect fine-grained performance data from SDN components, such as the software switches, and from their underlying system. Tracing is often used by developers and experienced administrators to monitor the execution of a running program for debugging and diagnostic purposes. Unlike other data collection techniques, the overhead that it incurs is minimal, which is essential to implement a low-overhead debugging tool.

4.4 Background

Open vSwitch [101] (OVS) is a multi-layer, open-source virtual switch, considered by the research community as one of the most successful implementations of SDN concepts. OVS is deemed to be a product-level virtual switch, thanks to its flexibility and its programmability through OF. It is hence widely used in various cloud platforms such as OpenStack and OpenNebula [102]. The forwarding decision-making process within OVS is determined according to the way it is configured. Three possible setups are identified: (1) OVS is connected to an out-of-the-box controller which uses OF to instruct it how to forward packets; or (2) OVS is installed in a standalone mode, and its forwarding decisions are made based on flow rules filled in by a command-line tool; or (3) OVS uses its internal control logic to perform a layer-2 packet switching like any conventional switch.

4.4.1 Open vSwitch Architecture

According to the OF switch specifications, an OF switch must include the following components: a set of ingress and egress ports, a pipeline of OF tables, and a secure channel to communicate with one or several remote controllers. The core function of an OF switch is to forward packets received at ingress ports to their intended output ports. Hence, for each received packet, the OF switch uses the pipeline of OF tables to search for a matching rule that defines the flow the packet belongs to, and to execute the actions specified for that specific flow.

The operating of an OF switch is thus based on a set of rules that are saved in memory within one or multiple flow tables. Each flow rule follows a match-action pattern and is therefore composed of two parts. The first part is defined by many values representing the condition that must be satisfied by a packet in order to be processed by that rule. These values correspond to specific fields in the packet header, alongside some metadata such as the ingress port, for instance. The second part is the actions set that defines what to do with the packet (forwarding to a specific port, modifying a given header field, cloning, dropping, etc.) [103].

The architecture of OVS complies with the specifications set by the Open Networking Foundation (ONF) [104] for OF switches as described in the first two paragraphs. Hence, as illustrated in Figure 4.1, OVS is composed of three major components: openvswitch, ovs-vswitchd, and ovsdb-server. The openvswitch is a kernel module that is responsible for fast flow-based switching. The second and third components are userspace daemons. The ovs-vswitchd is responsible for controlling the switch and implementing the OF protocol.

The ovsdb-server is a lightweight database server that uses the OVS Database Management Protocol (OVSDB) for configuring OVS. In the following paragraphs, we briefly explain the operating of the first and second components since they are the focus of this paper.

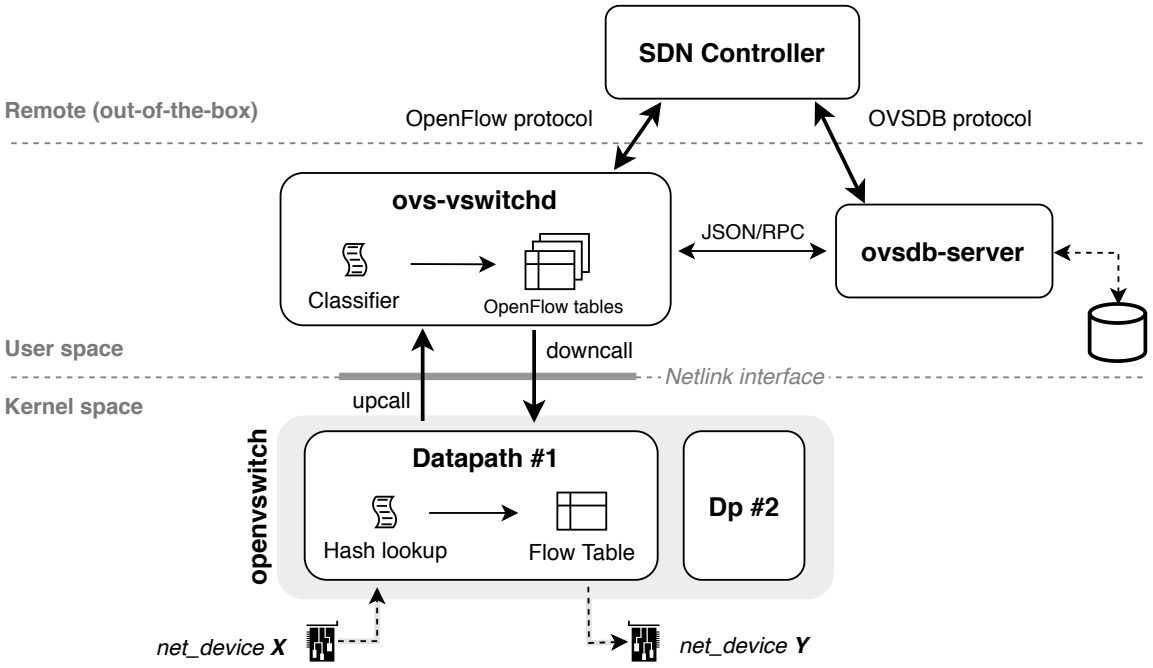

Figure 4.1 Architecture of Open vSwitch

Kernel module

The kernel module, which represents the forwarding plane of OVS, implements one or many datapaths. A datapath is a flow-based forwarding switch associated with one or multiple vports (virtual ports) [103]. Each datapath uses a single flow table for matching and processing incoming packets. The entries of a datapath flow table are generated and reactively installed by the ovs-vswitchd daemon. Thus, as the datapath classifies and processes incoming packets, its flow table is populated in response to flow lookup misses. This technique guarantees that only the most relevant and used flows are cached by the datapath.

Over time, many optimizations were carried out to the flow caching design [101]. For instance, wildcard matching in kernel allowed to reduce OVS memory consumption and to speed up the process of packet classification. The megaflow cache is thus implemented as a single flow lookup table. The entries of this table may include wildcard fields aggregating fine-grained connections into larger flows. Since priority is not supported in the datapath flow table, the forwarding efficiency is increased because a lookup is finished as soon as a match is found.

The recent versions of OVS support two levels of flow caching. The first level is the microflow cache, and the second level is the megaflow cache. The microflow cache (aka EMC for Exact Match Cache) is implemented as a fixed-size hash table mapping the microflow entries to their corresponding entry in the megaflow table. A microflow entry matches with all the packet header fields that are supported by OF. Thus, when a first packet matches an entry in the megaflow table, a new entry is inserted in the microflow table. This entry allows subsequent packets belonging to the same connection to be quickly directed to the right entry in the megaflow table.

Userspace daemon

Unlike the kernel module, the userspace daemon uses a pipeline of flow tables. The flow rules within these tables are OF rules and, therefore, they support priority. When none of the cached rules could be matched against a packet, the datapath sends an upcall to the ovs-vswitchd daemon to ask for instructions on how to handle it. Most of the tasks executed by this latter are performed by two categories of threads: (1) handler threads and (2) revalidator threads.

A handler thread is responsible for the processing of upcalls issued by datapaths. When it succeeds in matching the packet enclosed in an upcall with the highest priority flow rule, it sends back the action set to the datapath. Usually, the matching flow rule is also sent so that the datapath source of the upcall can handle the subsequent packets. If the handler thread fails to find a matching flow, the received packet is either forwarded to a controller or dropped if none is available. To sum up, a packet is said to take the "fast path" if the datapath successfully matched it with one of the rules in its flow table. Conversely, a packet is said to take the "slow path" if the datapath hands it to the userspace daemon.

On the other side, a revalidator thread fetches the flows from the datapaths flow tables and validates them. Revalidation cycles are periodically executed. A flow can be invalidated, and therefore deleted from the datapath flow table, for several reasons. For instance, a flow can be evicted after an inactivity timeout or when its semantic has been altered after a configuration change (e.g., a virtual port was deleted by the user).

4.4.2 Kernel/userspace interface

The OVS kernel and userspace modules have been developed based on a loosely coupled approach in order to facilitate their independent update and distribution [1]. A well-defined

1. The kernel module has been shipped in Linux mainstream since the version 3.3 [103]

interface was then set up between the two modules. This interface allows any version of the userspace daemon to interoperate with any implementation of the datapath. For instance, DPDK (Data Plane Development Kit) [1] can be leveraged to implement userspace datapaths, thus enabling OVS to run entirely in userspace.

In Linux, the kernel datapath can be controlled by ovs-vswitchd through the Generic Netlink protocol. Hence, it exposes four protocol families (datapath family, virtual port family, flow family, and packet family) [103]. For the three first families, a set of four commands are supported: NEW, DEL, GET, and SET. These commands allow to create, delete, retrieve, and modify an entity within a given family. For the packet family, there is just one command allowed which is EXECUTE. At the end of an upcall handling, the userspace daemon leverages this latter command to make the datapath execute a set of actions on a packet.

A kernel datapath uses several Netlink sockets to communicate with userspace threads. Hence, when a vport is created, this latter may specify one or many Netlink sockets PIDs (Port IDentfiers). The datapath uses these PIDs to send upwards the upcalls that are triggered by the reception of one or multiple packets on that port.

4.5 Design of the proposed framework

The contribution proposed in this paper is a framework capable of evaluating the performance of OVS and discovering the root causes of traffic bottlenecks. Although our tool implementation is bound to OVS, its design principles are general and could apply to any other OF switch. Figure 4.2 depicts the architecture of our framework, mainly composed of two subsystems. The first subsystem deals with the collection of meaningful performance data from virtual switches. The second subsystem is responsible for leveraging this data to elaborate relevant performance analyses and export their results through interactive views. These views should allow the practitioner to detect and explain the performance pathologies of OVS.

4.5.1 Data collection

There are many approaches to collect performance data from a running program. One of these approaches is to harness the information exported in the log files. The advantage of using this approach is that no additional tool is required. However, the existence of numerous formats, structuring the content of logs, alongside the high-level nature of the logging information itself, complicates its usage by an efficient and fine-grained performance analysis tool.

Software tracing, on the other hand, represents an efficient approach to collect low-level

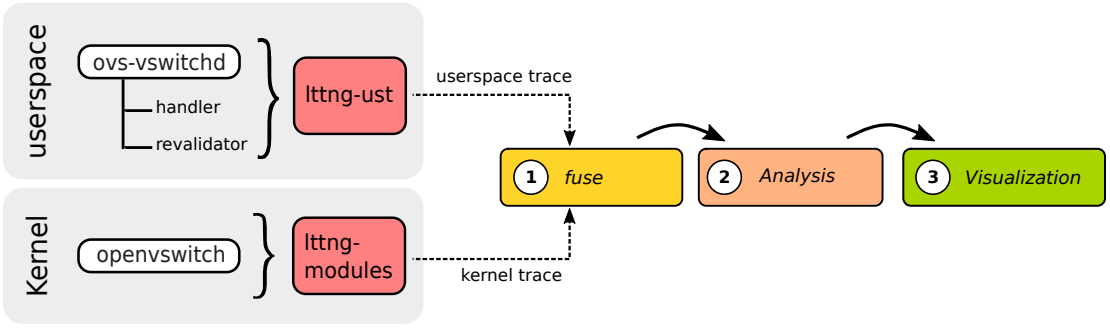

Figure 4.2 Architecture of the proposed framework

data. The analysis of the execution trace of a program can help diagnose different types of programming faults and performance issues, and also unveil their root causes. It can also be leveraged to explain the behavior of a complex system and comprehend the interaction of its internal mechanisms. Unlike logging, tracing is well suited to record a large amount of fine-grained performance data while incurring a low overhead on traced systems [105].

Since we needed an efficient and inexpensive method to collect the performance data required for our analyses, we chose to use tracing. There is a wide variety of tracing tools for almost every modern OS, such as LTTng [106], SystemTap [83], and Ftrace. After a thorough review of the characteristics of each tracer, we adopted LTTng. This tool can trace at both kernel and userspace levels at a low-overhead cost, an essential requirement for production environments.

Collecting relevant performance data using a tracer requires the insertion of probes at strategic points within the application source code. An efficient approach to determine where to place the instrumentation code is to identify all the complex mechanisms by which the application operates. In our case, we identified two major mechanisms in OVS: (1) A mechanism responsible for the forwarding of packets and (2) another for the flow revalidation. Both of these mechanisms require the interaction of OVS userspace threads with the kernel module.

Following the instrumentation step, we used lttng-modules to trace the openvswich kernel module and LTTng-UST to trace ovs-vswitchd along with the threads it spawns. We explain in the next paragraphs the details of the mechanisms mentioned above. We also show in tables 4.1 and 4.2 a subset of the tracepoints that were used to trace them.

Packet forwarding

Figure 4.3 shows how a datapath interacts with handler threads to forward packets. It also shows the most relevant events that are triggered following a packet reception. As explained earlier, a packet takes the fast path (blue line) when its matching flow rule has already been cached in the datapath. If it is not the case, it takes the slow path as indicated by the red lines.

To handle an incoming packet, a datapath first calculates a lookup key using several fields in the *sk_buff* data structure. Then, it computes a masked version of this key using a flow mask. A mask depicts the bits within the packet key that need to be compared with those within the flow rules keys to find a match.

To locate the right flow mask, the openvswitch kernel module first checks the microflow cache (*mask_cache* data structure). It is a per-cpu n-way associative cache that maps packets to probable masks based on their *skb_hash* header field. This latter is a packet header field used by the Linux flow dissector to store a hash value calculated from several other header fields. Examples of such fields are IP source and destination addresses, and UDP/TCP source and destination ports. Hence, OVS uses the hash value stored in the *skb_hash* field as a coarse flow identifier.

When the usage of the microflow cache results in a successful hit, the event *ovs_emc_cache_hit* is triggered, and a masked version of the lookup key is computed. The computed masked key is used to perform a table hash lookup to find a matching flow rule. If this is not the case, openvswitch iterates through an array of masks in the megaflow table and performs sequential masking of the lookup key. Each masked lookup key is used to perform an independent hash flow lookup. When a masked lookup key computed via a megaflow mask matches with a flow key, the event *ovs_megaflow_cache_hit* is triggered. When a matching entry is found in the datapath flow table, the event *ovs_flow_match* is fired, whether the mask was in the microflow or the megaflow cache.

In case none of the flow keys matched with the lookup key, the datapath encapsulates the received packet in an upcall sent to userspace, thus triggering an *ovs_dp_upcall* event. There are many cases in which an upcall is sent to the userspace: (1) the flow lookup results in a miss; (2) one of the actions of the matching flow rule is not supported by the datapath; (3) the packet header corresponds to an unsupported network protocol; (4) forwarding the packet to the controller is explicit in the flow rule. Consequently, a different upcall type is used for each case (e.g., MISS_UPCALL and USER_UPCALL).

A datapath is connected to ovs-vswitchd via several Netlink sockets. The handler threads,

spawn by ovs-vswitchd, iterate through these sockets to read and handle the received upcalls. When an upcall is read by one of these threads, the event *upcall_read* is triggered. Packets are processed by handler threads in batches. The processing of an upcall could result in two types of commands: a command to execute the actions of the matching flow rule and another to cache the new flow in the datapath (case of MISS_UPCALL type upcalls). Commands related to the processing of multiple upcalls are sent to the datapath in a single downcall, thus triggering the event *downcall_transact_multiple*.

When the datapath receives the instructions from the handler threads, it processes them, usually causing the execution of the flow actions and the caching of the new flow rule in the flow table. This would trigger the events *ovs_packet_cmd_execute* and *ovs_flow_cmd_new* respectively.

To be able to compute some metrics related to the queuing and processing of upcalls, our analyses require that each upcall be identified uniquely. We could not use the address of the *sk_buff* data structure as an identifier because the whole packet is copied in the upcall. Moreover, the packet may be sent to a remote controller if the handler threads fail to find a matching flow rule. To solve this issue, we implemented a kernel module to tag the packets that are enclosed in the upcalls. This module hooks into the function *ovs_dp_upcall()* and assigns a number to the *mark* field of the *sk_buff* data structure, just before the upcall is sent. Hence, we rely on this mark field to identify an upcall. We also leverage it to track the packets processing flow when it is split between the kernel and userspace.

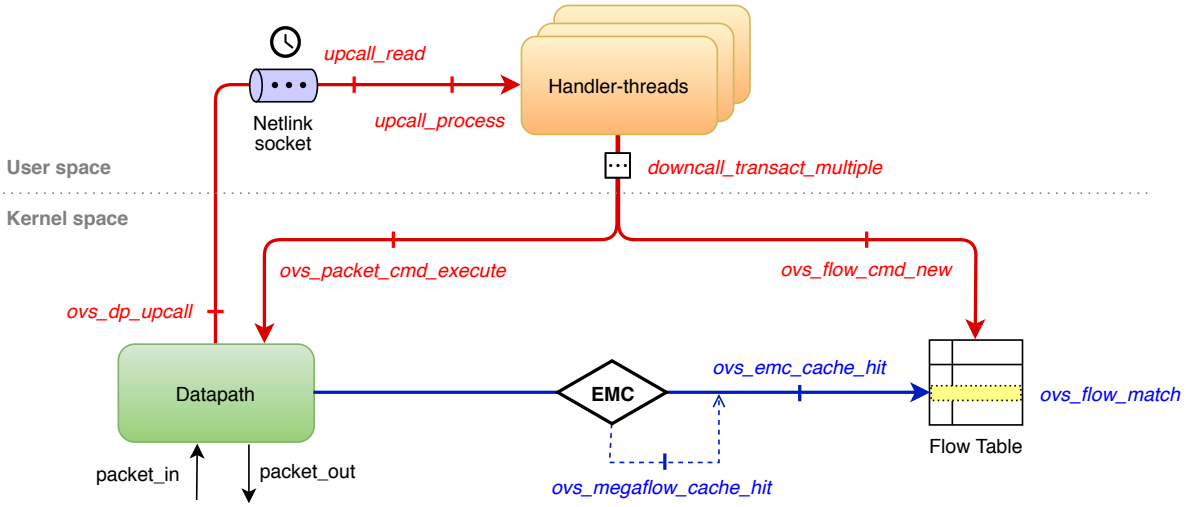

Figure 4.3 Packets processing paths within OVS and their corresponding tracepoints

Table 4.1 Description of a subset of events that are fired during packet forwarding

Tracepoint	Type	Description
ovs_emc_cache_hit	kprobe	This event indicates a hit in the microflow cache. This latter maps a packet with a probable mask based on a computed flow identifier.
ovs_megaflow_cache_hit	•	Following a microflow cache miss, a full lookup is performed by the datapath to match a packet with a rule.
ovs_flow_match	•	A packet has matched with a rule within the datapath flow table. The main attributes of this event are : dp_name, skbaddr, and ufid.
ovs_dp_upcall	•	An upcall was issued by a datapath. The main attributes are : dp_name, skbaddr, upcall_type, and packet_key.
ovs_packet_cmd_execute	•	The datapath executed the actions received by the userspace module in response to an issued upcall.
ovs_flow_cmd_new	•	A flow was installed by a datapath in its flow table. The main attributes of this event are : dp_name, ufid, and sw_flow_key.
upcall_read	uprobe	An upcall was read by one of the handler threads.
upcall_process	•	This event indicates the start of an upcall processing. Handler threads read and process upcalls in batches.
upcall_user_exec_start	•	Indicates the start of the execution of requested userspace actions.
upcall_user_exec_end	•	Indicates the end of the execution of requested userspace actions.
queue_amsg_controller	•	A packet was sent to the controller via an asynchronous message (following a table miss for instance).
downcall_transact_multiple	•	A datapath was asked by a handler thread to execute a set of operations. Each operation can either be the execution of a set of actions or the installation of a new flow.

Flows revalidation

Revalidator threads are responsible for two tasks : (1) ensuring that flows cached by datapaths remain valid and (2) their number is within the acceptable limits. To accomplish the first task, revalidator threads periodically fetch flow statistics through requests sent to datapaths. The event ovs_*flow_cmd_dump* is fired when a given datapath dumps the statistics of a set of flows to userspace. The statistics include various parameters such as the inactivity period of a given flow rule and the number of packets that matched with it. Hence, based on this information, a revalidator thread can ask the datapath to delete this flow if its idle time exceeds a timeout (the default max-idle parameter is 10 seconds). The event *ovs_flow_cmd_del* is fired when a flow is deleted on the datapath side.

To accomplish the second task, revalidator threads continuously monitor the number of flows cached by the datapath. To keep their number below the flow-limit parameter, the flow timeout could be decreased to 100 ms. Moreover, in case the number climbs above twice the flow-limit value, then all datapath flows are deleted as an emergency measure.

To delimit the period during which a flow might be invalid, the time required to perform a revalidation cycle is checked by revalidator threads. When this time exceeds a limit [2], the flow-limit parameter, as defined by the user, is overwritten and decreased. On the contrary, when it becomes less than 1 second, the flow-limit value increases slowly. Hence, using this regulation mechanism, revalidator threads adjust their load to keep the revalidation time-bounded. The events *revalidation_start* and *revalidation_stop* denote the beginning and the end of a revalidation cycle respectively.

4.5.2 Analysis and Visualization

To use our framework, we first need to initiate a tracing session to collect OVS performance data. Since two instances of LTTng are required to trace OVS modules, the events triggered by the kernel module are recorded in a trace file, and those triggered by userspace processes in another trace file. To ease the development of adapted analyses, it is vital to fuse the content of these two trace files into a single unified trace file. Fortunately, there is no need for reordering the recorded events since they are all timestamped with the same monotonic clock. An efficient approach to organize the gathered tracing data and to prepare them for analysis is to build a data model. This latter is a multidimensional data structure which stores processed data and exports the states of the traced system in a way that facilitates the development of automated analyses.

2. 1.3 seconds for the current OVS version (2.11.9)

Table 4.2 Description of a subset of events that are triggered during flows revalidation

Tracepoint	Type	Description
`ovs_flow_cmd_del`	kprobe	A flow was deleted by a datapath.
`ovs_flow_cmd_get`	•	Specification and statistics of a flow were sent to the userspace module.
`ovs_flow_cmd_dump`	•	The datapath dumped the whole content of its flow table to userspace.
`revalidation_start`	uprobe	A revalidation cycle was started by revalidator threads.
`revalidation_stop`	•	Indicates the end of a revalidation cycle.
`flow_limit_update`	•	Revalidator threads overwrote the flow-limit parameter that is set by the user.

The step following the data model construction is designing relevant analyses capable of pinpointing performance issues and shedding light on the operation of the traced system. A practical approach for building such analyses is to derive performance metrics by semantically correlating tracing events. Thus, we implemented in Trace Compass [87] a data analyzer that uses our data model to calculate the aforementioned metrics. Trace Compass is an open-source trace analyzer that provides several prebuilt analyses compatible with the CTF trace format. We present the design details of our data model and data analyzer in the following paragraphs.

Data modeling

Tracers are very useful in debugging complex applications, providing insights into their runtime behaviors, and identifying the root causes of potential performance bugs. However, since these tools are designed for high throughput tracing, the size of collected data may quickly become extremely large, increasing thus the cost of their processing. Fortunately, Trace Compass provides several facilities to prepare the tracing data for efficient analyses based on filtering, highlighting, and multi-level abstraction facilities.

We designed an interval-time tree-based data structure to store the attributes and states of monitored virtual switches. Those attributes and states are derived from the events contained in the trace files. Hence, when a trace file is loaded in Trace Compass, the useful information from events is extracted and transformed to populate this data structure incrementally. Data stored in this tree-shaped data structure can rapidly be retrieved in logarithmic time. Using

such a data structure gives the practitioner the ability to choose the granularity of the required performance metrics. It also enables him to apply the analyses on any time range. Indeed, organizing the tracing data with this data structure helped us develop our custom analyses and their underlying views.

As shown in Figure 4.4, our data model is built upon the data structure mentioned earlier. The branch entitled "Datapaths" is used to save the information related to flows processing like flow installation, deletion, and statistics dumping. As explained in the next section, the content of this branch is useful, for instance, to compute the size of the datapath flow table. The flow_key field, a concatenation of metadata and OF supported packet header fields, is used to identify flows when installed in the datapath flow table. However, when they are dumped or deleted, we use their corresponding unique flow identifier (UFID). The branch "Handlers", on the other hand, depicts the processing of upcalls by handler threads.

Data Analysis

Data analysis in our framework is driven by computing adapted performance metrics. We designed generic performance metrics, such as utilized bandwidth and packet rate, which can be useful in pinpointing bottlenecks and performance issues. The other metrics are specific to OVS and can be used to discover the causes of detected problems.

Datapath cache hit rate As explained earlier in this section, flow lookup efficiency is greatly influenced by the rate at which the first-level and second-level caches are hit. Accordingly, a cache usage metric would be an interesting indicator to evaluate the packet forwarding performance achieved by OVS. Hence, performance degradation can manifest through a low utilization of these caches, which might be caused, for example, by the reception of a high number of short-lived connections. In our analyses, we propose to compute the microflow and megaflow cache utilization based on the tracepoints presented in Table 4.1. For instance, to measure the percentage of microflow cache usage, the following formula is used:

$$Utilization_{EMC} = \frac{nbHit_{EMC} \times 100}{nbHit_{EMC} + nbHit_{Megaflow} + nbMiss_{Upcalls}} \tag{4.1}$$

Number of flows cached by the datapath Crossing the OVS kernel-userspace boundary is costly because it requires the issuing of expensive system calls and also because of the delays due to upcalls queuing and processing. Thus, the forwarding performance of OVS is seriously affected when datapaths frequently need the assistance of the userspace daemon.

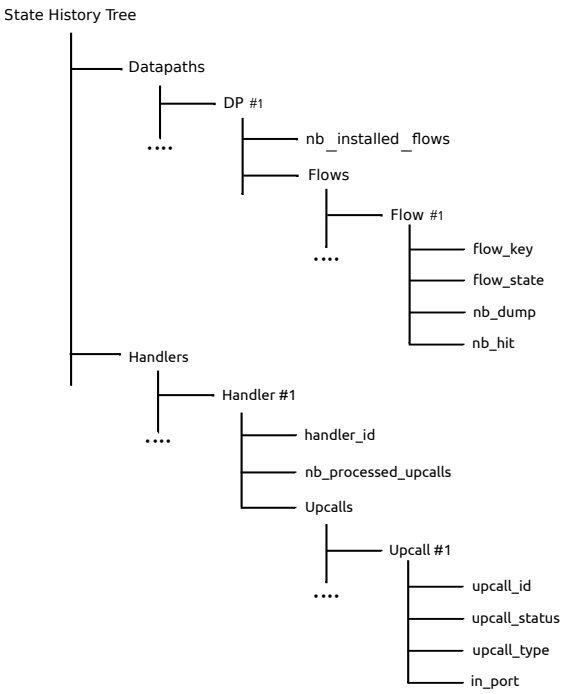

Figure 4.4 Excerpt of the interval-time tree-based data structure on which our framework is based

As explained in section 4.5.1, packet processing in userspace is done in these situations : (1) the datapath flow lookup results in a miss; or (2) one of the matching rule actions is not supported by the datapath; or (3) sending the packet to the userspace is explicit in the action set of the matching rule. Unlike in the second and third cases, it is possible to mitigate the number of upcalls caused by lookup misses by fine-tuning OVS configuration (e.g., increasing the size of the megaflow table, reducing flows maximum idle time). With a reduced number of upcalls, the overall performance of virtual switches would increase. Hence, knowing the number of cached kernel flows can help identify and diagnose some performance issues. For instance, a high frequency of miss upcalls can be explained by the fact that the datapath flow-limit value has been reached.

Upcalls average waiting latency This metric, alongside the size of the upcalls waiting queue, is leveraged to assess the load of handler threads. The waiting time of an upcall starts from the moment it is sent via a Netlink socket up to the time a handler thread reads it. To be able to compute this metric, each upcall must be uniquely identified.

Utilized bandwidth and packet rate metrics Utilized bandwidth (UB) represents the number of bits sent or received by a port at a given point in time. The packet rate represents the number of packets sent or received by a port per second. Combined with the packet loss rate, these metrics can be a good indicator of network congestion. Using the kernel events *net_if_receive_skb* and *net_dev_xmit*, we calculated the reception and transmission packet rate/UB per port. For instance, we calculate the UB using the formula below. The values B_{t2} and B_{t1} denote the byte counts at event timestamps t2 and t1 respectively, and T denotes the observation time interval.

$$UB = \frac{(B_{t2} - B_{t1}) \times 8bits}{T} \tag{4.2}$$

Packet rate per flow According to our experiments (see the use case presented in section 4.6.2), OVS does not ensure equal performance for all the flows. We thus propose to compute the metric per-flow packets processing rate. A given packet may match a flow rule either in the datapath megaflow table or in the OF tables. In the latter case, the userspace threads send the actions of the matching flow rule to the datapath through the Netlink interface. Exceptionally, a packet can match with more than one flow rule in the case of recirculation. This latter is a mechanism envisioned by OVS to allow a packet to re-enter the datapath packet processing path after being modified. For example, recirculation is required to decode IP header fields that lie beneath an MPLS header (Multiprotocol Label Switching).

4.6 Use Cases

This section validates our approach and shows how our tool can diagnose the performance bugs caused by misconfiguration and workload mismatch. The topology of the SDN network in our test environment is composed of 3 virtual switches (S1, S2, and S3), all managed by OVS in the same host machine (Figure 4.5). We did not choose a complex test deployment for simplicity and because the topology and the size of the network do not impact the performance and capabilities of the proposed approach. Since our focus is on diagnosing dataplane performance issues, we run OVS according to a standalone setup. Therefore, instead of relying on a remote controller, we used the command-line tools provided by OVS to configure the switches and manage their OF rule tables. We configured the switches differently in each use case, either to highlight a given OVS internal mechanism or to identify a specific performance issue. We connected two Docker containers to each switch to generate and consume network traffic. Table 4.3 shows the configuration setup used to run our experiments. The last use case corresponds to a real scenario in which we used our tool to troubleshoot a performance

bug in a data center SDN.

Table 4.3 Hardware and software experimental configurations

Hardware Environment		**Software Environment**	
CPU	Intel Core i7-7500U CPU @2.70GHz	OS	Ubuntu 18.04 (Kernel 4.19.5)
#Cores	4	OVS	version 2.11.9
GPU	HD Graphics 620 (Kaby Lake)	LTTng	version 2.10.8
RAM	16 GB	Trex	version 2.41

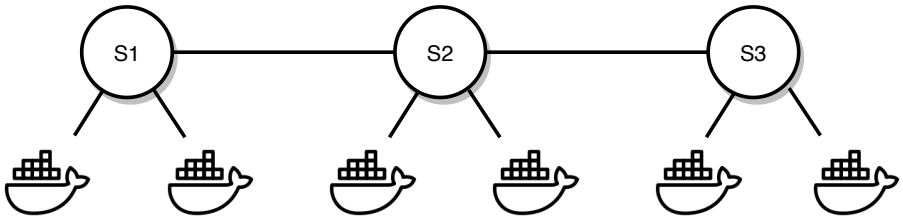

Figure 4.5 SDN topology of our test environment

4.6.1 OVS performance tuning

Most systems are likely to experience performance degradation when they are exposed to increased loads. Performance tuning refers to a set of techniques used to identify and remove bottlenecks from a given system. In this context, profiling tools can help pinpoint which part of the system is causing the bottleneck. Once this latter is identified, a fine-tuning of the configuration is often sufficient to address the issue.

In this use case, we show how our tool can be leveraged to diagnose potential bottlenecks in OVS. Hence, in the first experiment, We configured OVS with two handler threads and one revalidator thread. We set the datapath flow table maximum size (flow-limit parameter) to 200k so that all the created flow rules can be cached simultaneously. We used the command line to install 200k flow rules in S1 and Trex traffic generator to generate 200k different UDP streams by varying packet IP source addresses and UDP destination ports.

When traffic was issued at a rate of 500 kpps, performance degradation was noticed. To identify its root cause, we used our tool to check the upcall issuing rate at the port source of the traffic. As we can see in Figure 4.6, during the whole experiment, this rate was high (almost 30k upcalls per second). This diagnosis suggests a problem on the side of the datapath since it kept sending upcalls to the userspace.

To understand the reason behind the high upcall issuing rate, we used another view depicting the time needed by revalidator threads to perform revalidation cycles. The same view also shows the maximum limit for the flows that the datapath can cache (which corresponds to the flow-limit configuration option). As explained earlier, the revalidator threads adjust the value of this option according to the revalidation time. Hence, the revalidator threads try to keep the time needed for revalidation between 1000 ms and 1300 ms. However, when this time exceeds 1300 ms, the revalidator threads diminish it by decreasing the value of the flow-limit option.

As can be seen in Figures 4.7 and 4.8, when the revalidation time surpassed 1300 ms, the flow-limit value quickly dropped from 200k to 56k. As a result, the datapath named ovs-system deleted several flows from the cache in order to adjust the total number of flows authorized to be cached to that value (Figure 4.9). After that, the flow-limit value has slowly increased up to 100k by the end of the experiment. Overwriting the user-defined maximum authorized number of flow rules forced the datapath to cache less than half of the switch flow rules, and to send an upcall each time it received a packet causing a flow lookup miss.

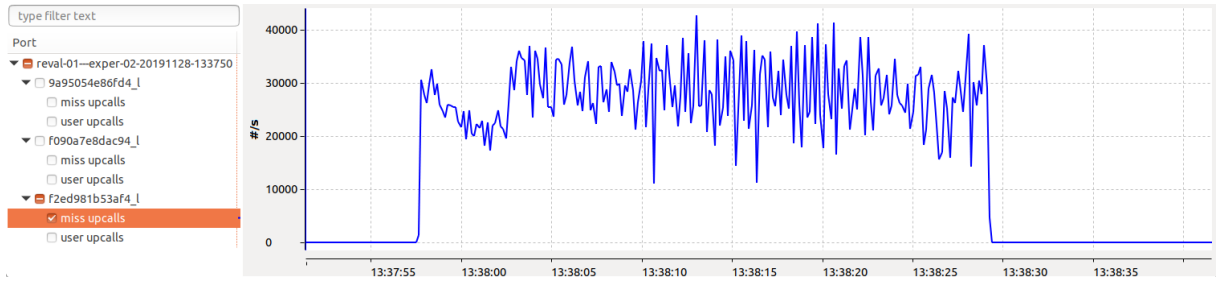

Figure 4.6 Rate of upcalls triggered by the port issuing the traffic

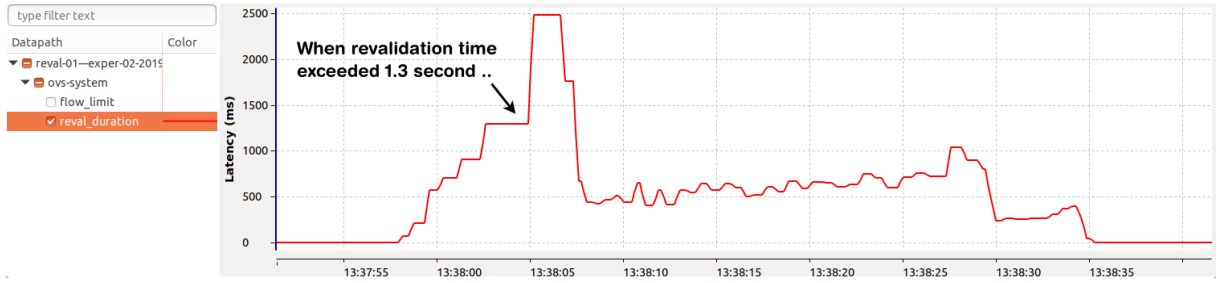

Figure 4.7 Time spent in revalidating the flows cached by the datapath

To sum up, the incapacity of revalidator threads to cope with the load imposed by our setup negatively impacted the forwarding performance. When we configured OVS to spawn an additional revalidator thread, we solved the issue, and the datapath could then cache all the

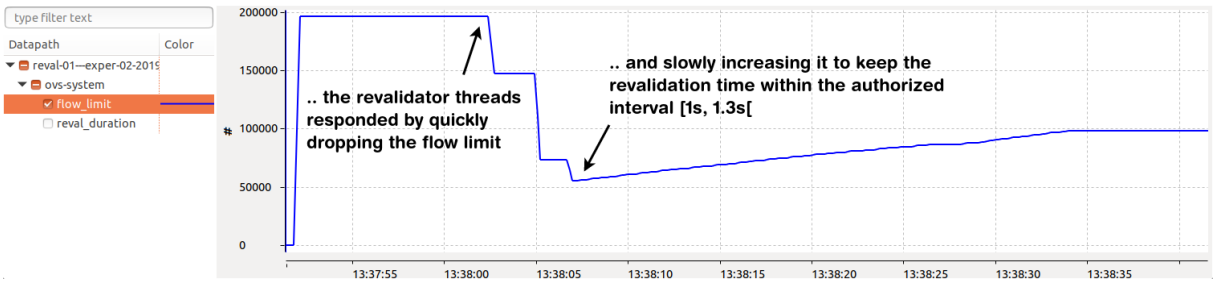

Figure 4.8 Flow-limit value calculated by revalidator threads based on the time spent in revalidation cycles

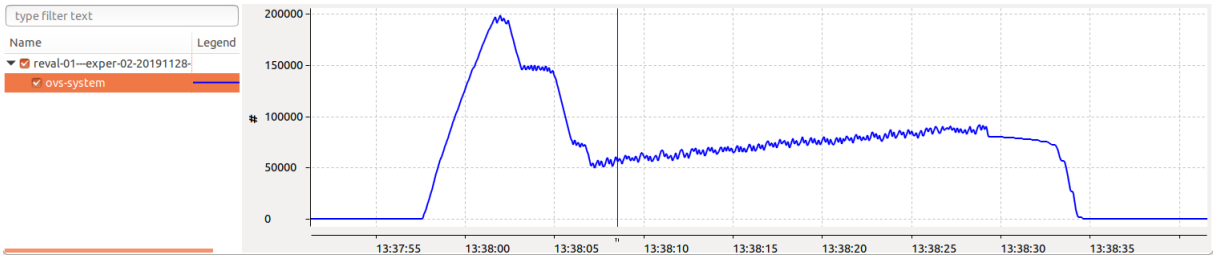

Figure 4.9 Number of flow rules cached by the datapath

created flow rules (Figure 4.10). OVS, during its setup, determines the number of handler and revalidator threads to spawn based on the number of available CPUs. This method does not consider other factors, such as the pattern and rate of the traffic, the type of machine on which OVS runs (VM or bare metal), and the case where some CPUs are isolated. In this use case, our tool enabled us to adjust the number of threads according to how they responded to their loads.

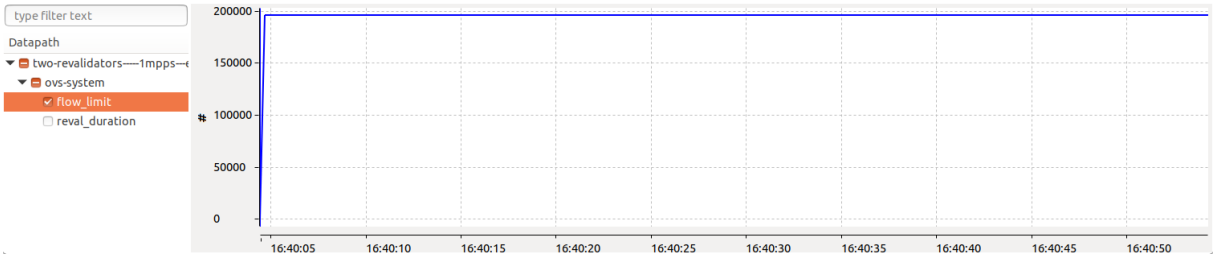

Figure 4.10 Variation of the flow-limit value after fixing the issue (adding another revalidator thread)

4.6.2 Overcommitment of the datapath cache

This use case aims to study to what extent the processing of a given flow by OVS can be impacted by the presence of other flows. In this experiment, we configured OVS to spawn two handler threads and two revalidator threads. We configured the maximum size of the datapath megaflow table to the minimum (1000 flows). We used the command *ovs-ofctl* to install 5k flow rules in switch S2. We utilized the Trex traffic generator to generate 5k continuous traffic streams with a transmission rate of 500 kpps. The experiment, which lasted 200 seconds, was traced and analyzed by our performance analysis tool.

Figure 4.11 shows one of the views provided by our tool. This view displays the metric per-flow packet rate. As we can see, the packet forwarding performance was not uniform for all the flows. As depicted in this figure, the packet rate of flow #10 (red color) was, at the beginning (for almost 3 seconds), approximately equal to the packet rate of flow #9 (blue color). However, the forwarding performance related to flow #10 dropped afterward. This performance gap is confirmed by OVS statistics, accessed through the command *ovs-ofctl dump-flows*, indicating that 3950 packets were handled by flow #10 while 5997 packets were handled by flow #9.

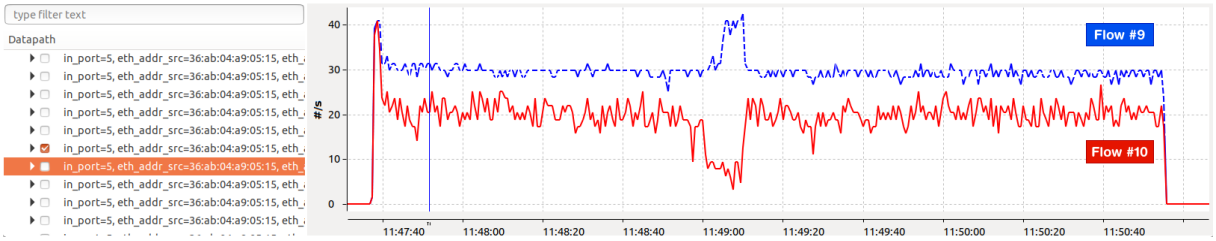

Figure 4.11 Difference between the packet rates of flow rules

To investigate the cause of the performance gap between the two flows, we used another view that shows the number of flow rules cached in the datapath. As we can see in Figure 4.12, the number of cached flows fluctuated for almost 3 seconds and then stabilized at 1518 flows. This means that 3482 flows were not cached, and the datapath was forced to pass their matching packets to the userspace to handle them (Figure 4.13). The delays involved in queuing the upcalls, processing them by handler threads, and sending back their corresponding action sets to the datapath to execute them, were the cause behind the drop of packet rates of the flow rules in question.

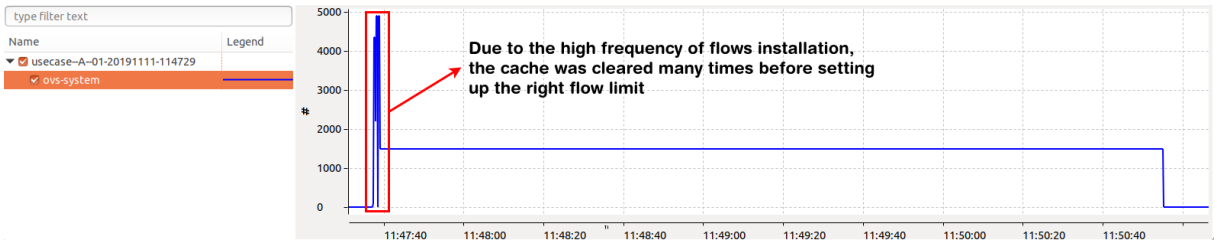

Figure 4.12 Number of flow rules cached by the datapath

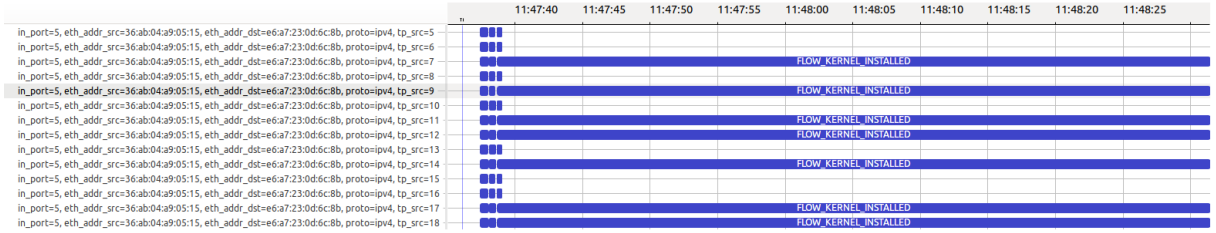

Figure 4.13 Caching duration of installed flow rules

4.6.3 Kernel/userspace flow key mismatch

Over time, the OVS flow key has continually evolved to support new network protocols and drop support for obsolete ones. So often, a mismatch between the flow key formats of kernel datapaths and the userspace is susceptible to occur. For instance, a flow key mismatch could happen when different versions of kernel and userspace modules are used. It could also happen when the userspace module has full support for a given network protocol, whereas the kernel module has partial support for it.

To overcome this problem and ensure forward and backward compatibility, whenever the datapath passes a packet to the userspace module, it also passes the flow key extracted from that packet [107]. Similarly, the userspace module extracts its own flow key and compares it against the one received from the kernel datapath. One of these three cases could be the result of this comparison: (1) the flow keys of the kernel and the userspace are similar; (2) the kernel flow key encompasses more fields than the userspace flow key; and (3) the userspace flow key encompasses more fields than the kernel flow key.

The first and second cases do not pose any problem in terms of performance since the userspace will always be able to install the corresponding flow rule in the datapath. However, setting up a flow rule becomes impossible in the third case because the datapath would be incapable of decoding the extra flow key fields. Thus, every packet received by the datapath and matching its flow key representation will be encapsulated into an upcall sent to

the userspace. As we will show in this use case, this would have a severe impact on the forwarding performance of OVS.

Both kernel and userspace modules of the OVS configuration used in our experiments support the MPLS protocol. Nevertheless, the kernel has support for a single MPLS label, whereas the userspace can process a label stack depth up to three labels. In the first experiment, we used Trex to generate network traffic where each packet has double MPLS headers. We fixed the transmission throughput at 200 kpps to minimize eventual packet losses. We also set up a flow rule in the S3 switch to forward received packets to the next port after popping one MPLS header. We generated traffic for 10 seconds and leveraged our tool to analyze the performance of the experiment. The view in Figure 4.14 shows the packet rate at which the NIC of the target container was receiving the traffic. Based on this view, we can effortlessly identify a performance loss because of the important difference between the packet rates at reception (approximately 120 kpps) and at transmission (200 kpps). To confirm these results, we ran a second experiment where the packets of generated traffic have only a single MPLS header. The packet rate at reception was approximately equal to 200 kpps.

The significant gap between the forwarding performance measured in the two experiments is a good indicator of a severe performance bottleneck. The reason behind the performance loss in the first experiment is explained by the high rate of user upcalls issued by the datapath, as illustrated by figures 4.15 and 4.16. The datapath sends user upcalls in many cases. For example, when the datapath does not support one of the actions of a cached flow rule or is unable to match specifically enough with a given flow key. Figure 4.17 shows that these upcalls were issued because of a flow key mismatch.

4.6.4 Microflow and megaflow cache

In this use case, we wanted to show if we can use our tool to optimize OVS performance. OVS is deployed within our internal data center to manage its underlying networking infras-

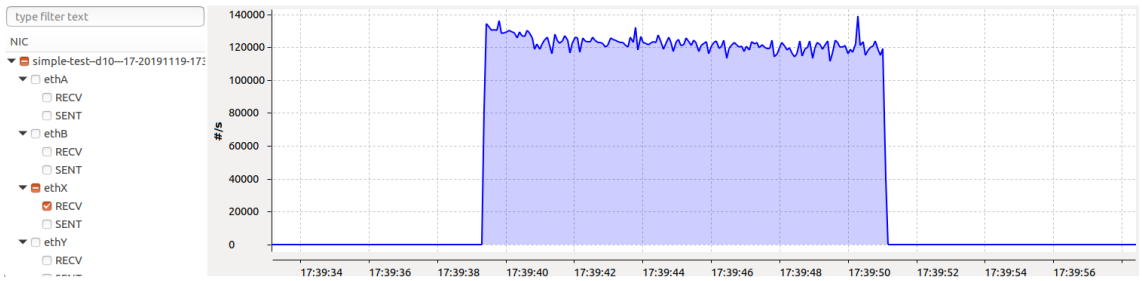

Figure 4.14 Packet rate at the reception for the traffic generated with double MPLS headers

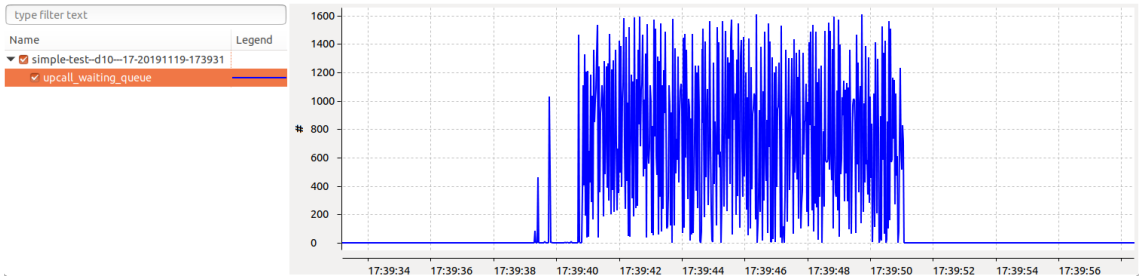

Figure 4.15 Waiting queue length of handler threads

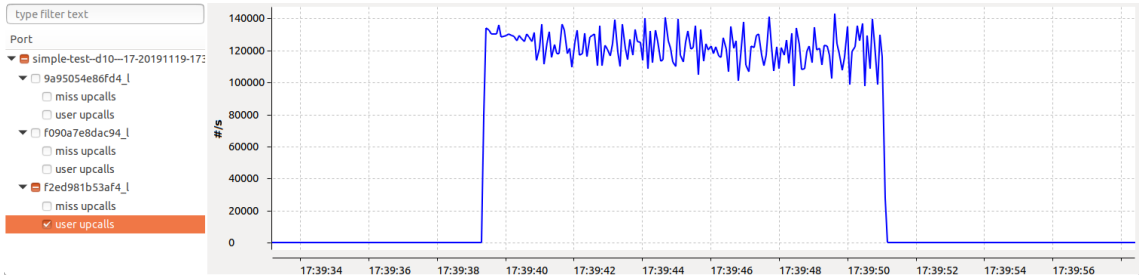

Figure 4.16 User upcalls issuing rate per port

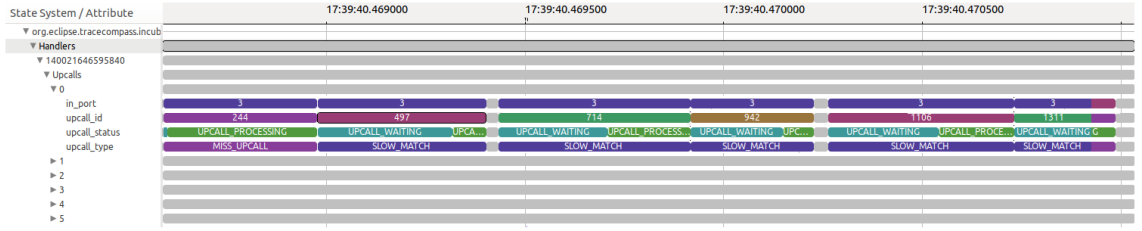

Figure 4.17 User upcalls are issued because of a flow key mismatch

tructure. To achieve this goal, we traced OVS operating in some host machines, and then we used our tool to collect and analyze the traces. The analyses conducted led us to interesting findings, such as the one shown in Figure 4.18. This latter depicts a low utilization of the datapath first cache.

As explained earlier in section 4.4, the kernel datapath uses two levels of caching to implement the fast path: the megaflow cache and the microflow cache. While the former implements wildcard matching based on the TSS (Tuple Space Search) algorithm, the second cache, on the other hand, was introduced to reduce the overhead of the sequential masking and lookups implied by the megaflow usage. The datapath uses the field *rxhash* of the *sk_buff* data structure as an index to access the microflow cache. The hash value stored in this field is calculated by the flow dissector of the Linux kernel. Various networking subsystems use this value to filter packets (e.g., firewalls) or assign them to specific flows.

There are a few reasons which could explain why the microflow cache utilization is low. For instance, the percentage of hits on this cache would decrease if the number of masks used by the active flows becomes considerable. This is due to the limited size of the microflow cache (256 entries per CPU). In our current use case, this justification does not stand because, according to our analyses, the number of active flow rules that are used by most of the traced virtual switches is low (around 200). Most of the cached flow rules were used for VLAN and MPLS traffic processing. By digging more in-depth into the data exported by our tracepoints, we discovered that most of the hash values used to hit the microflow cache were null. This result indicates that the kernel flow dissector failed to calculate hash values for packets having either MPLS or VLAN outer headers. When we checked the source code of the OS kernel (version 4.1.45) on top of which OVS was running, we found that the flow dissector does not implement the calculation of a flow identifier, except for UDP and TCP transport protocols. Installing a recent kernel (higher than 4.2) on the host machines solved the problem because its flow dissector supports a wider variety of networking protocols such as MPLS, VLAN, and ARP.

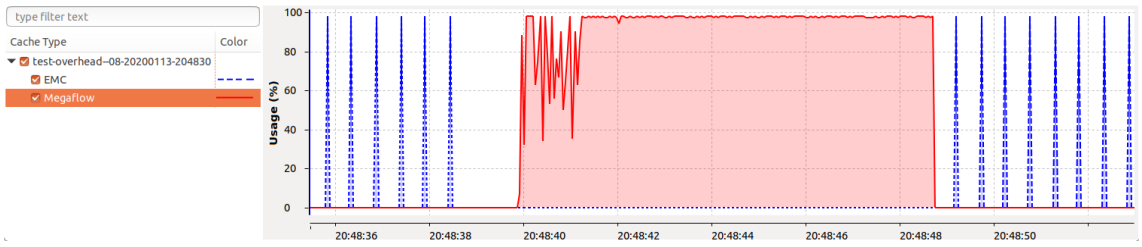

Figure 4.18 EMC and megaflow cache usage

4.7 Overhead Analysis

It is crucial for a performance analysis tool to minimize its overhead in order to reduce the probing effect. Indeed, a high overhead may negatively impact the performance and the behavior of monitored systems. In the present section, we assess and analyze the overhead induced by our framework. We first study the tracer overhead alongside the size of trace files. Subsequently, while not as critical, we also evaluate the analysis time. This latter represents the time required by Trace Compass to load trace files and to build analysis data structures.

To adjust the load of our tracer, we defined two tracing modes: lightweight tracing mode and exhaustive tracing mode. The goal of the first tracing mode is to detect potential performance issues, while the goal of the second is to troubleshoot the root causes of these issues. Hence, in the first tracing mode, only userspace tracepoints are enabled, and in the second tracing mode, both userspace and kernel tracepoints are enabled. We configured the tracer to activate the lightweight tracing mode and immediately switch to the exhaustive tracing mode if a performance issue is detected. Based on empirical analysis, we developed heuristic techniques to automate the transition between the two tracing modes. For instance, when the issuing rate of a specific event (i.e., *upcall_receive* event) goes beyond a threshold, the exhaustive tracing mode is activated. Our tool uses the detailed data recorded during the exhaustive tracing session to investigate performance issues.

To estimate the overhead imposed by the tracer, we ran a benchmark and measured the forwarding performance of OVS, with and without tracing. In this benchmark, which lasted 10 seconds, we varied the bit rate at which Trex generates traffic. We also varied the number of flow rules matched by the traffic. We set the maximum idle time of flow rules to 5 seconds and the configuration option flow-limit to 200k. Finally, we configured the daemon ovs-vswitchd to spawn two handler threads and two revalidator threads.

Hence, Table 4.4 presents how the trace size and analysis time vary depending on the transmission (TX) bit rate. Table 4.5 shows the tracing overhead while varying the transmission

Table 4.4 Trace size and analysis time according to transmission bit rate

TX Bit Rate (MB/s)	Lightweight Tracing		Exhaustive Tracing	
	Trace Size (MB)	Analysis Time (s)	Trace Size (MB)	Analysis Time (s)
320	12	3	368	49
80	12	3	131	31
20	14	3	85	23

Table 4.5 Impact of tracing overhead on OVS forwarding performance

TX Bit Rate (MB/s)	#Flows (K)	RX Bit Rate (MB/s)	Lightweight (MB/s)	Lightweight Overhead (%)	Exhaustive (MB/s)	Exhaustive Overhead (%)
320	100	319.78	319.61	0.05	315.26	1.41
	10	319.85	319.68	0.05	314.22	1.76
	1	319.94	319.82	0.04	315.2	1.48
80	100	79.91	79.86	0.06	79.1	1.01
	10	79.96	79.92	0.05	79.27	0.86
	1	79.99	79.95	0.05	79.33	0.83
20	100	19.95	19.94	0.05	19.85	0.50
	10	19.84	19.76	0.04	19.75	0.45
	1	19.99	19.91	0.04	19.91	0.40

bit rate and the number of flows. We estimate the overhead by measuring the reception (RX) bit rate when tracing is disabled and enabled. As a first observation, we can effortlessly note that the analysis time is tightly related to the trace size. The larger is the trace size, the more events it contains, the longer is the time required to process them.

It is worth noting that the tracing overhead is very low for lightweight tracing and is independent of the transmission bit rate. We can also remark that this latter does not have a noticeable impact on the trace size. In our opinion, this makes sense because most userspace events are leveraged to track the activity of handler and revalidator threads. As explained earlier, handler threads stay idle unless there are pending upcalls. In our case, upcalls were only issued at the beginning of the benchmark in response to flow lookup misses. On the other hand, revalidator threads perform their tasks periodically, and their states are only influenced by the number of cached flows and the time required to revalidate them. This explains why the trace size was not impacted by the bit rate, as illustrated by Table 4.4.

In the exhaustive tracing case, we can remark that the transmission bit rate significantly influences the overhead and trace size. We can explain this by the fact that many kernel events are triggered by the reception of packets on datapaths ports. Thus, the frequency of kernel events issuing is tightly correlated to the rate at which the traffic generator transmits packets. Considering a network with a heavy traffic load, the rapid increase in the trace size becomes a real problem that might potentially hamper the scalability of our scheme.

To solve this issue, one might suggest enabling the exhaustive tracing mode only when a forwarding performance problem is observed. However, the downside of this strategy is that many tracing events, which might be valuable for the accuracy of our troubleshooting

analysis, are prone to loss when the tracing modes are switched. The solution adopted by our framework consists of running jointly lightweight and exhaustive tracing sessions, though, without saving the output of the tracer on disk. This solution takes advantage of the LTTng snapshot feature that enables the tracer to record tracing events in memory and dump its memory buffers' content only upon request. Hence, the snapshots taken during the exhaustive tracing sessions are uploaded to a central traces collection machine for offline troubleshooting analysis. Using this architecture, collecting tracing data for our framework will not be costly in terms of disk space and execution time.

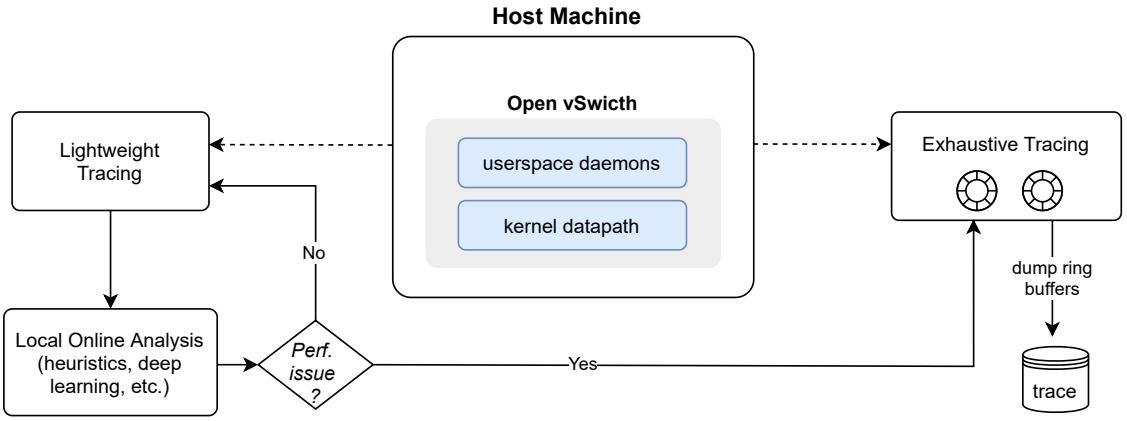

Figure 4.19 Adopted tracing architecture

In summary, our cost evaluation study shows that the impact of tracing on OVS performance is negligible (<2%). However, the cost associated with the storage of trace files remains a limitation. The key reason behind the increase in trace size is the high frequency of kernel events. To circumvent this issue, our approach leverages the LTTng snapshot feature to record tracing events only during periods of interest.

4.8 Conclusion

SDN is a network architecture that is backed by an active community and many major networking vendors, such as Cisco and VMware. This architecture fills the new applications' growing need for higher bandwidth, more flexibility, and better resilience against faults. The principal idea on which SDN is based is the separation of the network control from the forwarding infrastructure. Hence, in an SDN architecture, the applications use the API provided by one or multiple controllers to express their networking policies. The controllers, on the other hand, translate their demands into low-level configurations sent to the forwarding

infrastructure. As a result, the performance of both control and data planes is highly susceptible to programming errors and misconfiguration. Unfortunately, current debugging tools are unable to detect and diagnose performance bugs efficiently.

In this paper, we propose a diagnostic and troubleshooting tool that can diagnose the bottlenecks of SDN and help optimize its performance. The tool was tested in a production environment and was able to detect real-world networking issues. Thanks to this new tool, we could effortlessly discover the root causes of these issues. According to our study, the overhead imposed by our tool is low (<2%), thus enabling its use in production environments. As future work, we plan to extend this framework by supporting OVS-DPDK. Many studies have shown that the performance of OVS increases when DPDK is used as a userspace datapath [108]. Our tool would be more versatile if it could evaluate the performance of an SDN network, where many types of datapaths (kernel, userspace, and hardware) are used.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena and EfficiOS.

CHAPITRE 5 ARTICLE 2: PERFORMANCE ANALYSIS OF DPDK-BASED APPLICATIONS THROUGH TRACING

Authors

Adel Belkhiri, Martin Pepin, Mike Bly, and Michel Dagenais
Department of Computer and Software Engineering, École Polytechnique de Montréal, Montreal, H3T 1J4, Canada

E-mail : {adel.belkhiri, michel.dagenais}@polymtl.ca

Submitted to Journal of Parallel and Distributed Computing

5.1 Abstract

The Data Plane Development Kit (DPDK) is an efficient framework developed based on the kernel bypass approach to accelerate packet processing in userspace. This framework includes many libraries and Poll Mode Drivers (PMDs) that are optimized for maximum network performance. Several studies have demonstrated that porting existing data plane applications to DPDK can considerably increase their performance. Unfortunately, kernel bypassing makes traditional network monitoring and debugging tools obsolete. Therefore, considering the complexity of networking software, detecting unexpected behaviors, and diagnosing its root causes became a tedious task. The literature reports a few tools intended for debugging DPDK application bugs, but they are mostly ineffective against performance bugs.

In this paper, we propose a tracing-based performance analysis framework that is dedicated to DPDK applications. The framework provides many analyses enabling the practitioner to gain insights into the application internals and diagnose performance bugs. We use an efficient approach to collect tracing data from DPDK libraries and drivers, with a very low overhead (<0.2%). Our framework correlates the data collected to build a data model that illustrates the states of the monitored application. This model is leveraged to generate adapted performance metrics and to display them in synchronized graphical views. Our tool was tested with many use cases and was able to identify and characterize various bugs such as performance bottlenecks.

5.2 Introduction

The last decade has seen an explosion of the network traffic volume, triggered by the increasing use of live streaming applications, such as video-on-demand, voice-over-IP, and online gaming applications. To satisfy the bandwidth requirements for these applications, companies have been relentlessly trying to improve the performance of their networking infrastructures by taking advantage of the networking hardware capacity. Hence, modern Network Interfaces Controllers (NICs) bandwidth has considerably increased in recent years, thanks to many advances in communication protocols and hardware technologies. For instance, 100-gigabit Ethernet is commonly deployed in today's data centers. Nevertheless, studies from the state-of-the-art have shown that the built-in kernel networking stack is incapable of scaling up with the speed of those NICs. They reported several limitations that cause the kernel network stack to barely handle 1 Mpps traffic rate per CPU core [109]. We can summarize these limitations as follows. First, the operation of the traditional network stack is based on system calls and interrupts, which require costly context switches. Second, because the NIC driver runs in kernel space, there is an unnecessary copy of packet data to and from an application's buffers. Third, processing packets individually does not ensure efficient cache memory usage. Lastly, because the *sk_buff* data type is a very large general-purpose structure, its allocation in memory is time-consuming.

Over time, there have been multiple proposals for addressing those limitations that one can group into two main approaches. The first approach focuses on the improvement of the Linux network stack performance and its socket API. For instance, the eXpress Data Path (XDP) [39], a high-performance data path implemented within the kernel network stack [1], represents an interesting proposal based on this approach. On the other hand, the second approach adopted more radical measures, bypassing the Linux Kernel and using a parallel network stack. Few proposals from this approach, like those based on Remote Direct Memory Access (RDMA) [110], implemented the latter within the kernel while others, DPDK [1] and Netmap [40] for instance, moved it to userspace. Implementing a userspace network stack has been an attractive idea for many stakeholders for two reasons. First, it ensures a high level of flexibility since it does not require modifying the kernel source code, which saves the time and effort needed to convince the kernel maintainers of the effectiveness of this approach. Second, it enables the building of network stacks tailored to specialized applications, thereby eliminating the overhead associated with the traditional general-purpose network stack.

Among all the proposals of this latter approach, DPDK turned out to be particularly effective in circumventing the limitations of the OS in-kernel network stack. DPDK is a complete and

1. Since kernel version 4.8

efficient framework implemented based on the kernel bypass approach to accelerate packet processing in userspace. This framework comprises various libraries and network interface drivers that are optimized for maximum network performance. Considered as the de-facto standard of accelerating packet processing in high-performance networks, DPDK has fostered the trend of transforming networks from proprietary devices to general-purpose platforms and moving network functions from dedicated hardware to software implementations. Hence, several Network Function Virtualization (NFV) and Software Defined Networking (SDN) solutions have leveraged DPDK for acceleration. For instance, studies show that porting Open vSwitch (OVS) [5] to DPDK can improve switch throughput by a factor of seven [111].

Despite all the benefits that we mentioned above, porting an application to DPDK poses significant challenges, especially when it comes to debugging an unexpected behavior. Hence, when an application bypasses the OS kernel, it also bypasses all the isolation and security mechanisms the latter provides. More importantly, it makes traditional network management, monitoring, and debugging tools obsolete. For instance, it is not possible to use the system command *ifconfig* to access the statistics of NICs managed by DPDK PMDs. Besides, DPDK offers the developer various design and implementation choices. For example, to develop an application with complex pipeline processing stages, the developers can make use of a single or many threads. They can also leverage various libraries and scheduling mechanisms (e.g., pipeline and eventdev libraries). Wrong design decisions or misconfigurations can potentially lead to unexpected application behavior or performance bottlenecks. Thus, diagnosing application errors is often painful and time-consuming, especially in the case of performance bugs. Compared to functional bugs, performance bugs are hard to discover because of their non-failure semantics. Hence, they can cause severe performance losses (i.e., increased latency, lowered bandwidth, etc.) while maintaining the network connectivity.

Several approaches and tools have been proposed in the state-of-the-art to diagnose the performance bugs of data plane applications. Among them, a few are explicitly dedicated to DPDK applications, such as VTune Amplifier [112]. The latter is a performance profiler for multi-threaded applications and systems. This tool includes some analyses to enable the diagnosis of specific performance issues in DPDK applications. Nevertheless, it is commercial and closed-source, and the analyses it provides only cover a limited subset of DPDK libraries. Aside from VTune Amplifier, there exist open-source tools such as FlowWatcher-DPDK [113] and DPDKStat [114]. The operation of these tools is based on extracting high-level performance data from the DPDK application and computing statistics that describe the traffic processed by the application. Those statistics, such as per-flow packet count and per-flow interleaving gap, can be leveraged to monitor the network traffic and detect potential performance degradations. However, they do not enable the diagnosis of the root causes of

the detected performance issues.

In this paper, we propose a new performance analysis tool intended for DPDK applications. The tool implements many post-mortem analyses providing adapted performance indicators and full visibility of the DPDK application states. Gaining in-depth insights into the operating of DPDK applications allows the practitioner to discover potential performance bugs and pinpoint their root causes. In order to enable the diagnosis of complex performance problems, our tool computes relevant performance metrics and derives the states of DPDK software components from runtime data. We leveraged various tracing techniques to collect low-level performance data from the different threads and modules of the target application. Hence, we instrumented several libraries and drivers from the DPDK framework. The main advantage of this approach is that it does not require the instrumentation of the application source code. Often, access to the source code is prohibited for security reasons or when the application is proprietary. To trace the execution of DPDK applications, we used Linux Trace Toolkit Next Generation (LTTng) [81], a low-overhead tracer for Linux. Nevertheless, it is possible to use any tracer that supports the Common Trace Format (CTF) since our analyses are implemented to process traces in that format.

The contribution of this paper and its underlying work is fourfold: **Firstly**, we propose an efficient approach to collect performance data from DPDK applications. This approach is based on the instrumentation of DPDK libraries and drivers, thus making the tracing process transparent to the application. **Secondly**, we developed an effective stateful data model based on various data abstraction techniques in order to structure the collected tracing data and prepare it for analysis. **Thirdly**, we leveraged a popular open-source performance analyzer to implement several performance analyses that are specific to DPDK applications. Those analyses aim to help the practitioner discover performance pathologies in DPDK applications by deriving and analyzing the states of their components and by computing relevant performance metrics. **Finally**, we implemented a set of ergonomic time-synchronized graphical views to enable the practitioner to shed light on the operation of the monitored applications and to examine their behavior interactively.

The rest of this manuscript is organized as follows. Section 5.3 looks at the main debugging and diagnostic tools for DPDK applications. Section 5.4 overviews the architecture of DPDK and presents its main components. Section 5.5 describes our methodology to analyze the performance of DPDK applications and introduces the design and implementation of our framework. Section 5.6 presents two representative use cases that validate our approach and illustrate the usability and effectiveness of our framework for diagnosing performance bugs in DPDK applications. Section 5.7 evaluates the overhead incurred by our framework and

examines various ways to decrease it. Finally, section 5.8 concludes the paper and outlines future work.

5.3 Related Work

DPDK is a fundamental key enabler for leveraging commodity hardware to process networking workloads. It is thus widely used in cloud computing infrastructures to accelerate the performance of VNF and SDN components. Hence, in this section, we relate our contribution to the existing works focusing on debugging and monitoring VNFs and SDN.

The literature reports a limited number of diagnostic and debugging tools that support DPDK-based applications. Among those tools, we can mention VTune Amplifier, a commercial proprietary profiling tool developed by Intel for x86 based machines. This tool aims at analyzing the performance of serial and multi-threaded applications and pinpointing their bottlenecks. Hence, it provides various analyses such as hotspots analysis, concurrency analysis, and Input/Output analysis. In particular, the latter, intended for analyzing the performance of I/O intensive applications, allows the collection of some statistics from the polling threads of DPDK applications. Unfortunately, although the I/O analysis can help pinpoint various performance issues in DPDK applications, it only covers a few DPDK libraries. Furthermore, unlike our tool, VTune Amplifier is closed-source and platform-dependent.

There exist alternative solutions, such as FlowWatcher-DPDK and DPDKStat, which allow the network operator to collect statistics from DPDK applications and to monitor various data plane performance metrics. Nonetheless, those tools do not provide any diagnostic or troubleshooting capabilities. Hence, the data they collect do not enable conducting thorough performance analyses since it is high-level. It might, though, aid in pinpointing a performance degradation. FlowWatcher, for instance, is a lightweight DPDK-based traffic monitoring tool. This tool is tuneable and can provide fine-grained statistics at both packet and flow levels. It can also compute complex performance metrics such as the per-flow interleaving gap, a valuable metric to assess a flow burstiness. The operation of this tool is based on the usage of the RSS hash, computed by the NIC at the hardware level, to aggregate received packets into flows. FlowWatcher-DPDK presents two main limitations. First, it uses a unique hard-coded flow identification which limits its ability to monitor complex protocols. Secondly, it can not track bidirectional flows as it only monitors inbound traffic.

For SDN, runtime debugging [2, 25] and formal verification [92, 94] represent the main approaches for diagnosing networking bugs. Nevertheless, these two approaches are not very useful in diagnosing performance bottlenecks. Ndb (network debugger) [2], for instance, is a

debugging tool that is implemented based on the first approach. It leverages breakpoints and backtraces to collect a detailed report about the traffic forwarding in the network. The report includes the path taken by a given packet, the list of matching flow rules, and the changes in routers states. The information contained in the report enables the network operator to identify any issue that may affect the forwarding correctness. Vericon [92] is another tool that diagnoses inconsistencies in the forwarding logic of SDN applications by utilizing formal verification techniques. The operation of this tool is based on deriving a set of first-order formulas, called Verification Condition, and checking them with an automated theorem prover to pinpoint the events responsible for violating network invariants. The main limitation of Vericon is its inability to verify large-scale networks and to diagnose performance bugs. Aside from the tools mentioned above, there are a few tools that leverage profiling and tracing tools to diagnose performance bugs in SDN [24, 115]. For instance, the authors in [115] propose a diagnostic tool that can debug various performance bugs in SDN. This tool uses tracing to collect performance data from OVS switches and derives adapted performance metrics. The main limitation of this tool is that it can not diagnose performance bugs that affect userspace datapaths such as those instantiated by OVS-DPDK.

The literature also points out many tools dedicated to diagnosing performance bugs in VNF infrastructures. NFVPerf [42] and PerfSight [43] are examples of such tools. NFVPerf is an online monitoring tool capable of identifying hardware and software performance bottlenecks in NFV systems. The tool runs as a part of OpenStack, a popular cloud management system. It leverages passive traffic monitoring to calculate the load of VNF components based on throughput and per-hop delay metrics. It considers a VNF component to have a performance bottleneck if its load exceeds its previously learned capacity. The tool presents two significant limitations. First, it is incapable of performing fine-grained bottleneck diagnosis within VNFs. Secondly, it is unable to operate in case the network traffic aspects are unknown. PerfSight, on the other hand, is a diagnostic tool that is designed for debugging the performance issues of software data planes. The operating of this tool is based on collecting low-level statistics from various networking elements in the data plane and analyzing them to identify a bottlenecked NF. The main shortcoming of this tool is its incapacity to diagnose performance bugs in userspace data planes (e.g., DPDK data planes).

To sum up, the literature reports few tools intended to analyze the performance of DPDK applications. For instance, VTune Amplifier provides analyses capable of diagnosing certain performance issues of DPDK applications. Nevertheless, this tool is commercial and closed-source, and the analyses it provides only cover a limited subset of DPDK libraries. Other tools, such as FlowWatcher-DPDK and DPDKStat, calculate basic performance metrics that essentially monitor network traffic patterns (e.g., flow size). Those metrics are calculated

from coarse-grained data and, therefore, they do not enable the practitioner to conduct a root cause analysis in case performance degradation occurs. For SDN, tools based on runtime debugging or formal verification are focused on verifying the network's correctness and invariants. Hence, these tools can debug numerous networking issues, such as the lack of reachability and inconsistent flow rules, but they cannot diagnose network performance bugs. As for the tools dedicated to diagnosing performance problems in VNF infrastructures, our study shows that most of them can locate a performance bug at the NF level but fail to explain its cause.

This paper proposes an open-source framework that leverages userspace tracing to analyze the performance of DPDK applications and to shed lights on their behavior. This framework is particularly efficient in diagnosing the performance bottlenecks of monitored applications and revealing their unforeseen latencies. The design and implementation details of the proposed framework are presented in the following sections.

5.4 DPDK Overview

DPDK [1] is an open-source software project that aims to enable the development of high-performance network data plane applications. DPDK is developed based on the kernel bypassing approach, which consists in offloading packet processing from the OS kernel to userspace applications. Although there exist several solutions that use various kernel bypassing techniques to speed-up packet processing (e.g., netmap [40], and StackMap [116]), DPDK remains the most popular [117]. Compared to similar solutions, DPDK encompasses many radical optimization techniques in order to increase the packet processing speed. We can summarize the most relevant of those optimizations techniques in the following points: (1) Running NIC drivers in userspace instead of kernel space to avoid unnecessary packet copy; (2) Adopting the polling mode in retrieving packets from NICs ring buffers to reduce the processing latencies; (3) Using huge pages to reduce the overhead related to the Translation Lookaside Buffer (TLB) misses; (4) Support for the CPU affinity and cache alignment to increase the cache hit rate.

At the architecture level, DPDK is mainly composed of libraries and NIC drivers called PMDs. We describe those components in the following paragraphs.

5.4.1 Poll Mode Drivers (PMDs)

Initially, NIC drivers used to rely on interrupt mechanisms to process inbound traffic. Hence, when the NIC receives one or several packets, an interrupt is triggered, and the kernel is urged

to process them. The interrupt-driven mechanism incurs low overhead and achieves a reasonable latency when the rate of packets reception is low. However, when the packets arrival rate is considerable, like in high-speed networks, the processing performance is significantly decreased. The reason for this performance loss is twofold. First, the overhead of this mechanism quickly becomes significant due to the high frequency at which interrupts are triggered. Secondly, pathologies such as receive livelock become more likely to occur. A receive livelock happens when userspace applications are denied access to CPU cycles because the interrupt handlers are executed with a higher priority.

Many interrupt mitigation techniques were designed to circumvent the limitations of the interrupt-based approach, such as interrupt coalescing, polling, and NAPI (New API). Interrupt coalescing is a feature implemented in recent network controllers to moderate interrupts firing. The technique behind this feature involves starting a timer upon packets arrival and then delaying to trigger an interrupt until the timer is expired or a specific number of packets are received.

The polling approach consists in disabling the interrupts related to the arrival of packets and steadily checking the NIC reception (RX) buffer for new packets. When the polling rate is high, this mechanism decreases packet processing latencies, and, therefore, improves network throughput. However, since it is not possible to guarantee the presence of packets in the RX buffers of the NIC at all times, a high number of CPU cycles are vainly wasted. This mechanism is therefore only recommended for high load networks.

NAPI is an API that has been integrated into the Linux kernel to improve the performance of its networking subsystem. NAPI adopted a hybrid mode mixing interrupts and polling. When the NIC receives a burst of packets, it fires an interrupt, and the interrupt handler activates the NAPI subsystem. The latter handles the received packets using a poll function when their number exceeds a limit. The NIC driver registers this poll function during initialization. NAPI deactivates the RX interrupts coming from the NIC so that packet processing can proceed without interruptions. It also reactivates those interrupts and ends the polling when all the received packets are handled. Otherwise, it reschedules the poll function again.

Drivers provided by DPDK are called poll mode drivers because they poll network devices for packets. The reason for DPDK to adopt the polling mode is to achieve the lowest possible latency and to maximize throughput. Since DPDK is a kernel-bypass solution, its drivers run in userspace instead of kernel space. By doing so, the cost of context switching between userspace and kernel space is avoided. Thus, a PMD often requires a kernel driver like *vfio_pci, igb_uio*, and *uio_pci_generic* to initialize the hardware device. A NIC must be explicitly bound to one of the drivers mentioned above in order to be managed by DPDK.

5.4.2 DPDK Libraries

DPDK offers dozens of libraries that developers may use to implement high-performance networking applications. Those libraries can be grouped into classes, depending on the functionalities they provide. For instance, we can mention the core libraries that provide most applications with basic facilities (i.e., *librte_eal*, *librte_mempool*, and *librte_mbuf*). We can also mention the packet framework libraries which enable the development of complex pipeline-based packet processing applications (i.e., *librte_pipeline*, *librte_table*, and *librte_port*). We present in the following paragraphs a subset of the libraries that we instrumented.

EAL Library

DPDK is a multi-platform framework that supports a wide range of hardware architectures, NICs, and operating systems (i.e., BSD, Linux, and Windows). To reduce the complexity of developing sophisticated data plane applications, DPDK provides the Environment Abstraction Layer (EAL). The latter is a library that abstracts the underlying hardware and software environments (e.g., OS, compilers) by hiding the specifics of each. It also represents a generic interface through which DPDK applications can access low-level system resources.

For instance, EAL creates a set of execution units called logical cores (or lcores) as an abstraction for the available CPU cores. In Linux environments, the lcore is a Pthread thread that is pinned to a specific CPU core. As shown in Fig. 5.1, a DPDK application is executed by two categories of lcores: one primary lcore and many secondary lcores. The role of a primary lcore is to control and synchronize the operating of secondary lcores. It, therefore, executes the function *rte_eal_init()* that performs most of the initialization and, in particular, instantiates secondary lcores on remaining CPU cores. The primary lcore assigns workloads to secondary lcores by issuing commands (function pointers) to them via a set of pipes. Each secondary lcore performs an active wait for these commands via the execution of the function *eal_thread_loop()*. Once a command is received, the secondary core executes it, sends back its return value to the primary lcore, and enters the waiting state again.

A secondary lcore can also be used as a service lcore to run one or many services. That is a short routine that is executed repeatedly. When the hardware lacks a specific feature, the developer can implement the missing feature and run it as a service. DPDK itself provides many functionalities as services such as the Eventdev software scheduler. Services that are run by a service core are scheduled based on a round-robin algorithm.

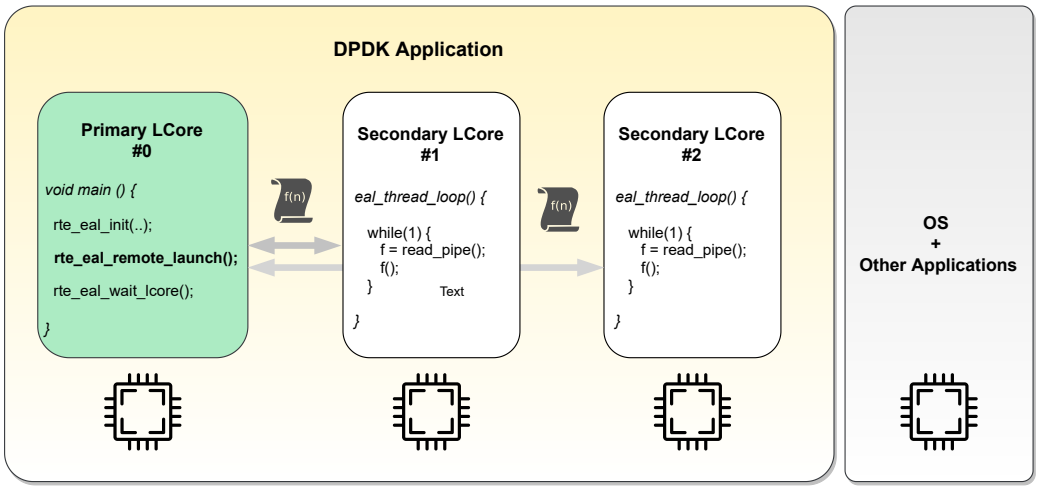

Figure 5.1 In a DPDK application, the primary lcore instantiates the secondary lcores and issues commands to them

Pipeline Library

To process packets, DPDK implements two models: the run-to-completion model and the pipeline model. In the run-to-completion model, the same thread is responsible for polling inbound traffic, processing it, and then forwarding it to output ports. In the pipeline model, traffic processing is split into a chain of functional stages handled by several threads.

The pipeline library, named *librte_pipeline*, allows the implementation of a data plane application as a set of interconnected pipelines. Each pipeline is composed of one or many stages. Typically, the logic of each pipeline stage is built around a lookup table. When a packet is dequeued from a port buffer, a classification process is started to match it with one of the flow rules in the lookup table. A matching flow rule determines what to do with the packet (i.e., forwarding to a specific port, dropping, encrypting) and its next processing stage.

A pipeline is made of three components: input ports, lookup tables, and output ports (Fig. 5.2). Input ports are connected to output ports via a tree-like topology of lookup tables. One input port cannot be connected to more than one lookup table. Nevertheless, a lookup table can be connected to one or many other tables. A pipeline uses the lookup tables to define the processing to apply on packet flows that it receives. When the outcome of a lookup process within a table is a hit, the action set of the matching rule is executed. The action set can cause a packet to be dropped, sent to an output port, or forwarded to another table, if further processing is required. A lookup table can optionally be configured with a default action set, to be applied in case of a lookup miss. One lcore can execute one or many pipelines, but a

given pipeline can not be executed by more than one lcore.

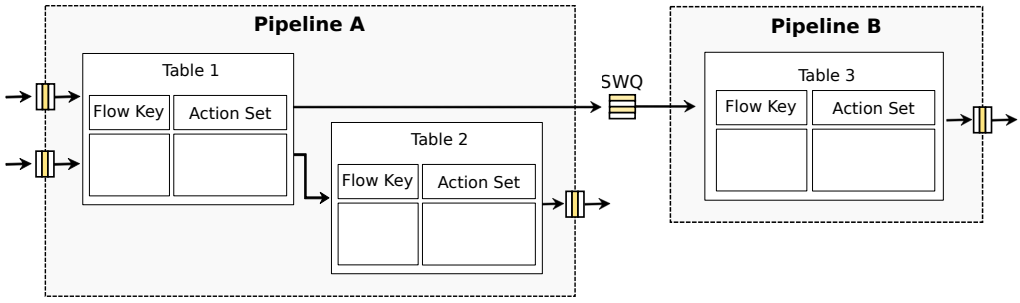

Figure 5.2 A super-pipeline is composed of two or more pipelines interconnected by software queues

Eventdev Library

The Eventdev library is a framework designed to introduce the event-driven programming model to DPDK. Compared to other models, the event-driven model enables applications to achieve better resilience and higher scalability by dynamically balancing workloads among several threads. Applications can use the Eventdev API to configure and manage event devices in the same way they do with Ethdev or Cryptodev devices. Moreover, they can leverage various sources and adapters (i.e., Ethernet NICs, Crypto devices, timers) to inject events into an event device.

Many specialized networking devices (e.g., Marvell OCTEON TX2) include a hardware event device, which is a built-in co-processor capable of scheduling and balancing workloads between CPU cores at the hardware level. DPDK provides a few PMDs to support widely used hardware event devices. It also provides a software implementation of an event device based on centralized and distributed scheduling mechanisms.

The scheduler can significantly impact the operation of an event device. Its principal role is to achieve load balancing by distributing injected events into event queues. The scheduler moves events to and from queues based on their scheduling strategies. The software event device supports three scheduling strategies: atomic, ordered, and parallel. Atomic scheduling ensures that there is only one CPU core processing the events of a given flow at any point in time. This scheduling type guarantees, therefore, that events are processed and output in the same order they were enqueued. Ordered scheduling preserves the order in which events are enqueued, but any CPU core can process the events of the same flow. Thereby, a reordering work is performed after those events are processed. Parallel scheduling is the

least constraining, since events of any flow can be processed by any CPU core. There is no guarantee on preserving the ingress order of events, or determining the order in which they will be processed or output.

5.5 Design of the Proposed Framework

This paper proposes a novel and efficient performance analysis framework for DPDK-based applications. Generally speaking, a typical performance analysis tool can be structured into three main layers: data acquisition, data analysis, and data presentation. The data acquisition layer focuses on collecting low-level performance data from the monitored system. On the other hand, the data analysis layer defines how to process the collected raw data. Its goal is to infer higher-level information that enables the practitioner to gain insight into the system runtime behavior. Finally, the data presentation layer displays and converts the processed information into various visualization elements, such as charts and graphs. This layer aims at helping the practitioner quickly and unambiguously pinpoint performance issues and identify their causes.

Following this generic architecture, our framework comprises three main modules (Fig. 5.3). The first module, called data collector, is responsible for collecting performance data from target applications. The second module, called data analyzer, is responsible for performing offline diagnostic analytics on the collected data. The third module includes a set of interactive graphical views showing different performance aspects of monitored DPDK applications. We describe the role of our framework modules in the following subsections.

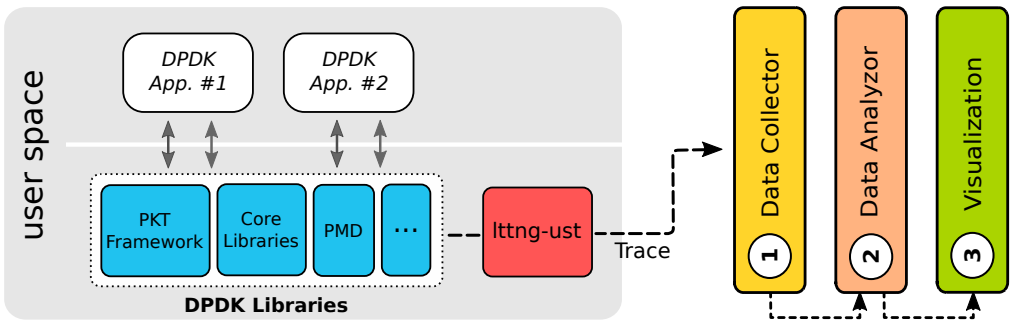

Figure 5.3 Architecture of the proposed framework

5.5.1 Data Collection

Collecting data is fundamental to understand what might have gone wrong in a program execution. Hence, to enable diagnosing the causes of performance bugs (e.g., performance bottlenecks), the first step a diagnostic tool must do is to collect data that describe all the changes in the states of the monitored application, and those of the systems it interacted with.

There exist two main techniques to capture performance data from a running program, namely logging and tracing. Logging involves emitting messages that denote either the progress of a given program execution or the occurrence of unusual behavior (i.e., warning and error messages). The logging information is habitually high-level and not overwhelming as it is intended to help administrators initiate a troubleshooting process in case a problem arises. Tracing, on the other hand, is an efficient technique for collecting low-level logging data. A diagnostic tool often needs such fine-grained data to solve intricate problems like pinpointing performance bugs or shedding light on the interactions between the elements of a complex system. Tracing, unlike logging, requires the use of an external software called a tracer. There is a multitude of compatible tracers for almost every OS. For Linux systems, for instance, we can find many popular tracers such as LTTng, Ftrace [82], and SystemTap [83]. However, before opting for a particular tracer, it is essential to take into consideration factors like the tracer licensing options (i.e., open-source, commercial), the required level of tracing (i.e., kernel space, userspace), and the supported instrumentation method (i.e., static, dynamic).

Our framework leverages the tracing technique because it enables the collection of fine-grained and low-level performance data required for our diagnostic analytics. In addition, to trace the execution of DPDK applications, we opted for LTTng, a high throughput tracer. LTTng is well-known for its ability to trace with very low overhead, thus making it practical for use in production environments. Currently, it is considered as one of the most advanced tracers, since it enables tracing Linux kernel (*lttng-modules*) and userspace applications (*lttng-ust*).

The simplest way to trace an application or a system is to instrument its source code. This technique, known as static instrumentation, involves inserting specific function calls, called probes or tracepoints, at appropriate locations in the application source code. Consequently, an event is fired, and its payload is logged, each time a probe is encountered during the application runtime. It is important to note that if the instrumentation is not carefully designed, the performance of the traced application could decrease significantly. Besides, if the traced application is a real-time application, the impact on the performance could lead to the violation of its timing constraints. Efficient instrumentation is therefore essential to minimize the impact on the application performance. Eventually, this comes down to

determining the instrumentation granularity by making a trade-off between the required amount of information to record and the impact on the performance.

Instrumenting DPDK libraries and drivers is a practical approach to gather performance data from DPDK applications. This approach eliminates the need to access the source code of the applications that use DPDK to process network traffic. Hence, we leveraged LTTng to instrument most DPDK libraries, especially those implementing DPDK scheduling and control plane mechanisms. For instance, we instrumented libraries like the Pipeline library, the Eventdev library, and the Elastic distributor library. To ensure that our instrumentation is effective, we kept the number of added tracepoints to the minimum. Besides, we adjusted the data exported through the tracepoints fields to the requirements of our analyses. As we will show in section 5.7, the tracing overhead increases with the amount of data exported via the tracepoints. To demonstrate our instrumentation method, the following subsections describe a subset of the tracepoints that we inserted in the Pipeline and Eventdev libraries.

Pipeline library

As we mentioned in section 5.4.2, a pipeline comprises three main components: input ports, lookup tables, and output ports. To build a pipeline, a developer must use the API provided by the *librte_pipeline* library to create those components and connect them to each other. To enable our framework to discover the application architecture, and more specifically, how pipelines are structured within it, we instrumented most of the API functions provided by the library. Moreover, to determine the paths that packets take through the pipelines input and output ports, we instrumented other libraries such as the *librte_port* library, which implements various port types (i.e., Ethernet, Source, and Sink ports). Super-pipelines are constructed by interconnecting pipelines using ring ports, which are efficient software packet queues. Our analyses discover interconnections between pipelines by comparing the identifiers of input and output ports. One pipeline is connected to another if at least one of its output ports is an input port for the other pipeline.

We also instrumented the *librte_port* library to compute performance metrics such as traffic RX and transmission (TX) throughput. Similarly, we instrumented flow classification libraries, like *librte_lpm*, *librte_hash*, and *librte_acl*, to enable the computation of per-flow performance metrics. Table 5.1 reports a subset of the tracepoints that our framework relies on to analyze the performance of DPDK pipeline-based applications. For instance, an application that uses pipelines to process packets must first use the API function *rte_pipeline_create()* to create a pipeline. It must also create input ports, lookup tables, output ports and then attach them to the pipeline using these functions *rte_pipeline_port_in_create*,

rte_pipeline_port_out_create, and *rte_pipeline_table_create* respectively.

Table 5.1 Description of a subset of tracepoints used to instrument the Pipeline library

Tracepoint	**Description**
rte_pipeline_create	This event is fired when a pipeline is created. The main fields of the event are *pipeline* (handler pointing to the pipeline instance) and *name* (pipeline name).
rte_pipeline_free	Free a pipeline identified by its handler.
rte_pipeline_port_in_create	The event is fired when a pipeline input port is created.The main event fields are *pipeline*, *port_id*, *burst_size*, *f_action*, and *h_port* (pointer to the low-level port instance).
rte_pipeline_port_out_create	The event is fired when a pipeline output port is created. The main event fields are *pipeline*, *port_id*, and *h_port*.
rte_pipeline_table_create	This event fires when a pipeline table is created. The main fields are *pipeline*, *table_id*, *entry_size*, *f_hit_action*, *f_miss_action*, and *h_table*.
rte_pipeline_port_in_ connect_to_table	This event is fired when an input port is connected to a pipeline table. Main events are *pipeline*, *port_id*, and *table_id*
set_next_table_id	This event indicates that a table is attached to a child table. The main fields are *pipeline*, *table_id*, *next_table_id*.
forward_to_next_table	This event fires when a table forwards packets to a child table. Main event's fields are *pipeline*, *port_id*, *from_tbl_id*, *to_tbl_id*, and *nb_pkts*.
rte_pipeline_table_default_ entry_add	This event indicates that a default entry is added to a pipeline table. The main fields are *table_id*, *action* (e.g., ACTION_DROP, ACTION_PORT, and ACTION_TABLE), and *next_port_table_id*.
rte_pipeline_table_entry_add	This event is fired when a flow rule is inserted into a pipeline table.

Eventdev library

DPDK provides the Eventdev library to enable applications to schedule events through the use of event devices. To get insights into the operation of these devices, we instrumented the available PMDs *event_sw* and *event_dsw*. On the one hand, the former PMD runs as a

service and implements centralized scheduling of the events. On the other hand, the latter distributes the scheduling task among all the threads attached to event ports. Therefore, the PMD *event_dsw* requires those threads to perform enqueue and dequeue operations periodically to achieve load balancing. In particular, producer-only threads should regularly execute the API function *rte_event_enqueue_burst()*, even if they do not have any event to enqueue. The focus of this paper is on the PMD *event_sw* since it does not present this limitation. In addition, we instrumented the Eventdev library, causing thus calls to its API functions to trigger tracing events with the same name (Table 5.2). We use the data exported by all the tracing events to compute performance metrics and discover the structures of the used event devices.

As we explained earlier, three logical entities are involved in the event device functioning: the queues, the ports, and the scheduler. At first, the application must configure the event device using the function *rte_event_dev_configure()*. It must then configure the event queues to be added to the device by specifying various options, such as the scheduling type. It also must create and configure the event ports and define the way to connect them to the created queues using the function *rte_event_port_link()*. Finally, the last step is starting the device by calling the function *rte_event_dev_start()*. Once the event device is started, threads can use the functions *rte_event_enqueue_burst()* and *rte_event_dequeue_burst()* to enqueue and dequeue events via their assigned ports.

A port is the point of contact between application threads and the event device. Therefore, each thread must be assigned a port to be utilized exclusively. An event port is mainly composed of two ring buffers: one RX buffer and one consumer buffer. When a thread processes an event and injects it into the event device for further processing, the event is stored in its port RX buffer. Then, the scheduler periodically moves the events from the RX buffers of the ports to their designated queues. It also moves events from queues to appropriate ports by storing them in their consumer buffers, depending on their occupancy and the type of the source event queues.

5.5.2 Analysis and Visualization

Tracing a software execution often results in a high event generation frequency, depending on various factors such as the instrumentation granularity and the required level of detail for tracing [118]. Luckily, advanced tracers, such as LTTng, are designed to handle a large number of high throughput event streams. The amount of tracing data that it may generate can, however, be overwhelming. Hence, the size of the generated trace might quickly become huge and could easily reach many gigabytes of disk space for a tracing duration of just a few

Table 5.2 Description of a subset of tracepoints used to instrument the Eventdev library

Tracepoint	**Description**
rte_event_dev_configure	This event is fired when the application configures an event device. The main event fields are *dev_id*, the event device identifier, and *nb_event_queues* and *nb_event_ports*, which denote the number of queues and ports to configure on the device, respectively.
rte_event_queue_setup	This event indicates the options used to set up an event queue. The main event fields are *queue_id*, the identifier of the queue, *scheduling_type*, the scheduling type of the queue (i.e., atomic, parallel, ordered), and *priority*, the priority assigned to the queue.
rte_event_port_setup	This event indicates the options used to set up an event port. The main event fields are *port_id*, the identifier of the port, and *dequeue_depth*, the maximum number of events that the port can dequeue at a time.
rte_event_port_link	This event indicates how a given port is linked to queues. The main event fields are *port_id*, the identifier of the port, and queues, a list of identifiers of queues to be linked with the port.
rte_event_dev_start	This event is fired when the device becomes ready to accept and process events.
rte_event_enqueue_burst	This event is fired when a worker thread injects events into an event device. The main event fields are *n_enq*, the number of enqueued events, *n_new*, the number of new events among those enqueued.
rte_event_dequeue_burst	This event is fired when a worker thread dequeues events from an event device. The main event field is *n_deq*, the number of dequeued events.

seconds. Consequently, the growing trace size makes any manual data analysis unfeasible, especially when this data is gathered from large-scale systems. Therefore, efficient automatic analyses are crucial to gain valuable insights into the program runtime from a massive volume of tracing data.

Building performance models from fine-grained tracing data is essential for pinpointing and diagnosing potential performance problems in complex systems (e.g., DPDK multi-threaded applications). Hence, we identify two major approaches to build such models from low-

level performance data: machine learning and data abstraction. The first approach involves applying machine learning techniques to the gathered data in order to discover (and sometimes predict) application behavior anomalies or performance pathologies. A state-of-the-art survey reports several proposals exploiting well-known machine learning classification algorithms for tracing data analysis, such as Naive Bayes (NV) [119] and Support Vector Machine (SVM) [120]. We believe that this approach is inadequate for our context because it requires a costly training process. Furthermore, the diagnoses achieved are often inaccurate due to the high number of false positives and false negatives.

On the other hand, the second approach is based on using data abstraction techniques to derive performance models from the collected traces. It considers, therefore, the model as an abstract representation of the trace. The leveraged abstraction techniques aim to decrease the level of details in the trace to the level at which the practitioner wants to analyze the performance problem. Hence, current performance analysis tools implement most of those techniques at the analysis and visualization levels.

At the analysis level, one interesting abstraction technique aims to analyze the dynamic behavior of the software by recovering its runtime states from the trace data. In essence, this technique models the software as a finite set of states. The software state is composite in that it can be broken down into the states of its internal modules or components. Thereby, a software component can be seen as a finite-state automaton, as it can be represented by a sequence of states, and a state change only occurs when events are triggered. For instance, when a thread is spawned in Linux, its state switches between the following values: ready, waiting, running, blocked, and terminated. Accordingly, the kernel event *sched_switch* triggered by the Linux scheduler indicates a transition in the states of two threads: one switching to the *Running* state while the other switching to the *Waiting* or *Blocked* state. Therefore, performance debugging consists in checking the consistency of the states of the software components and their underlying resources. Thus, an invalid state combination is a good indicator of a performance problem or software misbehavior. For example, a deadlock situation can be discovered by checking the states of the concurrent threads against the states of the solicited resources.

Another data abstraction-based technique is leveraging the payload of the tracing events to derive application specific performance metrics. Performance degradation is thus discovered when a set of metrics goes below or above explicit thresholds. For example, a DPDK application is considered overloaded if its packet loss rate and per-packet processing latency are found to be above certain thresholds.

To sum up, the framework that we propose in this paper relies on the data abstraction tech-

niques mentioned above to build various data-driven performance analyses. We implemented this framework as a set of plugins in Trace Compass [87]. This latter is an open-source trace analyzer that is empowered with robust pre-built analyses. We chose to use Trace Compass because it provides several functionalities facilitating the development of efficient trace analysis, such as events filtering and aggregating and multi-level abstraction facilities. Furthermore, our methodology to debug the performance bottlenecks of a DPDK application is based on a top-down approach. Our framework provides several interactive graphical views which enable the practitioner to monitor the application's global performance metrics. Nevertheless, when a performance problem is suspected, the practitioner can use more detailed views to understand the issue better and investigate its root cause. Those views reveal the changes that occur in the states of the software internal components and highlight their inconsistencies.

Data Modeling

In this context, we define data modeling as the process of designing adequate data structures for storing and managing the states of the monitored application and its resources (e.g., NIC Queues). Thus, to ensure the highest possible performance for the developed analyses, chosen data structures must respect certain performance requirements. The literature reports many data structures capable of storing intervals such as segment-tree, interval-tree, and R-tree. However, choosing those data structures was not an option for us because they are designed to store data in the main memory only. As we mentioned earlier, very large traces can easily exceed the capacity of the available RAM of the machine dedicated for analyses running. Besides, data structures that require trace rereading are also unacceptable since using them would affect the scalability of the analyses. For a satisfactory user interaction and experience, our framework must enable the practitioner to browse the traces interactively and vary the level of details without inducing a noticeable overhead.

To overcome all those limitations, we decided to use the State History Tree (SHT) proposed by *Montplaisir et al.* in [88]. The SHT is an efficient and scalable tree-based data structure that can store, manage, and retrieve any number of system attributes and state values. This data structure supports many trace formats and enables the definition of new attributes and states. From a performance perspective, it ensures fast read access to the stored data, estimated by the authors to O(log n) time. In addition, it is disk-based, which means that it does not impose a strict limitation on the size of the input trace.

Based on the SHT mentioned earlier, we implemented an interval-time tree-based data structure to store the states of the components of monitored DPDK applications. Hence, we

exploited this data structure to store the states and attributes of the different software components that a DPDK application might leverage, such as LCores, flow classification tables (e.g., LPM, Hash, ACL), and scheduling mechanisms (e.g., Pipelines and Eventdev objects). The analyses implemented within our framework involve two stages. The first stage reads the input trace sequentially, extracts the useful information from the payloads of its events, derives the state values relevant to the analysis, and incrementally populates the analysis data structures with the derived state values. The second stage reads and transforms those states and attributes into measures showing various performance aspects of the traced application. The graphical views that we implemented display those measures in the form of charts and graphs.

Fig. 5.4 shows an excerpt of the tree-shaped data structure that we mentioned in the paragraph above. We use the branch entitled "Pipelines" to store the states of the different components of pipeline objects, managed by the application via the DPDK Pipeline library. As explained earlier, this branch shows that a pipeline is composed of input ports, output ports, and classification tables.

To illustrate how our analyses populate this branch, let us look at the example of attaching a new input port to a given pipeline. Our analysis requires the presence of three events in the trace file to achieve this goal. The first event indicates that the application has configured a new device (i.e., the event *rte_ethdev_configure* for an Ethernet NIC). The payload of this event includes information, such as the id, name, and type of the configured device. The second event indicates creating a port for reading packets from a specific device RX queue (i.e., the event *rte_port_ethdev_reader_create* for an Ethernet NIC). Finally, the last event named *rte_pipeline_port_in_create* indicates that the newly created port was attached to a specific pipeline. On the other hand, the node attribute *pinned_to_tbl* indicates that each input port must be pinned to only one classification table. Our analysis extracts the identifier of the latter from the payload of an event named *rte_pipeline_port_in_connect_to_table*. This event is triggered when the application executes the API call having the same name (see Table 5.1). In the same way, the attribute *nb_rx* reports the number of packets read by the input port over time. This information is filled in using the payload of various events, depending on the type of the port (i.e., Ethernet port, Sink port, Source port, Crypto port, Ring port). For example, a pipeline thread uses the API call *rte_port_ethdev_reader_rx* to retrieve packets from a given Ethernet port, thus triggering an event with the same name. The most important fields of this event are the port identifier (*port_id*), the number of retrieved packets (*nb_rx*), and the timestamp (*ts*). Our analysis uses the value of the field *port_id* to locate the involved port within the tree and the values of the two other fields to update its state.

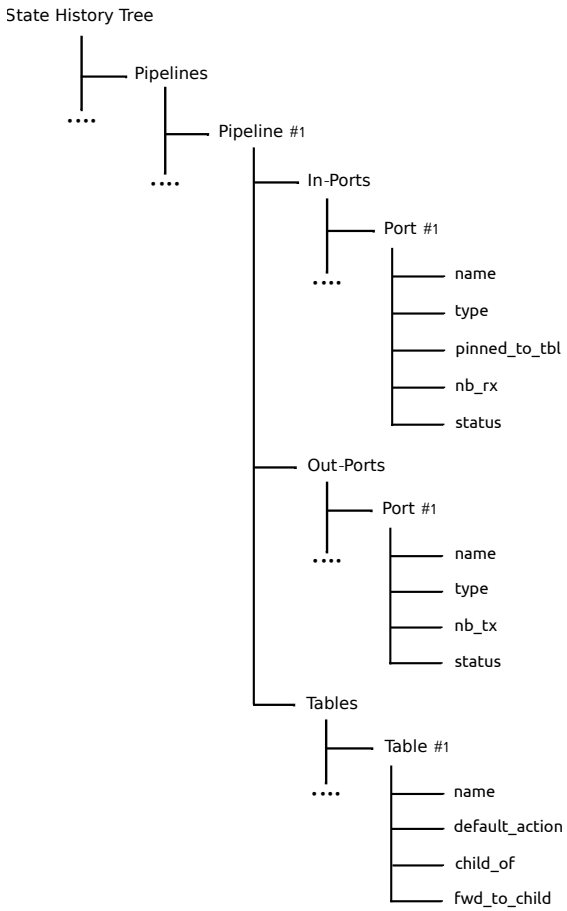

Figure 5.4 Excerpt of the tree-based data structure on which our framework is based.

Data Analysis

Our performance analysis framework relies on the computation of adapted performance metrics and the derivation of the states of DPDK object resources to pinpoint DPDK application bottlenecks. Some of those metrics are generic and can be leveraged to detect performance degradation like per-port RX/TX throughput and packet drop rate. Some others are specific to DPDK applications, such as the percentage of effective RX spins. We present a subset of those performance metrics in the following paragraphs.

Packet rate and network throughput The packet rate metric is the number of packets sent or received by a port per second. On the other hand, the network throughput is the number of bits transmitted or received by a port during the same time unit. The first metric is usually measured in packets per second (p/s or pps) while the second is in bits per second (b/s or bps). Often, a bottleneck is observed when the values of these two metrics fall below

expectation. Since DPDK applications retrieve and process packets in batches, collecting the data required to compute these metrics can be performed with low overhead. Formula 5.1, for instance, explains how to compute this last metric. The values B_{t1} and B_{t2} indicate the byte counts at event timestamps t1 and t2 respectively, while the value T denotes the observation time interval.

$$Throughput = \frac{(B_{t2} - B_{t1}) \times 8bits}{T} \tag{5.1}$$

Percentage of effective RX spins We leverage this metric to estimate how effectively the DPDK threads utilize their dedicated CPU cores. As we explained earlier in section 5.4.1, DPDK adopts a pure polling-based model to retrieve packets from the NICs, in order to minimize their processing latency. Hence, the polling threads of the DPDK application repeatedly call a *dequeue_burst* like API function to retrieve packets. For example, a DPDK thread can use the function *rte_eth_dequeue_burst* to poll packets from an Ethernet NIC, or the function *rte_eventdev_dequeue_burst*, to dequeue packets from an Eventdev device. The DPDK polling thread typically calls this kind of function from within an infinite loop, which causes their dedicated CPU cores to show 100% utilization, even when there is no incoming traffic.

Consequently, it is absurd to use the classic metric of CPU utilization to estimate the load of the polling threads, since this metric will always point towards values close to 100%. For this reason, we propose another metric called the percentage of effective RX spins. This metric gives a reasonable estimation of the effective CPU utilization of the polling threads, and allows evaluating to what extent those threads were being busy fetching packets. We calculate this metric by dividing the number of times a call to a dequeue_burst-like function results in one or many packets divided by the total number of calls to that function (Equation 5.2). Nevertheless, this metric presents a limitation: it does not show the difference in performance between a thread retrieving a large number of packets in each poll and another thread retrieving only a few.

$$Effective\ RX\ Spins = \frac{No.\ successful\ calls\ to\ X\_dequeue\_burst() \times 100}{Total\ number\ of\ calls} \tag{5.2}$$

Average number of dequeued packets We propose this metric to address the limitation we mentioned regarding the previous one. It denotes the average number of packets dequeued from a NIC by a polling thread. This metric is particularly useful to tune the frequency at which DPDK threads poll NICs. For instance, when the average number of packets dequeued

from a given NIC is very low, one can reduce the polling rate of its attached thread. It is also possible to assign an additional workload to that thread or use it to poll more than one NIC. Hence, the practitioner can use this metric alongside the metric *percentage of effective RX spins* to monitor the CPU resources and ensure that they are not wasted. We calculate this metric using the formula below.

$$Avg.\ No.\ Dequeued\ Packets = \frac{No.\ packets\ dequeued\ by\ X\_dequeue\_burst()}{Total\ number\ of\ calls} \tag{5.3}$$

Queue latency DPDK implements the pipeline model that allows packet processing to be performed in stages. According to this model, a DPDK thread performs one processing round on the received packets before passing them to another thread through a software queue to undergo another processing round. Software queues are often implemented based on the DPDK library named *librte_ring*. The latter provides a set of data structures (i.e., *rte_ring*) and algorithms enabling the creation and management of efficient FIFO, lock-less, multi-producer, and multi-consumer ring buffers. The instrumentation of this library allowed us to track the enqueue and dequeue operations performed by DPDK threads and, then to compute the waiting time that packets spend in these queues. Hence, we define the metric *Queue Latency* as the average time that packets spend in a given software queue, waiting to be dequeued and processed. Usually, a bottleneck situation is spotted when the value of this metric increases over time and becomes significant. Hence, the growing queue latency implies that the underlying consumer threads were slower than the producer threads. It is worth noting that the queue latency metric is mainly influenced by the size of the queue and the packet processing speed.

Visualization

Visualization is another abstraction level that can be applied to traces to highlight meaningful information. As we explained earlier, execution traces are often characterized by their huge size and an enormous amount of low-level events. Browsing and interpreting such a massive amount of raw data without appropriate tools is almost impossible because most events are only understood when correlated to each other within a specific context. Fortunately, there exist many visualization tools which can help improve the readability of the trace and reduce its complexity. Trace Compass, for instance, is a popular open-source trace visualization framework that is widely used for overviewing traces and browsing their contents. We chose this framework as our main development platform because it supports various visual abstraction features, such as filtering insignificant events and aggregating related ones.

Hence, we developed several interactive graphical views within this platform. These views allow the practitioner to gain clear insights into the DPDK application behavior by identifying relevant patterns and pinpointing anomalies. For instance, they can easily be used to overview a trace, zoom into areas of interest to get more details, and examine any trace fragment from different perspectives. The developed views are auto-synchronized, which means that selecting an event, a timestamp, or a time range in a view will cause the other views to update accordingly. Moreover, those views use diverse visual representations depending on the analysis data to display. Therefore, our views include XY charts (e.g., Fig. 5.5), time graphs (e.g., Fig. 5.6), and control-flow graphs (e.g., Fig. 5.7).

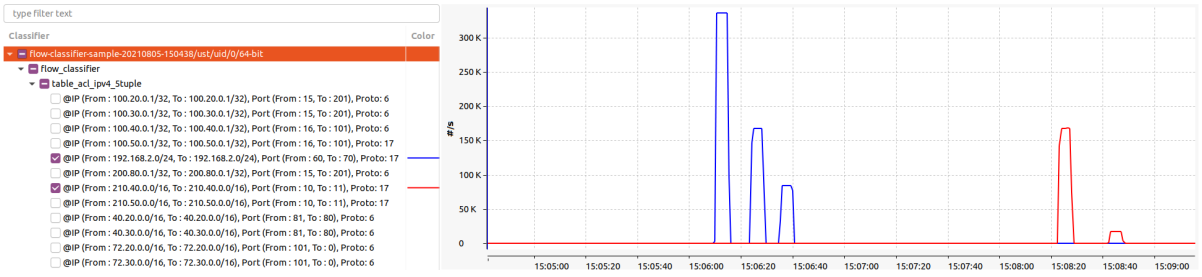

Figure 5.5 Example of an XY view showing the rate at which packets matched with the flow rules of an ACL classification table.

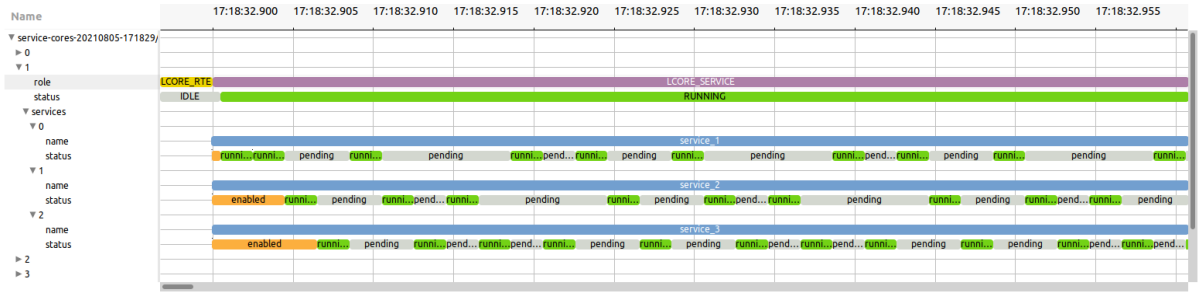

Figure 5.6 Example of a time graph view showing the status of application lcores, alongside the status of the services attached to service lcores.

5.6 Use Cases

A DPDK application, like any other application, is likely to experience a performance decrease if it is exposed to growing loads. What makes our tool important in this matter is its capacity to monitor the performance of this application and guide the practitioner to pinpoint which part of it is causing a performance bottleneck. In most cases, a fine-tuning of the application configuration suffice to solve the problem. Nevertheless, in some cases,

more elaborate measures are required, such as improving the application design and fixing implementation bugs.

In this section, we validate our approach and show how our tool can help the practitioner diagnose the performance bottlenecks of DPDK applications and pinpoint performance optimizations opportunities. Our strategy for diagnosing performance bugs is to identify the potential causes of the problem and to eliminate them one by one at each step of the debugging process. Hence, we present in the following paragraphs two representative use examples to showcase our tool capabilities. In these use cases, we analyzed the performance of a few sample applications available in the DPDK project. Our experiments did not cover more "complete" applications (i.e., OVS, VPP, Contrail vRouter) because they do not use the main features of DPDK. Hence, most studied software switches merely leverage DPDK PMD. On the contrary, the sample applications enabled us to illustrate our analyses potential and focus on targeted DPDK features. Table 5.3 shows the hardware and software configuration used to run our experiments.

Table 5.3 Hardware and software experimental configurations

Hardware Environment		**Software Environment**	
CPU	Intel Xeon CPU E5640 @ 2.67GHz	OS	Ubuntu 18.04 (Kernel 4.15)
# Cores	16	DPDK	version 19.08
RAM	189 GB	LTTng	version 2.12.0
		Trex	version 2.52

5.6.1 The Pipeline sample application

In a first experiment, we created a custom pipeline-based program using the Internet Protocol (IP) pipeline application (*./app/dpdk-ip_pipeline*). DPDK provides this sample application to ease the prototyping of complex pipeline-based applications. The IP pipeline application can indeed be leveraged to create real-world applications by instantiating and connecting various resources from the DPDK packet framework. These resources include pipeline instances, mempools, NIC RX/TX queues, software queues and traffic managers.

To create our custom application, we implemented a configuration file in which we described all the required resources and the processing to perform on packets. The IP pipeline application takes this configuration file as an input parameter. Hence, based on the aforementioned file, we defined a super-pipeline composed of three pipelines, each executed by a thread mapped to a different CPU core. We interconnected the pipelines using the software queues

SWQ1 and SWQ2. In addition to pipelines threads, our application also spawns two other threads. One RX thread to dequeue the packets received by the port net_vhost0, and another TX thread to transmit them through the port net_vhost1. The architecture of the super-pipeline implemented by our application is presented in Fig. 5.7.

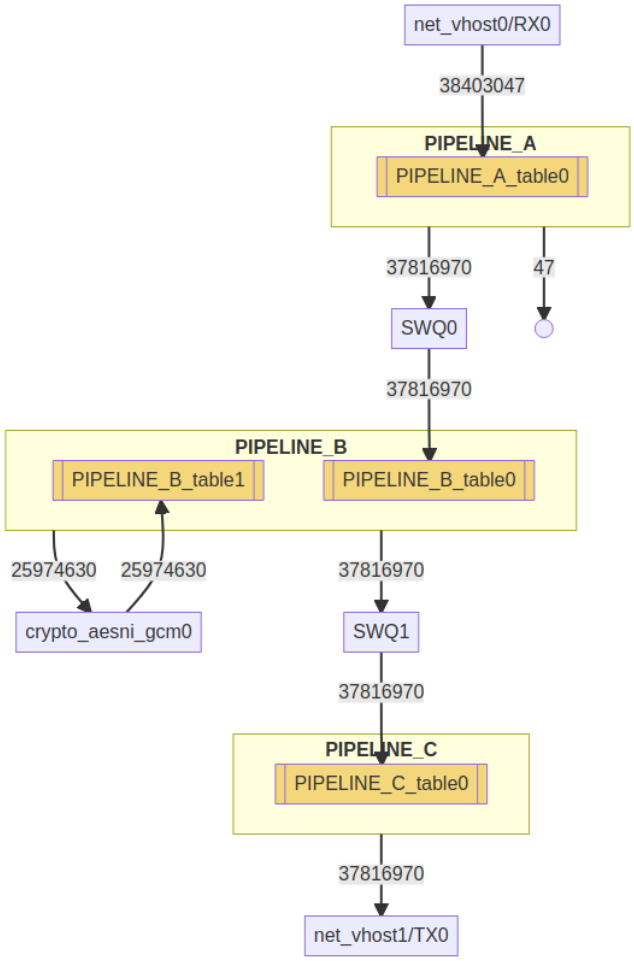

Figure 5.7 View generated by our tool showing the architecture of our super-pipeline and the flow of received traffic.

In our setup, packets are routed from the input ports of the super-pipeline to its output ports through different processing stages. We connected the input ports of the first pipeline to an ACL classification table. Traffic received on the input ports is filtered or routed towards the output ports, depending on the flow rules inserted in the pipeline classification tables. Filtered packets are sent to a sink port and are stored in a PCAP file. The second pipeline uses two classification tables to encrypt packets of specific flows before sending them to the next stage. This pipeline leverages the librte_cryptodev library to create a software crypto device that can perform symmetric encryption operations. Finally, the third pipeline

forwards the packets received from the second pipeline to its output ports. We connected our application to a VM running the Trex Traffic Generator using a pair of vhost-user NICs, namely *net_vhost0* and *net_vhost1*. Hence, packets that Trex transmits on the first NIC are received, processed, and forwarded to the second NIC by our application, running on the host side.

To avoid overflowing the RX/TX buffers of the two virtual NICs, we fixed the traffic generation rate at 320 kpps. Moreover, we fixed the rate of packets that require encryption at 220 kpps. We issued the flows of those packets in bursts with an inter-burst gap of 4ms. We issued the other flows at a rate of 100 kpps using continuous streams. In this experiment, which lasted 120 seconds, we noticed that the rate at which the NIC net_vhost1 was transmitting packets was lower than expected (316 kpps) (Fig. 5.8).

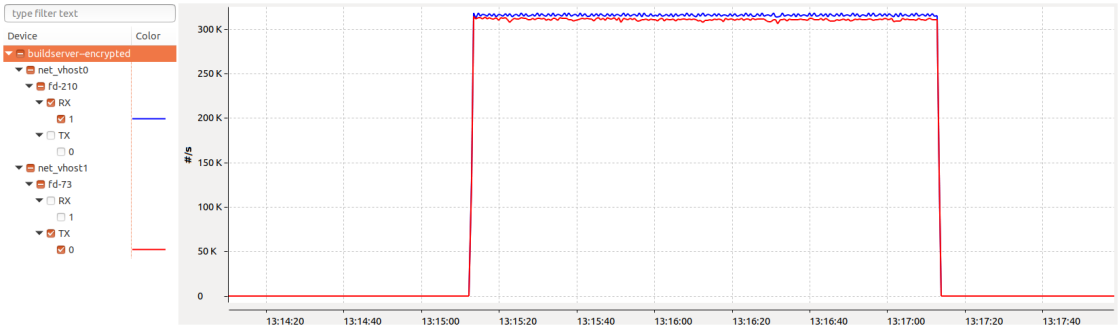

Figure 5.8 The rate of packet forwarding is lower than the rate of the packet reception.

We traced and analyzed this experiment to diagnose the cause of this performance issue (the gap between RX and TX packet rates). At first, we needed to confirm that our super-pipeline architecture corresponds to the specification as we indicated in the configuration file. For this reason, we used our tool to generate the view in Fig. 5.7. This figure shows that the first pipeline filtered 47 packets by sending them to a sink port, while the remaining ones traversed the super-pipeline until the TX queue of the NIC net_vhost1. It also reveals that the second pipeline forwarded almost two-thirds of the received traffic to the crypto device for encryption before routing it to its next stage. We can also notice that the number of packets that the Pipeline_A enqueued to SWQ1 is below the number of packets it received.

To identify the actual cause of packet loss, we had to check the occupancy of our two NICs' RX and TX buffers. Luckily, our framework can calculate and visualize the occupancy of NICs buffers during the application execution. As we can see in Fig. 5.9, the occupancy of the RX buffer of net_vhost0 and the TX buffer of net_vhost1 did not reach its limits. Therefore, we can exclude the hypothesis that packets were dropped due to a NIC buffer overflow.

We also checked the latency and the occupancy of the software queues connecting the pipelines of our application. Fig. 5.10 shows that the latency of queue SWQ0 is much higher than that of queue SWQ1. This suggests that the thread executing Pipeline_B was slower in dequeuing packets from that queue than the thread executing Pipeline_A. Fig. 5.11 also confirms this observation as often the occupancy of SWQ0 reached 100%, causing thus packets to be dropped. Packet encryption represents a heavy workload when performed by a single thread, which explains why the Pipeline_B thread was slow. Therefore, it is mandatory to reduce the load imposed on the Pipeline_B thread to avoid packets piling in SWQ0. Hence, our solution was to add another pipeline responsible for dequeuing packets from that queue and encrypting them.

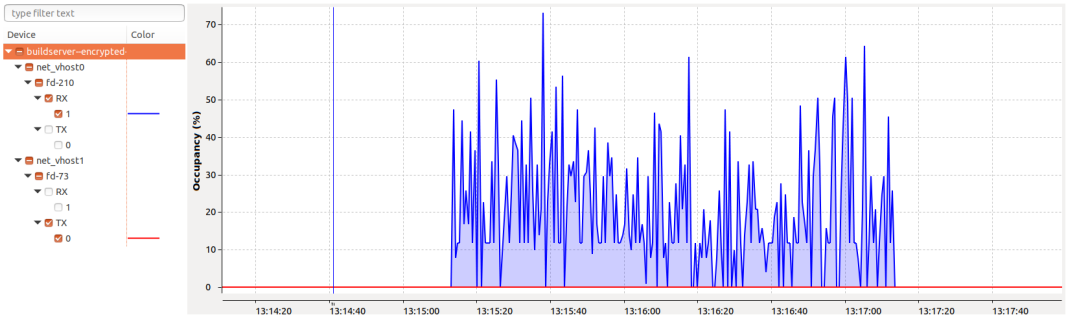

Figure 5.9 Traffic generated by TRex did not overflow neither the RX buffer of the first NIC nor the TX buffer of the second NIC.

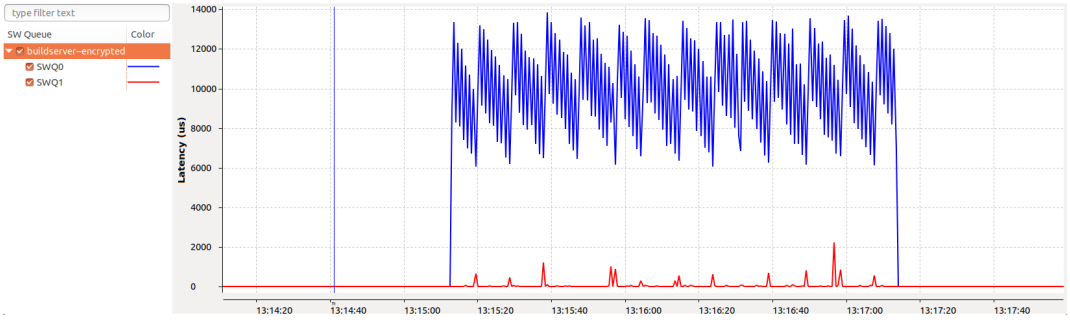

Figure 5.10 Latency of the software queues in the super-pipeline.

5.6.2 The Eventdev pipeline sample application

This second experiment is related to the application *./dpdk-eventdev_pipeline*, the Eventdev pipeline sample application. The latter leverages the Eventdev framework to configure a packet processing pipeline enabled with dynamic load balancing and multi-core scaling capabilities. Hence, we configured this application to run one RX thread, one TX thread, and

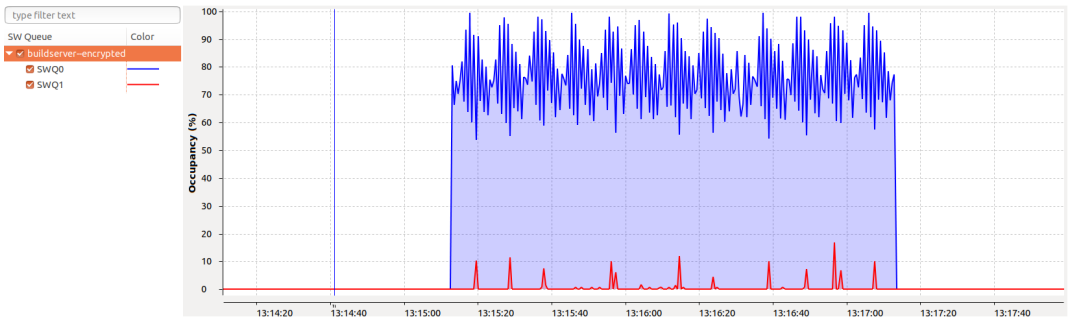

Figure 5.11 Occupancy of the software queues connecting the pipelines used by our application.

4 worker threads. Each of the six threads is run on a separate CPU core. As shown in Fig. 5.12, we configured our application to perform packet processing in two stages. Therefore, our pipeline is composed of two atomic queues and one single-link queue. Each atomic queue represents a distinct stage in the pipeline. The single-link queue is a queue that can only be connected to a single port. In our case, it is the one to which the TX thread is attached. We attached two worker threads to each atomic queue. Hence, we attached Worker0 and Worker2 to Queue0, and Worker1 and Worker3 to Queue1.

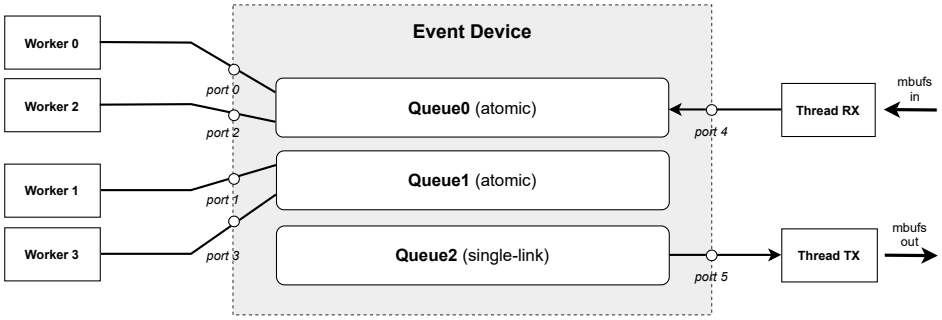

Figure 5.12 Architecture of the event device used by our monitored application.

In short, the operating of our setup is the following. The RX thread polls packets from a vhost-user NIC named net_vhost0, and then injects them into the first atomic queue using its port. The four worker threads rely on the Eventdev software scheduler to retrieve packets from their assigned queues via their corresponding ports. Packets that are dequeued by worker threads, are processed, and then returned to the event device. Worker0 and Worker2 collaborate to process packets retrieved from Queue0, the first stage in our pipeline. Similarly, Worker1 and Worker3 collaborate to process packets retrieved from Queue1 representing the second stage. The processing overhead in the first and second stage is the same because we are simply emulating 5k cycles of work per packet in each. When the processing of a packet

is finished, it is moved by the scheduler to the single-link queue. The TX thread polls packets from this latter queue and transmits them to the network via the vhost-user NIC, named net_vhost1.

We ran our DPDK application on the host machine, and the traffic it processed was generated by a traffic generator running within a VM. The application was connected to the VM via two vhost-user NICs, namely net_vhost0 and net_vhost1. For those NICs, we fixed the size of the RX buffer at 4096 mbuf descriptors and the size of the TX buffer at 128 mbuf descriptors. Hence, we used TRex Traffic Generator to generate six packet flows at the rate of 500 kpps. We traced the application execution for 60 seconds and used our framework to analyze the obtained trace. By examining the view depicting the RX/TX throughput of the application NICs, we noticed that the rate of packet forwarding was lower than expected (approximately 480 kpps).

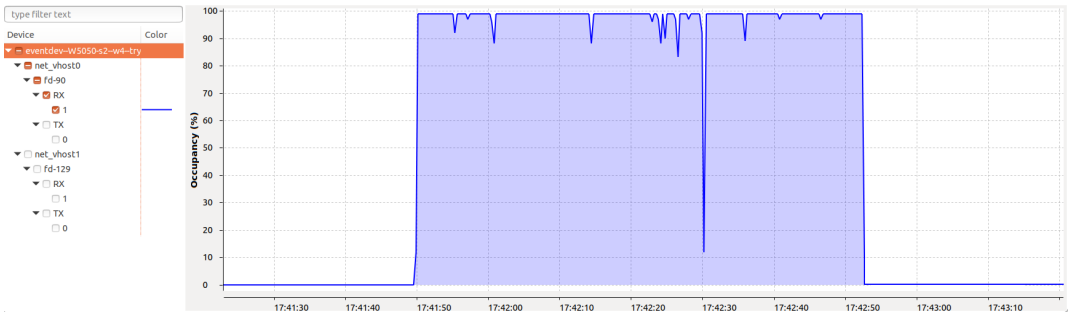

Figure 5.13 Occupancy of the RX buffer of the first vhost-user NIC.

Packet loss is often the cause of network throughput degradation. To check this assumption, we had to look at the occupancy of our virtual NICs RX and TX queues. As we can see in Fig. 5.13, which depicts the occupancy of net_vhost0 and net_vhost1 buffers, the RX buffer of the first NIC was almost full during the whole experiment. This means that the rate at which our application was dequeuing packets from the RX buffer of net_vhost0 was lower than the rate at which the traffic generator was enqueuing packets into it. Incoming packets are dropped by the driver whenever the NIC RX buffer is found full. To verify whether there exists a bottleneck within our application pipeline causing packets to get dropped, we had to verify the rate at which packets were enqueued and dequeued by all the threads. The view in Fig. 5.14 shows a significant fluctuation in the dequeue rate of Worker1 compared to that of Worker0. As a reminder, Worker0 and Worker2 are the threads responsible for the first stage of processing. According to this view, both threads dequeue packets from the event device event_sw0 at a stable rate of 240 kpps. By comparing the dequeue rate of worker threads, we can observe that the dequeue rate of the threads allocated to the second stage

was oscillating. Therefore, it is likely that the disruption in the operating of these latter threads might be the cause of the bottleneck.

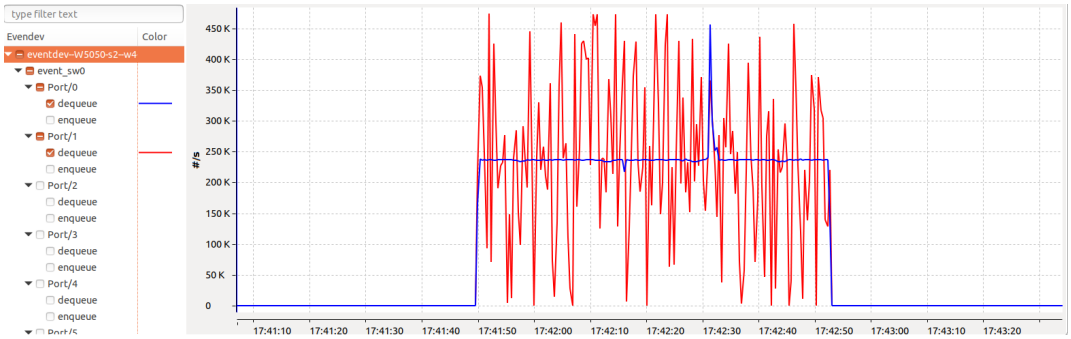

Figure 5.14 The packet dequeue rate of Worker1 (red) is characterized by a considerable fluctuation while it is stable for Worker0 (blue).

To confirm this finding, we compared the performance of the two groups of worker threads using another metric, the percentage of effective RX spins. This metric gives a good indication of whether the worker threads were overloaded or not, especially when contrasted with other metrics such as RX and TX buffer occupancy. Based on this performance metric, figures 5.15 and 5.16 show that the threads responsible for the second stage were much more overloaded than those responsible for the first stage. We can observe in Fig. 5.17, showing the same performance metric, that the former threads were not dequeuing and processing packets in parallel, but rather in turn.

To unveil the root cause of this behavior, we looked at another view showing how the flows were mapped to ports by the scheduler. As explained in section 5.4.2, the scheduler ensures that multiple threads do not process the events of a given flow simultaneously for queues configured with atomic scheduling. When the scheduler moves packets from an atomic queue to the buffers of attached ports, it first checks whether these packets belong to a flow that has already been pinned to a given port. If it is the case, the packets are moved to the buffer of that specific port. Otherwise, the scheduler selects among the ports attached to the atomic queue the least busy one, depending on the occupancy of their consumer buffers. Henceforth, the new flow becomes pinned to that port, and subsequent flow packets are moved to its buffer. Whenever the last flow packet is moved from an atomic queue, the scheduler unpins the flow from that port.

Fig. 5.18 clearly illustrates this mechanism. It shows for each atomic queue the time during which flows were pinned to the ports. For instance, it shows that at any time, there were three flows pinned to each port attached to Queue0. Those flows were sometimes pinned to port0 and other times to port2. However, regarding Queue1, the flows were aggregated into

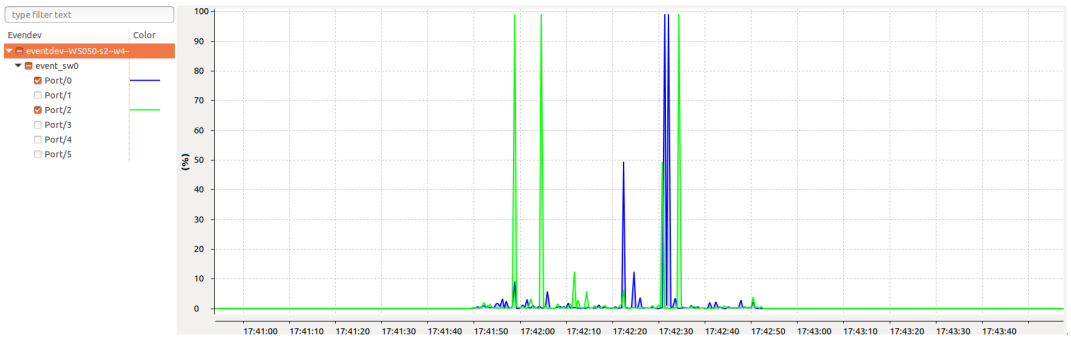

Figure 5.15 The performance metric percentage of effective RX spins shows that the worker threads responsible for processing packets in the first stage were underloaded

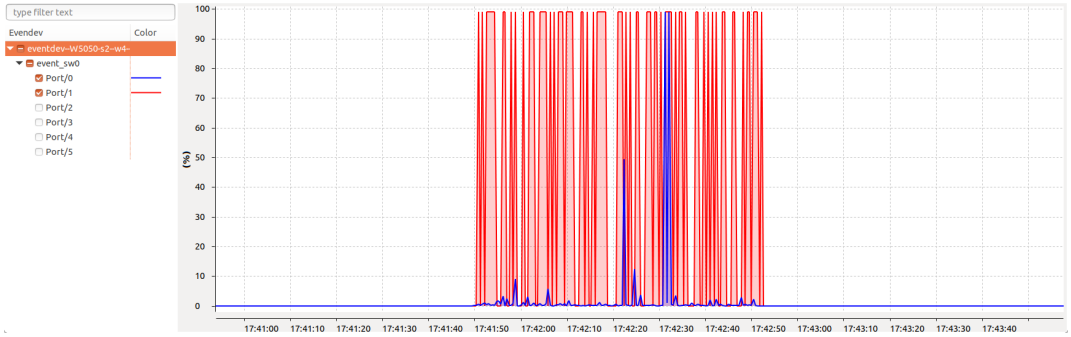

Figure 5.16 According to the metric percentage of effective RX spins, Worker1 (second stage) looks more overloaded than Worker0 (first stage).

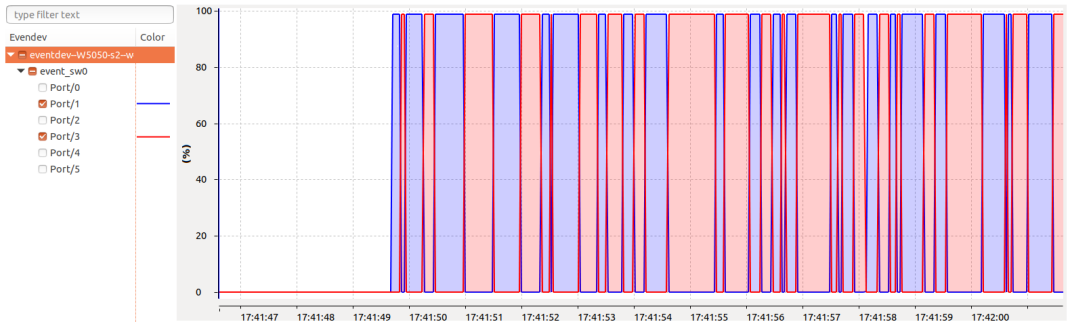

Figure 5.17 Zooming in the view calculating the percentage of effective RX spins, reveals that the worker threads responsible for the second stage were not dequeuing packets from the event device in parallel, but rather in turn.

a single elephant flow. The latter was sometimes pinned to port1 and other times to port3. This explains why Worker1 and Worker3 were dequeuing and processing packets in turn and not in parallel. The view also indicates that, in the first stage, the application changed the packets flow identifier to the value of zero. Indeed, when we looked at the application

source code, we discovered that worker threads update the value of the field *flow_id* of each dequeued event during the first processing stage. The new value is derived from the field *hash.rss* of the *mbuf* data structure. The *hash.rss* field contains a hash value that NICs generate at the hardware level. However, it turned out that the vhost-net driver managing our virtual NICs does not support the RSS feature. Therefore, we can conclude that the performance bug is caused by the fact that our application does not check the value of the field *hash.rss* before deriving a new flow identifier.

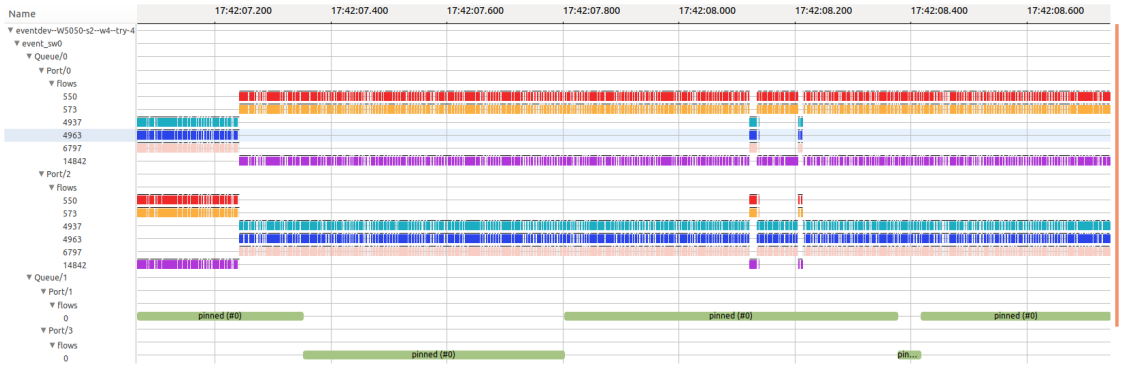

Figure 5.18 During the second stage of the pipeline, all the flows were aggregated in a single elephant flow that it is pinned alternately to Worker1 and Worker3

5.6.3 Use Case Discussion

These two use cases are sufficiently representative to clearly illustrate how the provided tool can be used to quickly identify and understand bottlenecks. With the Linux kernel level events, the state of each thread can easily be monitored (running, waiting for CPU, blocked on I/O). Thus, interference from other processes, or contention for resources, can easily be identified. With the additional information provided by the instrumentation of the DPDK libraries, dropped packets in queues, processing threads interactions, improper load balancing, or other similar problems, are easily visualized. With the combination of Linux kernel level and DPDK application level events, all the information is present to identify problems and get all the relevant details to understand the root cause, whether at kernel or application level.

5.7 Overhead Analysis

This section evaluates the performance of our framework by analyzing and breaking down the overhead that it induces. The overhead incurred by a performance analysis tool often

represents the first threat to its usability. Besides, the behavior of the monitored application may be altered when the collection of performance data incurs considerable overhead, thus affecting the accuracy of the proposed analyses. Hence, we calculate the overhead introduced by the data collector module of our framework in terms of CPU usage and disk space. Furthermore, while not as critical, we assess the analysis overhead by measuring the time required for Trace Compass to load the traces and build the required analysis data structures.

To estimate the overhead incurred by the data collection phase, we ran a set of benchmarks targeting several DPDK applications under various configurations. Hence, we report in the following the overhead of tracing three applications available in the official GitHub repository of the DPDK project: the Eventdev pipeline application, the IP pipeline application, and the testPMD application. We executed the two first applications according to the setup described in the first and second use cases. Nevertheless, we executed the third application using its default configuration. Hence, the testPMD application was executed using the input/output forwarding mode, which uses one single thread to pull packets from one port and transmit them to the next port. In our benchmarks, as illustrated by Table 5.4, we varied the bit rate of the generated traffic, and calculated the added overhead by measuring the variation in the application forwarding performance, when tracing is enabled and when it is disabled. It is important to note that we pinned the threads of the DPDK applications on isolated CPUs (using the kernel parameter *isolcpus*) in all our experiments. Thereby, we prevent the Linux scheduler from interrupting those threads and eliminate any interference from the other running threads (e.g., kernel threads). We ran the benchmark for 60 seconds, and measured the size of the generated traces, plus the time required for their analysis.

The benchmark results presented in Table 5.4 show that the tracing overhead of the three applications is very low (<0.2%). They also show that this overhead varies depending on the target application. For example, for a traffic bit rate of 1 Gbps, the overhead induced by tracing the Eventdev application was 0.12% while it was 0.02% for the TestPMD application. The cause of this variation is that the data path implemented by the first application is more complex and involves the usage of more DPDK libraries and drivers (e.g., driver of the Eventdev software implementation). Conversely, the processing performed by the TestPMD application on the inbound traffic, according to the input/output forwarding mode, was minimal since it simply involves copying packets from one port to another. Therefore, this forwarding mode requires the usage of fewer DPDK libraries (i.e., EAL, Vhost, and Ethernet RX/TX adapter libraries), which limits the rate of event issued. As explained earlier, the tracing overhead is tightly related to the frequency of event emission, which is significantly influenced by the granularity of the instrumentation and the number of enabled tracepoints.

Moreover, the results in Table 5.4 reveal that the traffic bit rate slightly impacts the tracing overhead. We can explain this by the fact that, in order to collect stats about traffic reception and transmission, we instrumented DPDK libraries intended to implement the applications data paths. For instance, each time one of the benchmarked applications uses the API function *rte_eth_dequeue_burst* to retrieve a burst of packets from an Ethernet NIC, the event corresponding to the added tracepoint is triggered. Therefore, when the rate of packet reception and transmission by the target application increases, the frequency of event issuing increases in turn. Consequently, a higher throughput event stream causes the tracer to impose more overhead on the traced application.

Similarly, we can also remark that the trace size is closely tied to the transmission bit rate, which seems logical since the trace grows according to the frequency of events emission. We can also note the large traces sizes, which increases linearly with the traffic bit rate. Hence, starting a 60-seconds tracing session to monitor the execution of the Eventdev application, processing 1 Gbps of inbound traffic, resulted in a trace file of 742 MB. The overwhelming amount of low-level events, often counted in millions and even billions, represents the main limitation of the current tracing tools. Unfortunately, this limitation may hamper the scalability of our framework when the DPDK applications are exposed to a heavy traffic load. Luckily, the literature reports several techniques that we can use to reduce the traces size, such as selective tracing, trace compression, and filtering and generalization of events [121]. Also, we can effortlessly observe that the analysis time is correlated to the trace sizes. This makes sense because the larger is the trace file, the longer the time required to read and process the events it contains.

It is worth noting that native tracing support was added to DPDK starting from release 20.05. This support comes as a tracing library enabling the developers to instrument their applications and the DPDK control and datapath APIs. The added tracing library presents many interesting features that we usually find in modern tracers, such as enabling and disabling tracepoints during runtime and saving traces on disk at request. To compare the performance of LTTng and the DPDK native tracer, we conducted a set of benchmarks aiming to evaluate the tracing cost for both tracers [2]. For the sake of accuracy, we only calculate the time required by the tracer to record the event in its memory buffers, and we do not consider the time required to save it on disk. Fig. 5.19 reports the results of these benchmarks and shows the latency required for each tracer to emit one event, depending on the type and the number of arguments. This figure clearly shows that the induced tracing overhead increases depending on the type and the number of arguments [3]. It also shows that the DPDK native

2. LTTng version 2.12 and DPDK version 21.05
3. Our results indicate that LTTng requires 318 cycles to emit an argument-less event while the DPDK

tracer presents a trace overhead that is much lower than that of LTTng.

The main reasons behind the outstanding performance of the native tracer are twofold. First, this tracer leverages the DPDK threading model to use per-thread variables and does not have to support asynchronous signals. It thus avoids costly atomic instructions and memory barriers otherwise needed. Second, the usage of huge pages minimizes the overhead due to TLB misses. Although our benchmarks show that the general purpose LTTng tracer presents more overhead than the DPDK specialized tracer, it remains an efficient and highly optimized tracer. Nevertheless, its generality and flexibility comes at a price. Since both LTTng and the DPDK native tracer generate traces in the CTF format, it is then possible to benefit from less tracing overhead, without though having to modify our analyses.

Table 5.4 Overhead induced by the collection and analysis of the performance data

Benchmarked Application	TX Bit Rate (Mbps)	Tracing Overhead (%)	Trace Size (MB)	Analysis Time (s)
Eventdev Application (*./dpdk-eventdev_pipeline*)	1000	0.12	742	83
	500	0.08	351	41
	250	0.08	188	20
IP Pipeline Application (*./dpdk-ip_pipeline*)	1000	0.07	538	72
	500	0.06	270	35
	250	0.06	133	15
TestPMD Application (*./dpdk-testpmd*)	1000	0.02	54	7
	500	0.01	28	3
	250	0.01	14	2

5.8 Conclusion

DPDK is an open-source software project developed and maintained by a vibrant community and many leaders in the network industry, including Intel, Cisco, and Mellanox. This project aims to provide a free and high-performance development kit for accelerating data plane packet processing. Hence, DPDK is considered as the most successful implementation of the kernel-bypass technique. In essence, its popularity stems from its ability to deliver high packet processing performance, alongside its support for a wide variety of NICs, CPU architectures, and operating systems (i.e., Linux, Windows, and FreeBSD). Thanks to its distinctive features, it is used by many software products, especially those implementing innovative networking concepts, such as SDN and VNFs. Examples of software using DPDK

native tracer requires only 25 cycles

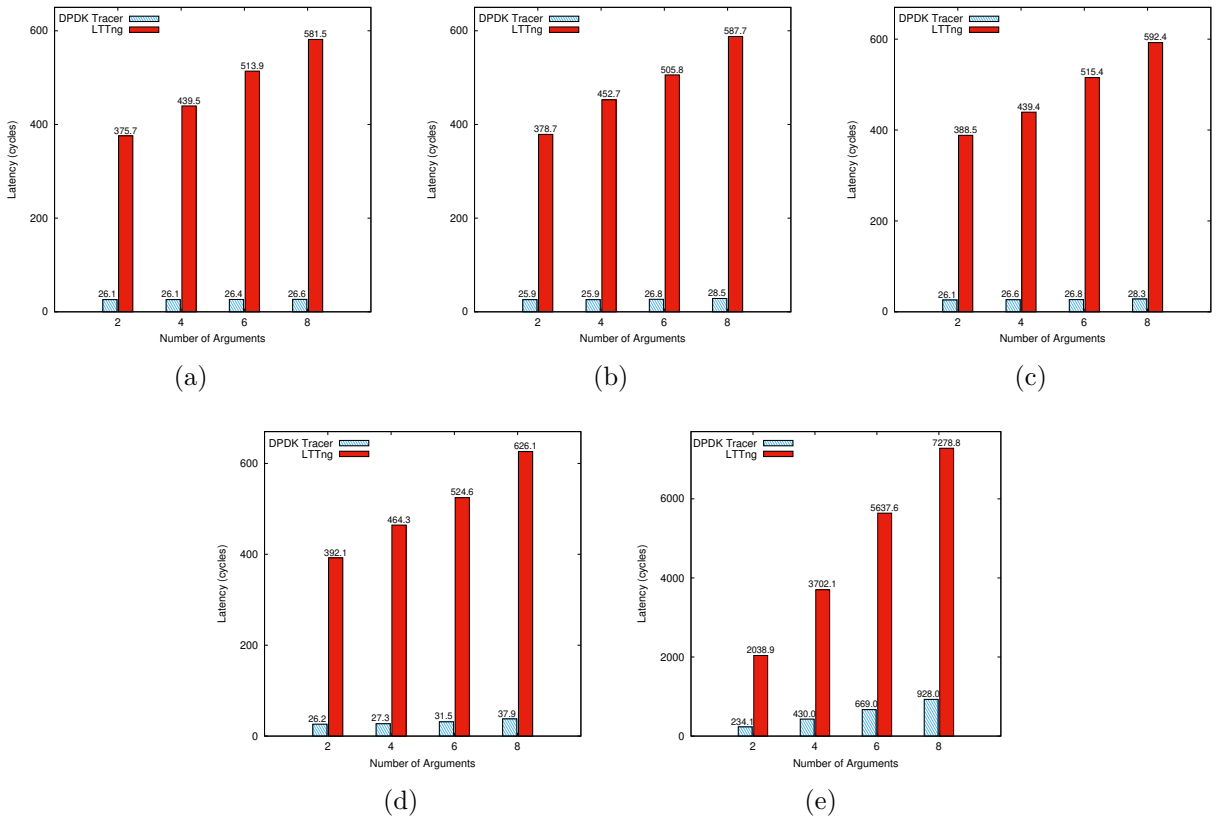

Figure 5.19 Average overhead for recording one event in tracer's memory buffer, depending on the number and type of its arguments: Char, (a); Int, (b); Float, (c); Double, (d); and String (256 characters) (e)

include Open vSWitch (OVS), Fast Data project (FD.io), and Open Daylight. Nevertheless, in the present paper we argued that the current debugging tools dedicated to this kind of software are incapable of detecting intricate performance bugs or diagnosing their root causes.

Seeking to address this problem, we have presented a novel and powerful performance diagnostic tool intended for DPDK applications. This tool can be leveraged by performance analysts to monitor the execution of DPDK applications, detect their performance degradation, and pinpoint their causes. To showcase the usefulness of our tool, we used it to analyze two different performance bottleneck cases, and we were able to determine the root causes of the issues. Furthermore, our overhead analysis shows that the proposed tool has a very low impact on the performance of monitored applications, less than 0.2 % overhead compared to those using non-instrumented DPDK libraries. Our research also highlighted the performance of the DPDK native tracer that has been made available in the recent DPDK versions. We are currently in the process of converting our tracepoints to the format supported by this tracer. Reducing further the overhead of the data collection would make our tool even more

suitable for production use. Luckily, adopting the DPDK native tracer would not constrain us to modify our analyses, since it generates traces in the same format as LTTng.

Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena and EfficiOS.

CHAPITRE 6 ARTICLE 3: PERFORMANCE ANALYSIS OF VIRTUALIZED GPUS THROUGH HOST-BASED TRACING

Authors

Adel Belkhiri and Michel Dagenais
Department of Computer and Software Engineering, École Polytechnique de Montréal, Montreal, H3T 1J4, Canada

E-mail : {adel.belkhiri, michel.dagenais}@polymtl.ca

Submitted to International Journal of High Performance Computing Applications

6.1 Abstract

Cloud computing providers have always been looking for new techniques to make High-Performance Computing (HPC) applications run faster and, therefore, reduce their costs. One of the most successful techniques to achieve this goal is to execute HPC applications on heterogeneous nodes equipped with multi-core processors and specialized hardware accelerators. The Graphics Processing Unit (GPU) is a popular accelerator distinguished by its highly parallel architecture and energy consumption efficiency. While cloud providers have successfully mastered the virtualization of processors and most I/O devices, the virtualization of the GPU has been more challenging. Major manufacturers in the graphics industry, such as Intel, AMD, and NVIDIA, eventually tackled the challenge and proposed virtualization solutions for their GPUs. The literature reports several tools intended to analyze GPU-accelerated applications' performance, but most of them do not support virtual GPUs (vGPUs).

In this article, we propose a performance analysis tool for systems using virtualized GPUs. This tool works within KVM Graphics Technology (KVMGT), the Intel virtualization solution, although its design principles and implementation are general and could apply to other vGPU-based systems. Our tool leverages various tracing techniques to collect low-level performance data and derive relevant metrics. It also provides several auto-synchronized graphical views that give valuable insights into the operation of KVMGT and enable the diagnosis of the performance bottlenecks of vGPU-enabled VMs. The main strengths of the proposed tool are its transparency to VMs and its low overhead. A few use cases are presented to showcase the usefulness of our performance analysis tool.

6.2 Introduction

Nowadays, cloud computing is a well-established paradigm in the Information Technology (IT) industry. This paradigm aims to make the provision of computing resources (i.e., storage, networks, and servers) more flexible. Thanks to it, customers are given on-demand access to a pool of computing resources according to the pay-as-you-go model. In the last few years, cloud providers have consolidated their offerings with accelerators to satisfy the growing need for high-performance computing and drive more businesses to use their services. Accelerators are specialized processors that assist the main central processing unit (CPU) by handling specific types of computation. Thus, when an adequately selected part of the computation running on the CPU is offloaded to one or more specialized accelerators, the system's overall performance improves significantly.

Over time, hardware manufacturers have created various types of special-purpose accelerators. Those accelerators include, but are not limited to, Accelerated Processing Units (APUs), Floating Point Units (FPUs), Digital Signal Processing units (DSPs), Network Processing Unit (NPUs), and GPUs. The focus of this paper is on the GPU, one of the most pervasive accelerators. This device has been initially designed for graphics rendering and image processing. However, its computational power has been increasingly harnessed for parallel number crunching in the past few years. Currently, GPU-accelerated applications are used in many domains, often unrelated to graphics, such as deep learning, financial analytics, and general-purpose scientific calculations.

On the other hand, virtualization is undoubtedly a crucial key enabler for most cloud computing concepts. The role of this technology is to ensure the sharing and multiplexing of virtual resources between tenants. In particular, GPU virtualization allows to increase the sharing of the computing resources, lower the energy consumption of cloud data centers, and eventually reduce their operational costs. Hence, various new applications that harness the power of vGPUs in the cloud have emerged. For instance, deep learning, Virtual Desktop Infrastructure (VDI), and artificial intelligence applications are often vGPU-accelerated.

Whereas CPU and most I/O devices virtualization relied on well-established technologies, GPU virtualization turned out to be more complex. There exist many obstacles that complicate the development of GPU virtualization solutions: First, most GPUs brands are very different at the hardware architectural level. Second, some of the most popular GPU brands, such as those manufactured by NVIDIA, are shipped with closed-source drivers. Third, the lack of a built-in sharing mechanism in most GPU designs represents an additional obstacle. Therefore, when a process uses this type of GPU, it obtains exclusive access to its resources,

and other processes cannot preempt it. GPU drivers developers claim that the overhead of preempting processes is much higher on the GPU than on the CPU because of the large number of cores and registers of the former [1].

Before GPU virtualization solutions were developed, most cloud providers used the passthrough access mode to offer optimized VMs attached directly to the physical GPUs (pGPUs). Google Cloud is a cloud provider that is still using this technique [122]. The passthrough technique presents several limitations, despite the performance it achieves is the highest compared to other virtualization techniques. Examples of those limitations include the non-enabling of GPU sharing between VMs and deactivating a few cloud management techniques, such as live migration. Major GPU vendors, like NVIDIA, AMD, and Intel, have developed full-featured virtualization products that only support their brands. Those products are respectively named NVIDIA GRID [79], AMD MxGPU [78] and GVT-g [73]. While the first two solutions rely on hardware virtualization capabilities, GVT-g (*Graphics Virtualization Technology - Grid generation*) implements a full GPU virtualization through software. This latter solution has been of greatest interest to us since its source code is freely available and included in Linux mainline kernel (since version 4.10).

To operate cloud platforms and manage their growing complexity, efficient and fine-grained monitoring and performance analysis tools are required. Several reasons can explain the need for such tools [123]. First, resources provisioning, a task of paramount importance in cloud computing, can be planned based on performance analyses. Performance data generated by performance analysis tools can help determine the type and the amount of the required resources. Second, those tools can also aid in detecting and diagnosing performance bugs within data centers. Moreover, cloud billing systems can use performance analysis tools to measure resource usage and charge customers based on their consumption.

Most GPU performance analysis tools lack capabilities and do not provide in-depth insights into the operation of this device. Besides, the mechanisms involved in sharing the GPU resources are hardly understood, as is their impact on the performance of virtualized GPUs. Designing tools capable of monitoring and debugging vGPUs in the cloud is indeed challenging for many reasons. Cloud platforms are conceptually composed of several layers (i.e., hardware, middleware, host/guest OSs, and user applications). These layers increase the GPU resources' isolation and abstraction, thus hindering the visibility and complicating the diagnosis of performance problems.

Moreover, because of the disparity of the GPU virtualization approaches and technologies,

1. Some recent GPUs include support for preemption at the kernel, thread, and instruction levels (i.g. NVIDIA Pascal GPU)

developing tools capable of monitoring vGPUs becomes complex. This explains why the number of tools dedicated to profiling and debugging GPU-based applications is limited compared to those intended for the CPU. Furthermore, these tools are often closed source and architecture-dependent.

This paper aims to propose a novel performance analysis framework capable of analyzing the performance of GPUs virtualized based on the GVT-g technology. This framework uses an agent-less VM tracing approach to collect performance data with a low-overhead cost. Tracing is an effective technique to collect low-level performance data from complex systems. Our framework does not require installing any agent or tool inside VMs because we limit the tracing to the kernel space of the host Operating System (OS). The benefits of this approach are numerous : (1) ease of deployment, (2) keeping the privacy of the VM owner, and finally (3) cost-effectiveness. In summary, the contributions brought by this paper are the following:

1. Efficient instrumentation of the KVMGT source code and implementation of modules within Linux Trace Toolkit Next Generation (LTTng) [106], a low-overhead tracer, to collect performance data from this virtualization solution and the Intel GPU driver.
2. Building a unified stateful model to filter and organize the collected tracing data in a way that facilitates its processing. Our analyses leverage this model to compute relevant performance metrics.
3. Implementation of many analyses within Trace Compass [87], an open-source performance analyzer, and designing a set of auto-synchronized graphical views. The practitioner can leverage those views to shed light on the internal mechanisms of KVMGT and diagnose certain performance issues related to the usage of vGPUs.

The remainder of this paper is organized as follows: Section 6.3 presents a literature review of the approaches and techniques used to virtualize GPUs. It also presents a subset of the most popular GPU profiling and performance analysis tools. Section 6.4 shows the design details of our framework and describes some relevant performance metrics. Section 6.5 introduces three use cases that showcase the effectiveness of our framework. Section 6.6 evaluates and discusses the overhead incurred by our framework. Finally, section 6.7 concludes this paper.

6.3 Background and related work

Cloud computing providers aim to provide their customers with high-performance computing at a low cost. Therefore, the recent advances in I/O virtualization methods have encouraged them to include vGPU-enabled VMs in their offerings. In this section, we overview the

approaches and techniques used to virtualize the GPU. We also present few monitoring tools that support the debugging and profiling of this device.

6.3.1 GPU Virtualization Approaches

Virtualizing the GPUs, unlike other I/O devices, is challenging for two reasons [124]. The first reason is that most GPU drivers provided by GPU vendors are closed source due to intellectual property protection. The second reason is the absence of standards on which modern GPUs can be designed. Despite all these obstacles, many GPU virtualization solutions were implemented based on the following approaches.

PCI Passthrough

This approach uses hardware virtualization technologies, such as Intel VT-d [125] and AMD-Vi [126], to virtualize the GPU at the hardware level. The PCI Passthrough approach enables the GPU driver of the VM to connect to the host pGPU directly. It thereby provides direct access to the GPU memory, through Direct Memory Access (DMA), without any intervention from the hypervisor. The GPU interruptions are, likewise, directly mapped to the guest OS. A VM accessing a pGPU based on this approach takes advantage of all the GPU capabilities with a near-native performance [127]. However, this approach can not share the GPU between multiple VMs because it dedicates all its resources to one VM. This limitation makes PCI Passthrough by far the most expensive approach among all the others. Nevertheless, many cloud providers, such as Google Cloud, use this approach to provide their customers with GPU-enabled instances [122].

SR-IOV (Single Root – Input/Output Virtualization)

Unlike PCI Passthrough, the SR-IOV is capable of sharing a single physical device between multiple VMs. SR-IOV is an extension to the PCI Express (PCI-e) specifications which allows isolating the resources of a PCI device by dividing its hardware functions into Physical Functions (PFs) and Virtual Functions (VFs). The PFs are used to configure, control, and discover the features of the device. One PF can be connected to several VFs. A VF is a light PCI-e function that appears as a real PCI-e device for the guest OS if the host OS and the hypervisor support the SR-IOV specifications. Most modern hypervisors such as KVM, Xen, VMware, and Microsoft HyperV support SR-IOV [128,129]. As in the previous approach, the guest driver accesses the PCI device resources transparently. The performance achieved is though slightly lower than that of PCI Passthrough. In addition, this virtualization approach

does not enable the VMs live migration.

Based on the SR-IOV specifications, companies like NVIDIA and AMD have released new GPUs that support virtualization at the hardware level. For instance, the NVIDIA GRID virtualization [79] presents to each VM a vGPU that the guest OS can access in the same way as with the Passthrough approach. AMD has released MxGPU (*Multi-user GPU*) [78], a hardware-assisted virtualization technology that is integrated in its GPU FirePro S-series. In summary, the SR-IOV virtualization approach presents several advantages. First, each VM has access to an equal share of the pGPU regardless of its workload. Second, the performance of vGPUs is easily predictable since the number of resources dedicated to each VM is fixed at the hardware level. Finally, the SR-IOV-based virtualization approach ensures isolation between the vGPUs and thus more security for the VMs.

API remoting

This approach involves installing a front-end in the guest OS to intercept the application API calls to well-known GPU frameworks such as OpenGL, OpenCL, and CUDA. The front-end consists of a wrapper library for those frameworks that intercepts the API calls and sends them to a backend installed in the host OS. The backend forwards the received calls to the GPU driver of the host OS. This latter executes the requests and returns the results to the application via the reverse path. Many proposals [68–72] have used this technique to virtualize the GPU at a high level. The downside of this approach is that the installed wrapper libraries must be kept up-to-date each time the GPU framework is modified.

Para and full virtualization

Many virtualization solutions were developed based on this approach to virtualize the GPU at the driver and hypervisor levels [73–75, 130]. Most of these solutions were developed based on open-source GPU drivers, such as i915, the driver of Intel GPUs. On the other hand, others were developed based on custom GPU drivers implemented via various reverse engineering techniques. The driver of NVIDIA GPUs [131], named *Nouveau*, is an example of such custom drivers. In the paravirtualization approach, the guest OS uses a custom GPU driver, whereas it uses an unmodified GPU driver in the full virtualization approach. The framework proposed in this paper is dedicated to the performance analysis of GVT-g, the Intel technology for GPU virtualization, which is based on this approach.

GVT-g [73] is a product-level virtualization technology developed by Intel for its on-chip GPUs. Initially, this virtualization solution was developed for Xen hypervisor under the

codename XenGT. Later, it was ported to KVM and was known as KVMGT [76]. This later version has supported Broadwell and newer architectures. GVT-g uses the Mediated Pass-Through (MPT) technique to enable VMs to utilize vGPUs efficiently. The MPT technique allows a VM to access the pGPU's performance-critical resources without intervention from the hypervisor. Accessing the GPU's privileged operations, though, is trapped and emulated. According to Intel developers, the most important performance-critical resources of the on-chip GPUs are the Frame and Command buffers. GVT-g distributes the Global Graphics Memory (GGT) among the vGPUs so that they can access these buffers directly while maintaining high isolation between each other. GVT-g presents several advantages compared to other GPU virtualization solutions. For instance, it is so far the only GPU virtualization approach that supports live migration [77]. GVT-g also presents limitations, such as the limited number of vGPUs that it can manage. The literature reports many proposals providing various techniques to increase the density of vGPUs per pGPU [74, 75].

6.3.2 GPU Performance Analysis Tools

Because the number of GPU virtualization solutions is limited and the industry standards are still immature, the number of monitoring tools for vGPUs is relatively low. To the best of our knowledge, almost all these tools are proprietary, and their design and implementation details are obscure since most GPUs drivers are closed-source. Therefore, this section also overviews the GPU monitoring tools that lack support for vGPUs.

NVIDIA vGPU Management and Monitoring Framework [132] is a virtualization software provided by NVIDIA to manage and monitor its vGPUs. This software installs agents in the host and guest machines to give the user insights into how each vGPU-enabled VM consumes the resources of the pGPU. Whether this solution is accessed from the host or the guest, it provides various metrics calculating the percentage of utilization of the vGPU, the pGPU, the graphics engine, and the frame buffers. Moreover, NVIDIA provides the NVIDIA Management Library (NVML) and Windows Management Instrumentation (WMI) interface as part of the GRID Management SDK to encourage the development of third-party monitoring tools. For instance, Goliath Performance Monitor for NVIDIA GPUs [133] is a monitoring tool that is developed based on this SDK.

CodeXL [63] is a profiling and performance analysis software for CPUs and GPUs manufactured by AMD. It is one of the most popular open-source software in the GPUOpen project [134]. CodeXL is empowered with many GPU profiling and debugging features that can help the practitioner analyze the execution of OpenCL and HSA applications and detect potential performance bottlenecks. CodeXL collects performance data from different sources

such as the hardware performance counters, the application, and the OpenCL runtime to enable the developers to identify performance optimization within their applications.

Intel vTune Amplifier [62] is a performance analysis software developed by Intel. This tool can profile the application's execution on local and remote machines and is supported on many platforms, such as Windows, Linux, and macOS. vTune Amplifier offers many profiling capabilities that enable the practitioner to explore the execution of OpenCL kernels and correlate their activities on the CPU and GPU.

Nvidia Nsight [61] is a debugger and profiling software developed by NVIDIA for its GPUs. This software enables developers to debug CPU and GPU code simultaneously and integrates with popular development environments, such as Microsoft Visual Studio and Eclipse. It can also track the application calls to the API functions of many popular GPU frameworks (i.e., OpenGL, Direct3D, and CUDA).

TAU (Tuning and Analysis Utilities) [59] is an open-source profiling and tracing toolkit for parallel and distributed applications. This tool can analyze the performance of GPGPU applications written in CUDA and OpenCL. TAU gathers performance data through GPU performance counters and the instrumentation of the source code of applications. TAU uses many graphical interfaces to present the results of performed analyses.

CLUST (OpenCL User Space Tracepoint) [67] is a tracing-based tool intended for analyzing the performance of OpenCL accelerated programs. The operation of this tool is based on a set of wrapper libraries for the OpenCL framework and a userspace tracer to collect performance data describing the execution of OpenCL kernels. CLUST requires the preloading of the wrapper libraries before launching the GPU-accelerated program to be analyzed. These libraries intercept the API calls for the most relevant functions of the OpenCL and build a call stack based on triggered tracing events. The analysis of this call stack enables the performance analyst to get useful insights into the OpenCL kernels' executions.

LTTng-HSA [66] is another tool that uses the same technique to profile HSA-based programs. Heterogeneous System Architecture (HSA) is a cross-platform specification used to boost the communications between computing devices, such as the CPU and the GPU. The main advantage of this tool is that it does not require the instrumentation of the source code of monitored applications to collect performance data. However, since this tool is closely tied to the HSA framework, it can not provide a system-wide analysis of all the workloads that a GPU can execute.

6.4 Architecture of the Proposed Framework

The main contribution that we propose in this article is a new performance analysis framework for vGPUs managed by KVMGT. The architecture of our framework is composed of three main subsystems: **(1)** the data collection subsystem, **(2)** the analysis subsystem, and **(3)** the visualization subsystem. The first subsystem is responsible for collecting meaningful performance data from KVMGT and the Intel GPU driver. The second subsystem is responsible for processing the collected data and deriving relevant performance metrics. The visualization subsystem consists of several graphical views that display the results of the performance analysis performed by the second subsystem.

We implemented the visualization and analysis subsystems as extensions to Trace Compass [87], an open-source trace analyzer. This tool can read a huge amount of tracing events and analyze them to enable the user to diagnose various performance bottlenecks that affect the traced system. Trace Compass supports several features such as interactive graphical interfaces, pre-built analyses, events filtering, and traces synchronization. We present the design details of those subsystems in the following subsections.

We used KVM [135] to build our virtualization environment since it is the default hypervisor of Linux. The role of KVM is to enable VMs to leverage the hardware virtualization capabilities of the host machine (e.g., VT-x and AMD-V on Intel and AMD machines, respectively). In Linux, KVM is implemented as three kernel modules: *kvm.ko*, *kvm-intel.ko*, and *kvm-amd.ko*. At the user-space level, we used QEMU [136] to run the guest OSs of our VMs.

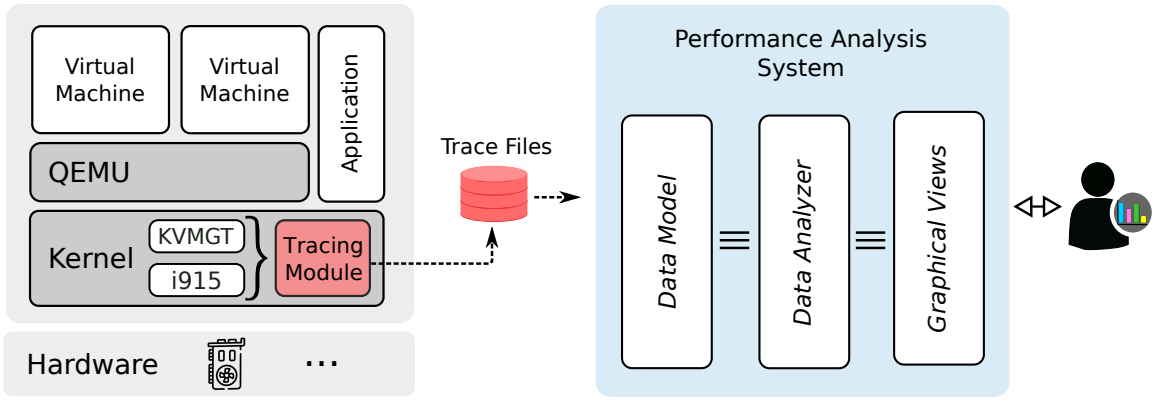

Figure 6.1 Architecture of the proposed framework

6.4.1 Data Collection Subsystem

Logging and tracing are two widely-used techniques to collect performance data from running software. Logging is the process of recording, often in one or several log files, data describing applications' failures and misbehavior, alongside the status of its ongoing operations. Since logging aims to help users quickly troubleshoot bugs, the logging data is usually high-level and human-readable. On the other hand, tracing is the process of recording low-level data that describe software execution. Since tracing aims to diagnose intricate functional bugs and identify performance bottlenecks, the data it collects is usually noisy and overwhelming. Besides, and unlike logging, tracing requires the use of a specialized tool called a tracer. The literature reports several tracers for almost all modern OS. For instance, for Linux systems, we can mention Ftrace [137], Perf [138], and LTTng.

Our framework leverages the tracing technique to collect the low-level data required for our framework. Since LTTng incurs a very low overhead, we chose it to be our tracer. We limited our tracing to the host machine only to ensure minimum intrusiveness. Moreover, since KVMGT is implemented as a kernel module, named *kvmgt.ko* on Linux systems, we limit our tracing to the kernel space. Hence, our analyses are based on the instrumentation of the source code of the KVMGT module to insert new tracepoints and leveraging a subset of existing tracepoints in the host GPU driver. We present in Table 6.1 a subset of the tracepoints upon which our analyses rely.

6.4.2 Performance Analysis Subsystem

Data modeling

The main goal of this work is to propose a tool capable of investigating vGPU performance issues and analyzing the way GPU-accelerated VMs use the GPU resources. Therefore, it is essential to understand how a GPU request is handled from the time a VM process issues it until the time the hardware executes it and delivers a response back.

Figure 6.2 shows the different states a GPU request can be in during the life cycle of its existence in the system. First, a process inside a GPU-accelerated VM issues a request as part of the GPGPU or graphics processing operations. The GPU driver of the VM receives this request and inserts it in its waiting queue **(1)**. Because a GPU request may depend on other requests, its dependencies must be resolved before it becomes in the state "Submitted" **(2)**. The GPU driver of the guest OS removes the request from its waiting queue and sends it to the vGPU to execute it if all the previous queued requests are already executed **(3)**. Then, KVMGT traps the request and place it in the waiting queue of the vGPU attached

Table 6.1 Tracepoints Description

Tracepoint	Description
gvt_workload_queue	This event indicates that a GPU request has been added to the queue of the vGPU, after being issued by a process running in a VM.
gvt_workload_submit	This event indicates that this request has been forwarded by KVMGT threads to the host graphics driver (i915).
gvt_workload_complete	This event fires when the execution of the GPU request is finished and removed from KVMGT data structures.
gvt_sched_switch	This event reports the time slice left for each vGPU every time it is scheduled out.
i915_gem_request_add	This event indicates that the GPU request has been queued by the host GPU driver.
i915_gem_request_submit	This event fires when all the dependencies of the request are resolved.
i915_gem_request_in	This event indicates that the request has been sent to the hardware to be executed.
i915_gem_request_out	This event indicates that the request has been removed from the host GPU driver data structures.
i915_intel_engine_noti	This event indicates that one of the GPU engines has finished the execution of a GPU request.

to the VM **(4)**. When the vGPUs scheduler selects that vGPU to be scheduled in, the request is then forwarded to the host GPU driver **(5)** which in turn inserts it in another waiting queue **(6)**. Finally, when the dependencies of the request are resolved **(7)** and all the previous queued requests are executed, the host GPU driver removes the request from its waiting queue and sends it to the hardware for execution **(8)**. When the request finishes its execution, KVMGT is notified **(9)** and, in turn, it notifies the guest OS through the injection of a virtual interrupt.

Modeling the collected performance data is mandatory to compute relevant performance metrics, such as the waiting latency and execution time of GPU requests. To this end, we used Trace Compass [87], a powerful trace analyzer that integrates several interesting pre-built LTTng analyses. These analyses save the data representing the state of the system to be monitored in a disk-based data structure called State History Tree (SHT) [88]. The state of the system that is used by our analysis is stored in an attribute-tree data structure (Figure

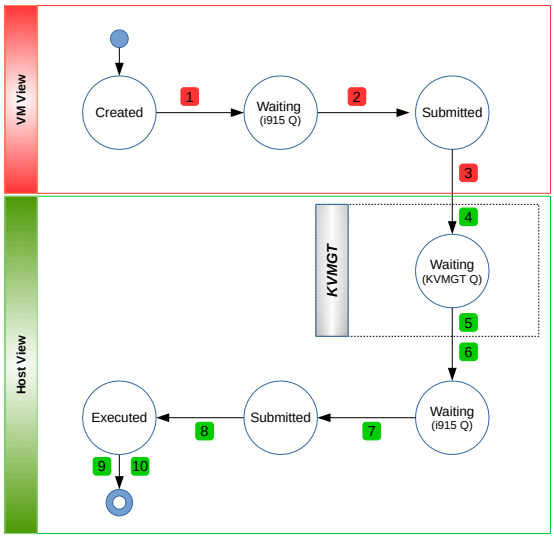

Figure 6.2 States of a GPU request along with the corresponding tracing events

Table 6.2 Captions of Figure 2

#	Event
1	`i915_gem_request_add`
2	`i915_gem_request_submit`
3	`i915_gem_request_in`
4	`gvt_workload_queue`
5	`gvt_workload_submit`
6	`i915_gem_request_add`
7	`i915_gem_request_submit`
8	`i915_gem_request_in`
9	`intel_engine_notify`
10	`i915_gem_request_out`

6.3). This data structure exhibits sets of hierarchical attributes, which are updated each time one of the events presented in section 6.4.2 is handled. Thus, according to our model, one or many vGPUs can be attached to one pGPU. Both a pGPU and a vGPU each have their Driver Waiting Queue. A GPU request is identified by its sequence number (*seqno*), its context (*ctx*) and the engine to which it should be sent for execution (*ring*).

Performance metrics

A performance metric is a quantitative measurement of the performance of the system to be monitored. Mainly, performance metrics help assess the status of a system and debug its performance problems. In our framework, we defined the following performance metrics.

GPU Utilization. We use this metric to measure the occupancy of the pGPU. It indicates the percentage of time during which the GPU was busy. Because the GPU uses various engines to stream several commands simultaneously, we chose this metric to be per engine. We compute this metric using the following formula. It is the sum of the requests execution durations divided by the observation period.

$$Utilization = \sum_{i=0}^{n} \frac{te(Request_i) - ts(Request_i)}{T_{end} - T_{start}} \times 100 \tag{6.1}$$

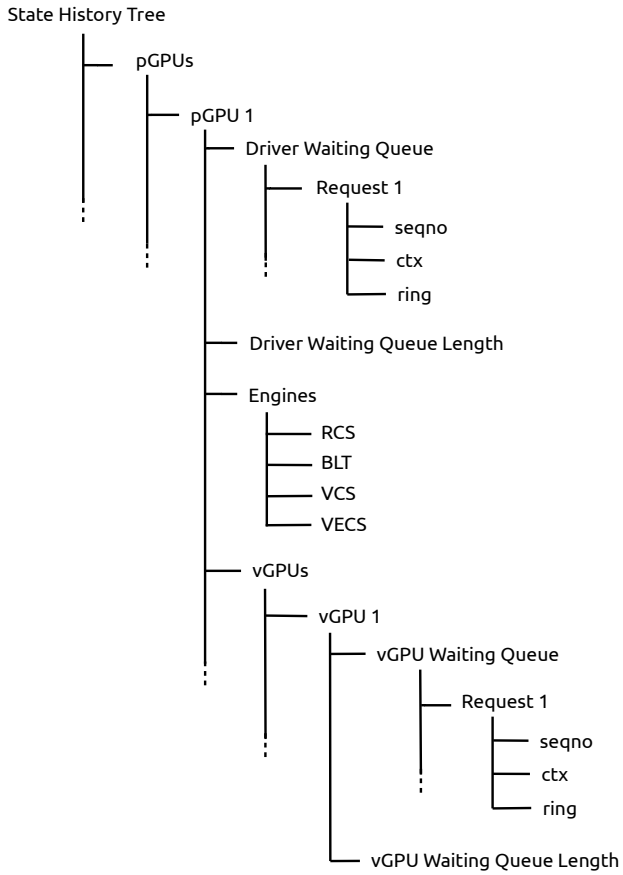

Figure 6.3 Excerpt of the tree-based data structure upon which our analyses are based

Waiting Queue Length. The length of a waiting queue denotes the number of requests waiting to be executed by the hardware. This metric can help spot a bottleneck or a performance degradation when combined with other metrics. The Waiting Queue Length metric varies depending on the rate at which requests are issued, the size of requests, and the rate at which the hardware executes them. As Figure 6.4 shows, this metric is a counter incremented whenever a request is inserted in the queue and decremented whenever a request is removed.

Requests Latencies. As we have mentioned before, a GPU request is not directly executed by the hardware, but it spends some time in many queues waiting for the pGPU to be available. While the execution time of GPU workloads is related to the hardware performance, it is evident that the performance of a GPU virtualization solution is proportional to the amount of time spent by GPU requests in the different waiting queues. The request latency metric indicates the time needed to process a GPU request from the moment it is issued

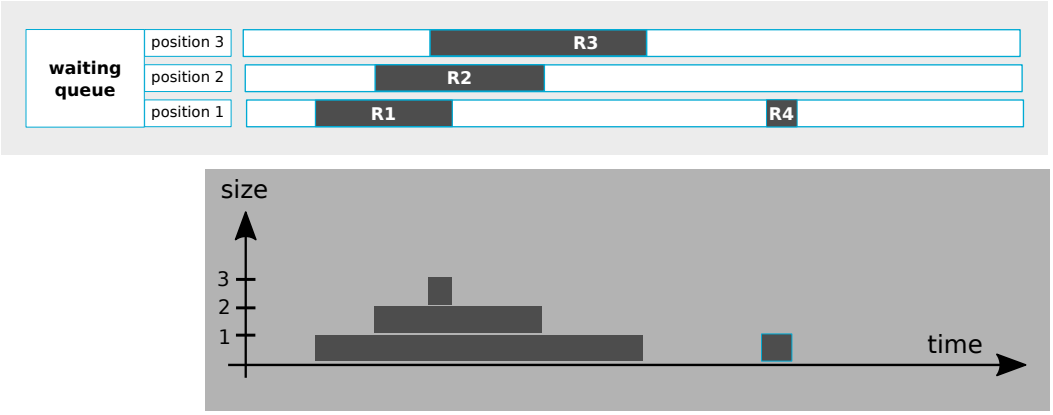

Figure 6.4 Evolution of the waiting queue length

until completed. In our analysis, as shown in Figure 6.5, the request latency includes the waiting time and the execution time. The waiting time starts when the request is inserted in the waiting queue of the guest GPU driver and ends when it is submitted to the hardware for execution. The execution time denotes the time needed by a GPU engine to execute the request. We can calculate the average latency of a GPU request using the formula below.

$$Avg.\,latency = \sum_{i=0}^{n} \frac{waiting\,time(Request_i) + exec\,time(Request_i)}{n} \tag{6.2}$$

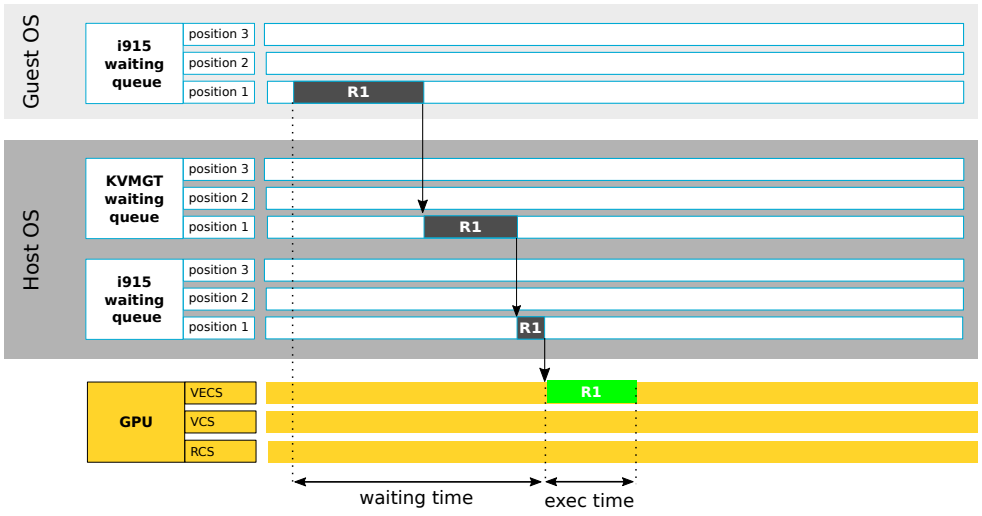

Figure 6.5 GPU requests latency metric

6.5 Use Cases

In this section, we leverage our framework to shed light on the internal operating of KVMGT. The scheduler is undoubtedly the most critical mechanism that could impact the performance of a GPU virtualization solution. Hence, the operation of KVMGT is based on a kernel thread named *gvt_service_thr* and a set of other threads called *workloads thread X*. The "X" is a number that identifies one of the engines supported by the pGPU, such as the render and video command streamers. The thread *gvt_service_thr* is responsible for vGPUs scheduling and context switches handling. Once a vGPU is scheduled in, the *workloads thread X* threads are woken up and, concurrently, they browse its waiting queue to forward GPU requests to the host GPU driver.

This section aims to validate our approach and show the effectiveness of our framework in debugging and investigating some performance issues of GPU-accelerated VMs. We present in Table 6.3 the experimental setup used to conduct our experiments.

6.5.1 KVMGT Scheduling Algorithm

The aim of this use case is to leverage our framework to shed light on the operation of KVMGT scheduler. Hence, in a first experiment, we created two GPU-accelerated VMs (VM1 and VM2) sharing the same pGPU through KVMGT. The vGPUs of the two VMs have similar configurations and thus the same weight. The weight is an integer value set by KVMGT at instantiation time and represents the vGPU's share of the pGPU resources. Hence, we executed in VM1 several benchmarks from the GPU benchmark suite Rodinia [139]. We executed again the same benchmarks immediately after shutting down VM2. Figure 6.6 depicts the time duration of the two executions. As we can notice, the benchmark execution took much longer when VM2 was idle (first execution) than when it was shut down (second execution). The difference between the two execution durations is more noticeable in the case of Gaussian, Lud and Heartwall benchmarks. In this context, we define a VM as idle if it does not execute significant GPU workloads.

To diagnose the cause of the performance gap between the two executions, we executed and traced the Gaussian benchmark in VM1 according to the same setup. Figure 6.7 shows a view from our framework which displays the execution time and the GPU usage related to the benchmark execution. As we can see in this figure, the first benchmark execution lasted 59s with an average GPU usage almost equal to 45% whereas the second execution of the

Table 6.3 Hardware and software experimental configurations

Host Machine		**Virtual Machine**	
CPU	i7-7500U CPU @2.70GHz	vCPU	One vCPU (Kabylake Model)
GPU	HD Graphics 620 (KabyLake)	vGPU Mem.	Low: 64 MB / High: 384 MB
RAM	16 GB	RAM	2GB
OS	Ubuntu 16.04 (Kernel 4.14.15)	OS	Ubuntu 16.04 (Kernel 4.15)
Qemu	Version 2.11.1		
LTTng	Version 2.10.8		

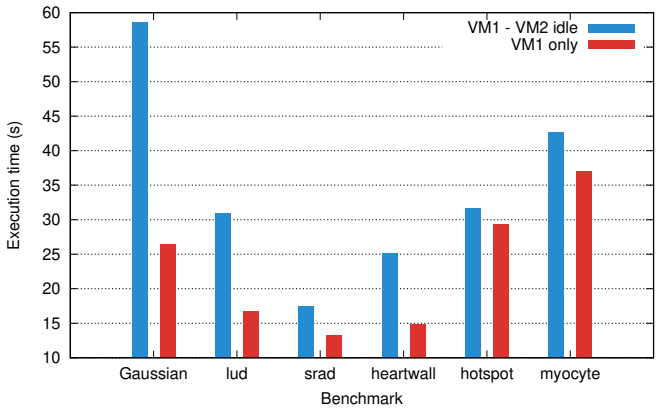

Figure 6.6 Latencies of benchmarks when executed in multi-VMs and VM only contexts

benchmark lasted 27s with a GPU usage of almost 80%. This view clearly shows that the performance of VM1 was deeply affected by the presence of VM2, the idle VM.

To comprehend the cause of this performance loss, we leveraged our framework to shed light on the KVMGT scheduler internals (Figure 6.8). In KVMGT, the *gvt_service_thr* thread represents the scheduler and is therefore responsible for executing the vGPUs scheduling algorithm. This scheduler uses time multiplexing to share the pGPU resources between several vGPUs. Moreover, it simultaneously schedules all the GPU engines to avoid synchronization deadlocks and reduce the scheduling algorithm complexity. Based on the weight of each vGPU, the scheduler computes a time slice representing its share of the GPU time (see the formula below). The time slices are thus proportional to the vGPUs weights and are, in addition, cumulative. The scheduler subtracts the sum of execution times of requests sent by a vGPU from its time slice. When a vGPU consumes its entire time slice during a period (100 ms), the scheduler schedules it out. Time slices are recomputed every ten periods, that is 1 second, and debts and lefts from previous periods are forgotten.

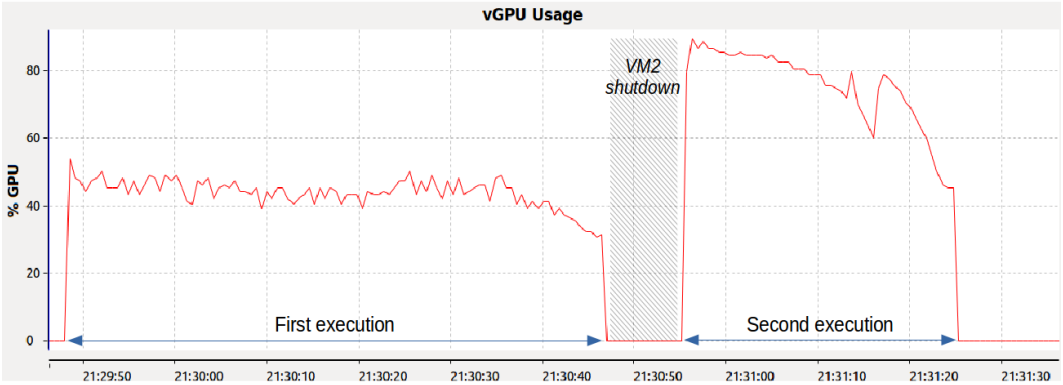

Figure 6.7 GPU usage related to two executions of Gaussian benchmark in VM1

$$Time\ Slice = 100ms \times \frac{vGPU_{weight}}{\sum_{i=1}^{n} vGPU^{i}_{weight}} \tag{6.3}$$

Example. Supposedly, we have two vGPUs sharing the resources of the host GPU and configured with the same weight (like in our current use case). Then the KVMGT scheduler would grant to each vGPU a time slice of 50 ms per period to execute its workloads. If one of the two mentioned vGPUs is removed from the host machine, the other vGPU will be granted a time slice equal to 100ms (complete period). □

As explained earlier, the KVMGT scheduler uses a round-robin scheduling policy based on the vGPUs weights. This scheduler ensures that vGPUs configured with higher weight values will get more pGPU time. Hence, the *gvt_service_thr* thread responsible for implementing this scheduling policy is cadenced with a timer that fires every 1ms. This thread sleeps for 1 ms and then iterates through the available vGPUs' queues to check for the presence of GPU requests. If one vGPU has enqueued one or several requests, then the *gvt_service_thr* thread verifies whether the time slice of that vGPU has already been consumed and, if not, sets it as the active vGPU. Furthermore, it wakes up the *workloads threads X* to process the GPU requests of the active vGPU and to update some statistics related to the previous vGPU. The scheduler tries to schedule another vGPU if the time slice of the active vGPU becomes less or equal to zero. Figure 6.8 depicted the operation of the scheduler as explained above. For instance, when vGPU1 utterly consumed its time slice in the first execution of the benchmark, the scheduler scheduled it out. The requests of vGPU1 remained in its waiting queue and were not forwarded to the host GPU driver until it obtained a new time slice in the next period.

To sum up, the views of our framework reveal that the KVMGT scheduler does not allow

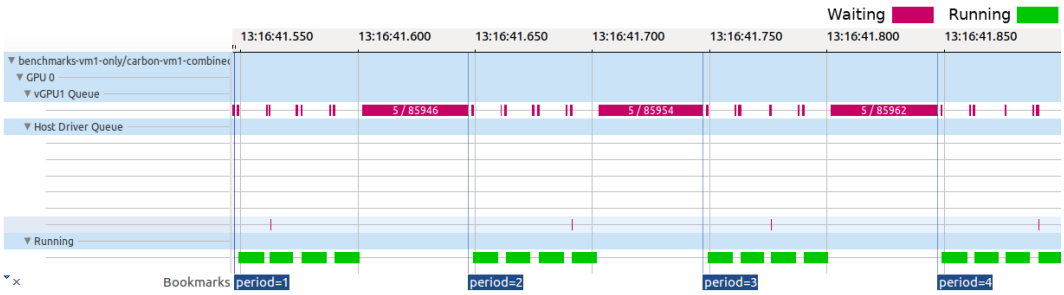

Figure 6.8 Wait latencies of GPU requests in KVMGT and host driver queues along with their execution times

one VM to use another VM's time slice, even when the vGPU of the latter is idle. Although this makes sense in a cloud computing context (pay-as-you-go model), this operating is undoubtedly inadequate to enable vGPU acceleration in a private cloud data center. Using the KVMGT scheduling algorithm in those environments would not ensure the high utilization of the pGPU resources. Therefore, we think modifying the KVMGT scheduler algorithm to support this latter use case would be beneficial. Hence, the new algorithm should enable one VM to execute its GPU workloads when no VM competes for the host GPU resources. Choosing which scheduling algorithm to activate can be made at startup based on a configuration option.

6.5.2 Contention Between vGPUs

This second use case aims to study to what extent the performance of a vGPU-enabled VM could be affected by other VMs. In this experiment, we have two collocated VMs sharing the host GPU resources through KVMGT. We configured the vGPUs of these VMs identically as illustrated by Table 6.3. We launched in VM1 the Gaussian benchmark and in VM2 many benchmarks that are characterized by various degrees of intensity of access to the host GPU resources (e.g., small and big workloads, different request issuing frequencies, etc.). For instance, Figure 6.9 illustrates the impact of the size of workloads issued by those benchmarks on the execution time of the Gaussian benchmark. As we can observe in this figure, the execution time of this benchmark significantly increased when this latter was concurrently executed with the Hotspot benchmark. Figure 6.10 sows that the Gaussian benchmark GPU usage did not exceed 20% in the best cases when the Hotspot benchmark was running. However, it increased to reach 40% as soon as the execution of the Hotspot benchmark had finished.

This experiment proves that the performance of one vGPU might significantly be affected

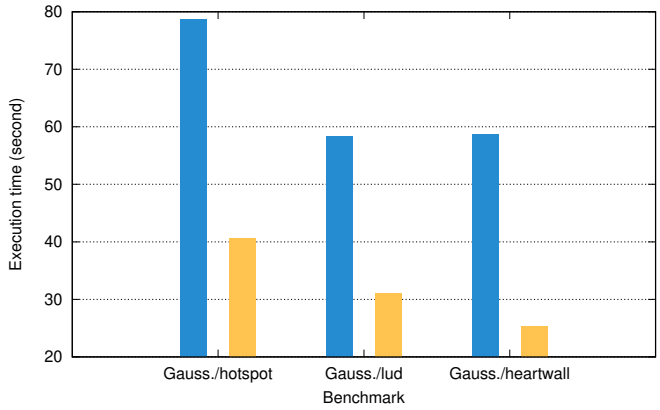

Figure 6.9 The execution time of Gaussian benchmark (VM1) increased when it was concurrently executed with Hotspot benchmark (VM2). Contention on GPU resources between the two VMs is a plausible cause

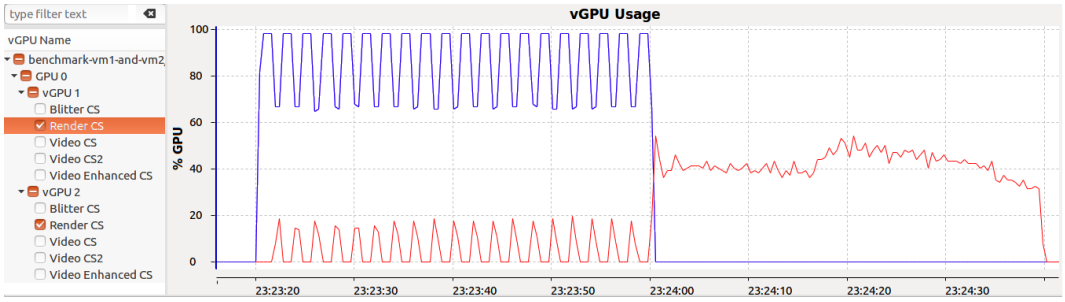

Figure 6.10 GPU usage during a concurrent execution of Gaussian/VM1 (red) and Hotspot/VM2 (blue) benchmarks

by another vGPU. Our diagnosis shows that the root cause of this behavior is the disparity of workload sizes: almost 18ms for vGPU1 workloads (Gaussian benchmark) and 800ms for vGPU2 workloads (Hotspot benchmark). Because the i915 GPU driver scheduler does not support preemptive multitasking, it cannot interrupt the execution of a workload once it is started. Hence, the KVMGT scheduler implements a coarse-grained sharing of the host GPU usage time to overcome the i915 lack of preemption. As explained above, the scheduler grants time slices for the vGPUs that are proportional to their weights. It also subtracts the execution times of GPU requests from the vGPU time slice and considers the debts and lefts from previous periods. These debts and lefts are only forgotten after ten periods. This explains why the considerable size of workload issues by vGPU2 caused the vGPU1 to perform poorly.

This approach presents the advantage of improving the scheduling fairness between vGPUs and therefore increasing the responsiveness of applications running inside VMs. Nevertheless,

its downside is that the time-sharing of the host GPU is coarse-grained and causes the vGPUs to perform poorly when the size of the workloads issued by a given vGPU becomes considerable (> period). To better explain this idea, let us consider the example illustrated by Figure 6.11. In this example, let's suppose we have two vGPU-enabled VMs configured with equal weights. At the beginning of each period, the scheduler affects a time slice of 50ms to each vGPU. The Figure shows that when the execution of workloads sent by vGPU2 (colored in blue) lasts 200ms, the vGPU1 will have less time to execute its workloads (colored in red).

In summary, this use case shows that the KVMGT scheduler cannot ensure fairness between the available vGPUs in some situations. The main cause of this incapacity is the non-support of the i915 driver for task preemption and how the coarse-grained sharing mechanism is implemented. Since the goal of forgetting the debts and lefts in the scheduling algorithm is to maintain the responsiveness of interactive programs, we propose to deactivate this mechanism if the VMs are only intended to run non-interactive programs.

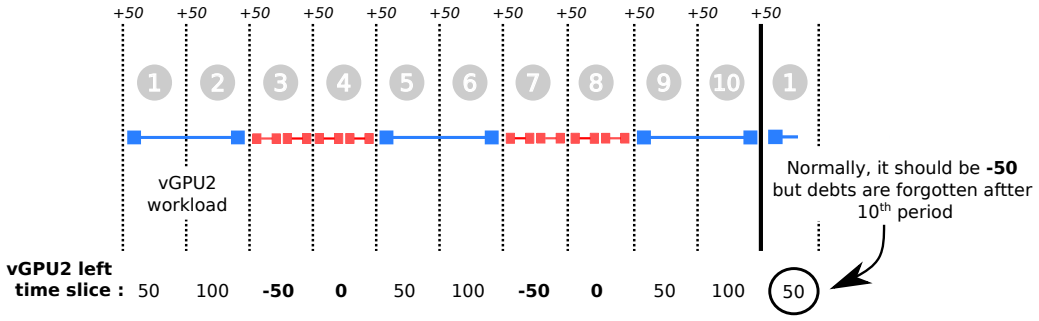

Figure 6.11 When vGPU2 is sending big workloads (blue), the performance of vGPU1 (red) is affected

6.5.3 Virtual Machine Placement

VM migration is an interesting feature that is supported by most modern hypervisors. It consists of relocating a VM when its physical host machine becomes overloaded or underloaded. This feature allows increasing physical resource usage within a data center and respecting the service level agreement. On the other hand, VM placement is conducted as part of the VM migration process and consists of finding the best physical machine to host a VM. Many placement algorithms were designed overtime to accomplish various goals, such as increasing cloud data centers' resource usage and decreasing energy consumption. Studies show that these two factors have a direct and considerable impact on data centers' operational

costs [140].

VM placement algorithms usually rely on efficient monitoring tools to quantify the resources needed by VMs and to discover the performance bottlenecks caused by resource overcommitment. When resources are overcommitted, VMs tend to compete for them by interrupting each other frequently. Unfortunately, this contention often results in an increased overall execution time for running programs. Several VM placement algorithms adopt the strategy of grouping in the same host VMs that seek different resources to solve this issue. For instance, grouping a CPU-intensive VM with another disk-intensive or network-intensive helps improve the resource usage and also the global performance of VMs. Similarly, it is practical to group VMs soliciting CPU and GPU resources alternatively in the same host. Grouping together CPU-bound and GPU-bound VMs allows maximizing the utilization of both compute devices and reducing the frequency of VMs preemption.

Hence, we used our performance analysis framework in this context to improve the VM placement algorithm of our private cloud by taking the GPU usage of running VMs into account and considering it as an additional input. The optimization consists of migrating VMs when their CPU usage goes beyond a threshold for a certain time to another host machine where the hosted VMs are more GPU-bound. This approach helped us increase resource utilization within our cloud and reduce the execution time of applications running inside the hosted VMs. Figure 6.12 shows a view from our tool depicting the execution of a VM alternating CPU and GPU computing phases.

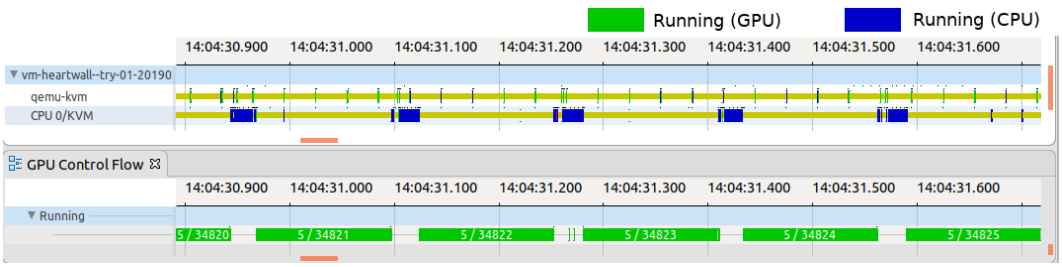

Figure 6.12 A VM alternating CPU and GPU computing phases

To validate our approach, we used the framework developed in [141] to calculate the number of VM preemption. We traced the OS kernels of our cloud host machines and used our framework to tag VMs that were either GPU intensive or CPU intensive. For instance, we identified in some hosts many collocated VMs which were CPU intensive and were constantly preempting each other. Figure 6.13 depicts this category of VMs where, for instance, VM-532 and VM-533 were competing for CPU resources and were preempting each other. At the same time, we identified in other hosts many VMs which were running GPU intensive pro-

grams and were not very dependent on CPU resources. We labeled the two VMs categories as CPU-intensive and GPU-intensive, respectively. The optimized placement algorithm moves the CPU-intensive tagged VMs to new host machines where hosted VMs are GPU-intensive tagged whenever the number of preemption due to CPU overcommitment exceeded a threshold. After applying the optimized algorithm, we noticed that the number of VM preemption has significantly decreased. As seen in Figure 6.14, VM-532 is now rarely interrupted by the new relocated VM-535. The number of VM-532 preemption dropped from 23125 to 15087.

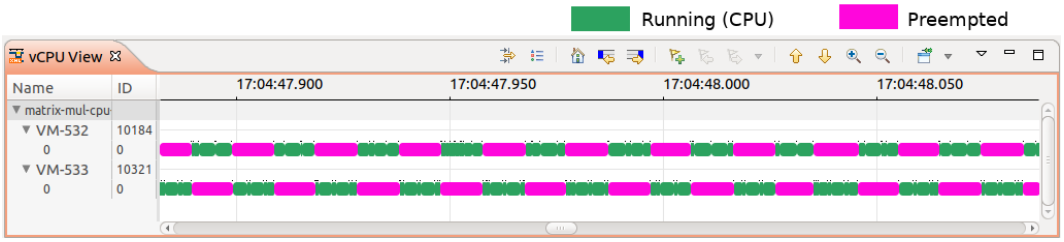

Figure 6.13 VM-532 and VM-533 are preempting each other because they are competing for the same CPU resources

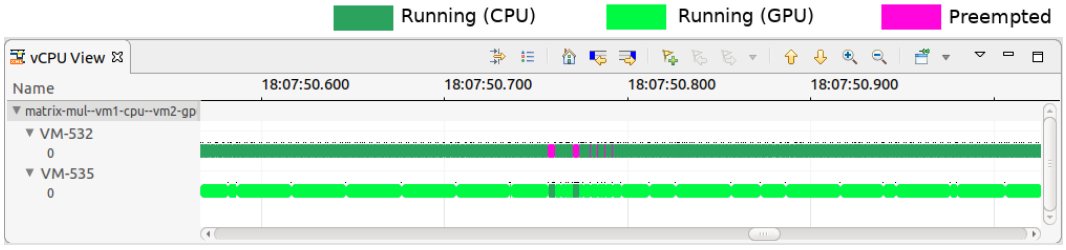

Figure 6.14 After applying the optimized VM placement algorithm, VM-532 is rarely preempted by the GPU intensive VM-535

6.6 Tracing Cost Analysis

In this section, we analyze and evaluate the cost incurred by our framework. Since we use our framework for offline analyses, we consider only the overhead of the tracing data collection phase. As explained earlier, our approach relies on the efficient tracing of the host kernel. Nevertheless, the induced tracing overhead must be maintained extremely low to avoid impacting the traced system's performance or behavior.

To measure the tracing overhead of our approach, we launched a VM whose software and hardware configuration is presented in Table 6.3. We leveraged this VM to run some bench-

marks from the Rodinia benchmark suite [139]. We chose several benchmarks that are characterized by different workload sizes and submission frequencies. Table 6.4 shows that executing these benchmarks with our tracepoints turned on involves a low overhead estimated to 1.01%.

Tracing involves another cost which is the storage on the disk for the generated traces. The considerable trace files size is a common limitation of all tracing tools. Hence, the two factors that impact this tracing cost are the frequency of events and the storage format. Table 6.4 clearly shows that the size of generated traces and the execution times of benchmarks are not correlated. Instead, the size of traces is tightly related to the number of events issued, which in our case reflects the number of submitted GPU requests. Since our analysis needs to activate only a few tracepoints, the size of trace files remains acceptable.

Table 6.4 Tracing overhead

Benchmark	Exec. Time (ms)	With Tracing			Overhead
		Exec. Time (ms)	#Events	Trace size (ko)	
Gaussian	26393.95	26906.97	49255	1228	1.906
Hotspot	29455.27	30161.53	9122	248	2.341
lud	16736.80	16862.21	19866	512	0.743
myocyte	36935.18	37061.90	158587	3788	0.341
srad	13206.65	13238.15	73973	1843	0.237
Heartwall	14795.18	14873.27	4110	148	0.525
				Average	**1.016**

6.7 Conclusion

The GPU is a specialized accelerator endowed with many interesting features, such as its highly parallel structure and energy efficiency. Therefore, it is widely deployed in heterogeneous and embedded systems to accelerate various applications, such as automatic driving, media transcoding, and cloud gaming applications. The advantages of this device have motivated cloud providers to add vGPU-accelerated VMs to their offerings. Hence, provisioning vGPU-accelerated VMs enables cloud providers to sidestep most of the CPU bottlenecks in their data centers. It also enables cloud customers to execute their graphics and general-purpose applications on powerful computing devices. Unfortunately, the literature reports a few monitoring and performance analysis tools dedicated to the GPU, which most of them do not support vGPUs.

This paper proposes a novel performance analysis framework dedicated to analyzing the

performance of vGPUs. We utilized KVMGT, Intel virtualization technology for on-die GPUs, to showcase our framework's capabilities. This framework can help get in-depth insights into the behavior of KVMGT and analyze the performance of vGPU-enabled VMs. We used LTTng, a low-overhead tracer, to collect the low-level data required for our analyses. Hence, we leveraged this tool to trace the OS kernel of the host machine. We also developed several auto-synchronized views in Trace Compass to reveal how vGPU-accelerated VMs share the host GPU resources. Our overhead analysis indicates that the cost of tracing is meager (≈1.01%). Future work will generalize our approach to other GPU virtualization technologies such as MxGPU and NVIDIA Grid. Nowadays, major cloud providers are deploying these two technologies in their data centers.

Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena and EfficiOS.

CHAPITRE 7 ARTICLE 4: VIRTUAL NETWORKS LINK-LAYER TOPOLOGIES DISCOVERY THROUGH HOST-BASED TRACING

Authors

Adel Belkhiri, Ahmad Shahnejat Bushehri, and Michel Dagenais
Department of Computer and Software Engineering, École Polytechnique de Montréal, Montreal, H3T 1J4, Canada

E-mail : {adel.belkhiri, michel.dagenais}@polymtl.ca

Submitted to International Conference on Computer Networks and Communications

7.1 Abstract

The network link-layer topology, also known as the physical topology, describes the networking hardware devices, their corresponding placement, and the interconnection between them. Discovering and maintaining updated information about the network topology is of utmost importance for many protocols and monitoring tools. For instance, the algorithms of most routing protocols include modules in charge of the automatic discovery of the link-layer topology. On the other hand, network virtualization is one of the technologies that enable Virtual Machines (VMs) in a cloud platform to share the same physical network infrastructure. This technology has been one of the most important key players in the Cloud Computing industry. Unfortunately, research communities have put a modest effort into developing efficient algorithms capable of discovering the topology of Virtual Networks (VNs).

In this paper, we address this issue and propose an approach to recover the link-layer topology of a VN from an execution trace. Our approach uses tracing techniques to collect data from the kernels of host machines. Since we limit the data collection only to the host machine, this approach is completely transparent to VMs. Furthermore, we propose a prototype framework that implements our approach.

7.2 Introduction

For many decades, the Internet has been a widely adopted technology that has successfully satisfied various telecommunication needs. Hence, it has brought new communication modes to its users and supported several sophisticated and distributed applications. To enhance

the Internet structure, one needs to diversify network designs, considering the support for the existing and legacy protocols and arrangements. Even so, the current architecture of the Internet can not easily be adapted to any innovative technology, as it requires an inclusive agreement among multiple hardware/software manufacturers [142]. The research community, which witnessed the slow adoption of the IPv6 protocol, argues that the popularity and success of the Internet have been the main reasons for its rigidity. This phenomenon, known as Internet ossification, makes any modification to its current architecture very challenging. Needless to mention that the rise of new computing and business models, such as Cloud Computing, big data, and the Internet of Things (IoT), has fostered the emergence of innovative and architecturally different applications. The main common characteristic of this type of applications is the need for higher bandwidth, flexibility, and resilience. Several researchers believe that Network Virtualization (NV) and Software-Defined Networking (SDN) are the best solutions to circumvent the Internet ossification problem [4, 143].

NV refers to the co-existence of multiple logical and independent networks above the same physical substrate. The main idea behind this paradigm is to decouple network functions, from the physical network equipment on top of which they run [144]. Essentially, a VN is composed of virtual nodes and links. A virtual node represents a software component running network functions (e.g., routing, load balancing, and packets filtering). A virtual link is a logical interconnection between two virtual nodes, which appears as a direct physical link. Hence, NV provides many benefits, such as (1) simplifying the deployment, configuration, and creation of complex networks; (2) reducing the inherent cost of ownership and the maintenance of networking hardware; (3) and increasing the flexibility, security, and availability of networks.

On the other hand, the network topology can be described at three distinct dimensions: link-layer level, network-layer level, and overlay level. The link-layer topology describes the physical connectivity relationships between the link-layer devices (e.g., switches, bridges, and hubs) and how the host machines are connected to them [145]. The network-layer topology, also known as the Internet topology, describes how the hosts, routers, and autonomous systems are connected. An overlay topology is a logical network that is built atop of another network. For instance, peer-to-peer systems are a typical example of overlay networks. The topologies of virtual and overlay networks are prone to quick and unpredictable changes, since they are highly influenced by the participation of nodes via the join and leave mechanism [145].

The link-layer topology discovery is of utmost importance, for many reasons, in different types of networks. First, it allows the operators of conventional networks to compile an inventory

of the available network equipment (e.g., Network Card Interfaces (NICs), switches, and routers). Getting knowledge about the available network resources is essential for managers to make a strategic plan for the company network. For instance, it gives them insights into the hardware purchases that are required to maintain, improve, and extend the network infrastructure [146]. Secondly, many monitoring tools rely on a topology discovery process to detect security threats, such as the presence of malicious nodes, or diagnose performance issues, such as connectivity problems [147]. Similarly, several networking protocols, such as routing protocols, require information about network nodes, their properties (e.g., number of active ports, vendors, and firmware), and interconnection to operate [148].

As with traditional networks, the operation of many applications requires knowledge of the topologies of VNs. These applications include, but are not limited to, load balancing solutions, Intrusion Detection Systems (IDS), and firewalls. Moreover, the purpose of Virtual Network Embedding (VNE), a key concept in network virtualization, is to dynamically find the optimal embedding of many VNs on a physical substrate network [142]. The principal input data required by most VNE algorithms are the topologies of the virtual and substrate networks, alongside the properties of their corresponding resources [149]. The topology discovery of VNs is challenging because of their volatility and the dynamic behavior that characterizes their structures [150]. For example, when the user specifies new requirements for the deployment of the VN, the topology, size, and resources might be prone to quick changes. Hence, many researchers claim that the current topology discovery algorithms are still immature [149, 151].

Although much attention was paid to VNs in the last decade, the research on developing compliant topology discovery algorithms is extremely limited compared the efforts dedicated to traditional networks. The literature only reports proposals for discovering the link-layer topologies of conventional networks. Unfortunately, algorithms proposed for discovering the topologies of traditional networks are not suitable for VNs for many reasons. Most of these algorithms rely on well-known management protocols to collect data from the network forwarding devices. The usage of those protocols incurs a significant overhead, which reduces the efficiency of those algorithms if applied in VNs. Furthermore, VNs are mainly composed of software switches, such as Open vSwitch (OVS) [101] and VPP [152] switches. These virtual devices do not support SNMP natively, which limits the applicability of these algorithms. Also, installing software agents in host machines to access the Forwarding Databases (FDB) of the virtual switches is not always possible, for privacy and security concerns.

In this paper, we present a prototype framework that is capable of discovering the link-layer topologies of VNs. This framework leverages tracing techniques to collect data required for

building the topology. The proposed approach is agent-free, which means that it does not require access to VMs. Moreover, we present two methods to infer, from a kernel execution trace, the topology of a VN, even that spanning over multiple physical host machines. We developed many graphical interfaces within our framework, to illustrate the feasibility of our approach.

The remainder of this paper is organized as follows. Section 7.3 presents a short review of topology discovery algorithms designed for traditional and virtual networks. It also explains what tracing is and presents multiple tracing analysis tools. Section 7.4 describes the design and the implementation details of our framework. Section 7.5 discusses the limits of our approach and evaluates its overhead. Finally, Section 7.6 concludes this article.

7.3 Related Work

To the best of our knowledge, the literature does not report proposals for discovering the link-layer topologies of VNs. Therefore, we survey, in this section, the approaches used to discover the link-layer topologies of traditional networks.

Algorithms for network topology discovery can be classified into two main categories. The algorithms of the first category rely on a central node, called a crawler, to gather information about the topology from the other nodes in the network. On the other hand, those of the second category require the cooperation of all (or at least several) nodes to discover the network topology. In addition, collecting the requisite data for deriving the network topology can be performed passively or actively. The passive measurement consists in passively collecting the traffic of the network, without interfering in its activity. By contrast, the active measurement injects custom traffic in the network, to probe its constitutive elements. Many tools, such as the *traceroute* program, leverages the latter method to determine intermediate nodes in the path between source and recipient nodes.

There exist many proposals to discover the topologies of traditional networks [153–156]. For example, the algorithm proposed in [153] has been designed to discover the topologies of heterogeneous IP networks automatically. This algorithm relies on the Simple Network Management Protocol (SNMP) to query the Address Forwarding Tables (AFTs) available at the level of specific network nodes, such as routers and switches. To infer the topology of a switched domain composed of several subnets, the authors assume that the AFT of each node in the network is complete, correct, and easily accessible by the agent running the algorithm.

Another algorithm was proposed in [154] to discover the topology of Ethernet Local Area Networks (LANs). This algorithm uses SNMP to collect topology information from the AFTs

of network switches. It infers the network topology from only a subset of the information available in the AFTs, thanks to the concept of direction awareness. This capacity to build an accurate network topology using incomplete topology information increases the efficiency of this algorithm in large multi-subnet networks.

Authors in [155] proposed a method to discover the link-layer topology of a network without relying on SNMP, or the information available in the Management Information Base (MIB) data structures. Their approach consists in installing software agents on selected hosts that are directly connected to switches. The proposed method assumes that no link-layer filtering mechanism is enabled in the network. It discovers the topology by transmitting custom Ethernet datagrams, and observing which datagrams have been forwarded by which switches.

Algorithms proposed for discovering the topologies of traditional networks are not suitable for VNs for many reasons. First, most of the algorithms reviewed so far rely on various management protocols (e.g., SNMP) for collecting data from the hardware forwarding devices. Using those protocols incurs a considerable overhead, which makes the algorithms not efficient in VNs. Secondly, VNs are mainly composed of software switches, such as Open vSwitch and Linux native switches. These virtual devices do not support SNMP and, therefore, are transparent to the topology discovery algorithms [157]. Thirdly, installing software agents in host machines to access the FDBs of virtual devices is not always possible, for privacy and security concerns. Because of all these obstacles, researchers tend to make several assumptions about the properties of networks targeted by their algorithms.

This paper proposes an open-source framework that leverages kernel tracing to recover the link-layer topologies of VNs. Tracing is a software engineering technique that software developers use for fine-grained and low-overhead logging. We present the design and implementation details of our framework in the following sections.

7.4 Design of the Proposed Framework

In this section, we propose a framework that can automatically discover the topology of VNs. To demonstrate the effectiveness of our approach, we successfully run several experiments in our LAN. As shown in Figure 7.2, the architecture of our network is composed of many interconnected physical machines that each host several containers. A container is a lightweight virtual machine capable of efficiently running an isolated system on a host. Containers within a single host machine can communicate using the network virtualization capabilities of the Linux kernel. We used the Linux bridge module, to create the virtual switches, and *veth* peer devices to connect containers to them. We opted for Docker to build

our virtualization environment. Fundamentally, Docker is a container-based technology that leverages *namespace* and *cgroups* kernel capabilities, provided by recent Linux distributions, to develop and ship applications into packages called containers.

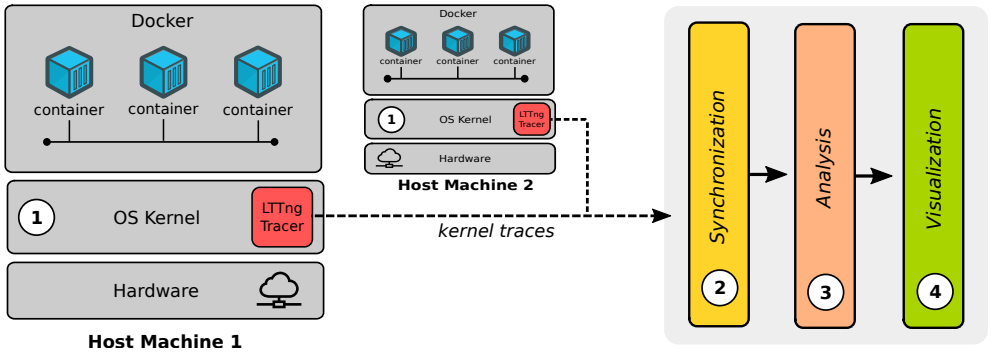

Figure 7.1 Architecture of the proposed framework

As illustrated in Figure 5.3, the proposed framework comprises three subsystems: The first subsystem is responsible for collecting tracing data, while the second one is in charge of analyzing this data and building a link-layer topology. The third subsystem plots the generated topology into a graphical user interface. Our framework is developed according to an offline analysis design. In other words, the analysis phase is deferred until the data collection phase is completed.

Our framework can discover the topology of a VN that is distributed across multiple physical host machines. Figure 5.3 illustrates the steps to be taken for using our framework: 1) Enable the required tracepoints while starting a tracing session using the LTTng tracer in each host machine; 2) Collect the locally generated traces from the host machines and synchronize them offline; 3) Apply the analysis on the generated synchronized trace to build the topology, and finally 4) Visualize the derived topology in a graphical web interface.

7.4.1 Data collection

Tracing is a practical approach to solve difficult problems, such as understanding the behavior of complex systems, unveiling the interactions between their internal mechanisms, and debugging delicate software errors. The simplest way to trace an application or a system is to instrument its source code. Whenever a tracepoint is encountered during the runtime, an event is fired, and its data is saved in memory. Most recent tracers record the payloads of the events, along with various generic information, such as the event name, its timestamp, and the CPU on which it occurred. Often, the tracer dumps its memory buffers to the disk,

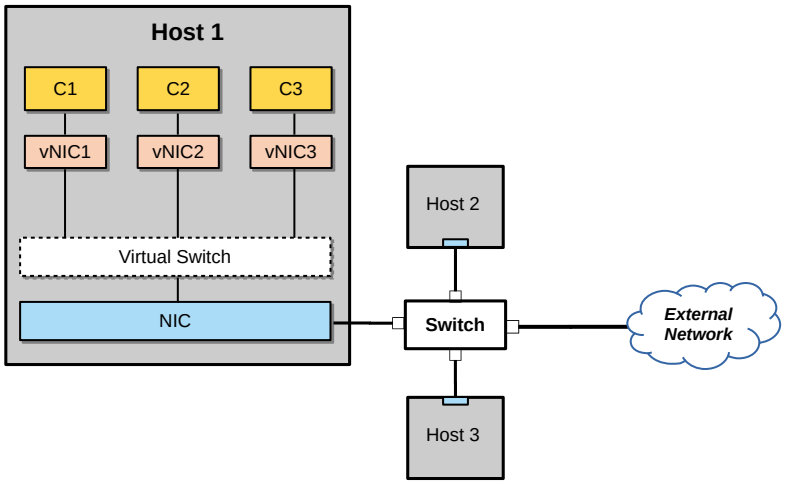

Figure 7.2 Virtual network of reference for our framework

in the form of one or many files called a trace. Depending on the events size and generation frequency, tracing can easily result in a huge trace. Nevertheless, the literature reports many proposals that leverage various abstraction techniques to make the trace as compact as possible [121].

On the other hand, a tracer is a tool that enables the operating system and userspace applications to implement configurable and low-overhead logging [158]. There is a wide variety of tracers that differ in their licensing options (open source versus commercial), tracing level (kernel space versus userspace level), and supported features (e.g., dynamic tracing, events aggregation, and filtering, etc.). For instance, Ftrace [82], LTTng [81], and SystemTap [83] are a few popular tracers in this domain.

A trace viewer is a tool that converts the trace content into human-readable information. A trace analyzer is another tool that integrates a set of sophisticated functionalities into the trace viewer, to enable the user to deduct useful analyses from raw data events. Trace Compass [159] is a powerful open-source trace analyzer that is endowed with an interactive graphical interface. It is shipped with many interesting pre-built views and analyses dedicated to LTTng traces. Babeltrace [86], on the other hand, is a trace viewer which can be used to navigate into a CTF trace.

Our framework uses LTTng as a tracer, due to its low impact on the traced system performance, and its ability to trace at kernel and userspace levels. The collection of performance data here is limited to the kernel space of the host machine only. Furthermore, no internal access to containers is required, since our approach is agent-free. We added many tracepoints to the Linux kernel, for the sake of our analysis, as shown in tables 7.1 and 7.2.

Table 7.1 Subset of the tracepoints leveraged by the first method

Tracepoint	Description
br_fdb_update	Fired when a new entry is inserted into the FDB of a bridge through a self-learning process.
br_fdb_add	Fired when a new entry is manually added by a user into the FDB of a bridge.
br_fdb_external_learn_add	Fired when a new entry is added to the FDB by an external tool.
fdb_delete	Fired when an FDB entry is deleted (for example due to the expiration of its associated timer).
lttng_statedump_network_ifce	LTTng state dump that is fired at the beginning of the tracing session. It shows the virtual and physical interfaces along with their MAC and IPv4 addresses.
lttng_statedump_network_bridge	LTTng state dump that lists which network card interface is connected to which Linux bridge.

7.4.2 Data Analysis

This section presents the analyses that we implemented to determine the topology of a VN from generated traces. Before starting any analysis, it is crucial to synchronize the traces collected from different hosts by establishing order on their events. In a distributed system like ours, it is impossible to achieve this task by relying on a global physical time. Moreover, the clocks in host machines are not necessarily synchronized and are potentially susceptible to drift. However, it is sufficient for our analysis to ensure a causal order between events. For instance, this partial order can indicate that the events related to the transmission of a packet in machine X happened before those related to its reception in machine Y. Hence, we leveraged the *convex hull* algorithm, presented in [160], to synchronize our traces.

To build complex link-layer topologies, where the nodes of the target VN are scattered over many host machines, we used a network virtualization technology called VXLAN (Virtual eXtensible LAN). This technology allows the creation of link-layer logical networks capable of spanning the boundaries of the physical machines. VXLAN works by setting up virtual tunnels where link-layer Ethernet datagrams are encapsulated into UDP datagrams. The endpoints of these tunnels, also known as VTEPs (VXLAN Tunnel EndPoints), can be either virtual or physical switch ports.

The analysis developed in our framework considers a VN topology as a graph. The graph

vertices represent the switches, their ports, and the host machines, which are the containers in our context. The graph edges represent the data exchange connections between the vertices. Our framework proposes two different methods for inferring the topology of a VN. We implemented both strategies in Trace Compass.

First Method

In this method, the topology discovery is accomplished through two steps. The first step consists in discovering the direct connections between containers and switches, while the second step discovers connections between switches. Figure 7.3 illustrates these two types of connections. A solid line corresponds to a container-to-switch connection, and a dashed one corresponds to a switch-to-switch connection. The events leveraged by this method are presented in Table 7.1.

The events *lttng_statedump_network_interface* and *lttng_statedump_network_bridge* are used to discover the direct connections between the NICs of containers and the switches. The first event provides detailed information about the physical/virtual NICs and switches in a host machine, such as the interface name, namespace ID, MAC address, and IP address. The name of the network interface (*name* field) and its namespace ID (*ns_id* field) are used as its unique identifier. The second event is used to indicate which NIC is directly connected to which switch. It contains the switch ID and the enslaved interface ID.

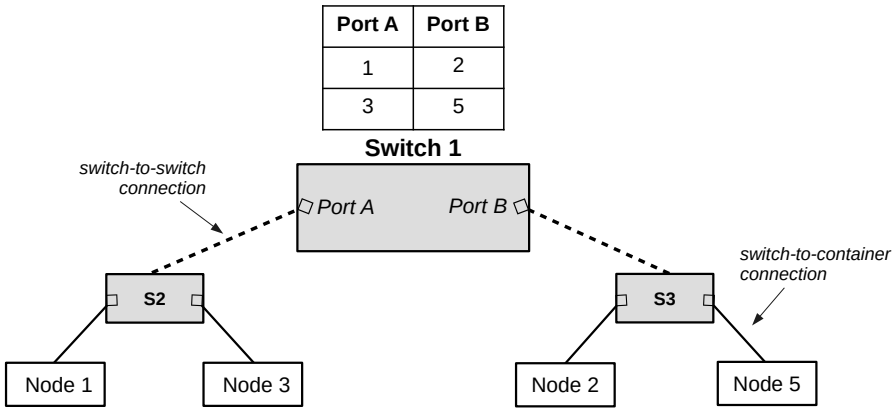

Figure 7.3 Switches use their FDB to forward frames between network segments

We leverage the data in the switches FDB tables to discover the switch-to-switch connections. The switches populate these tables during their learning process. An FDB table represents

the VN topology from the point of view of a switch. Each entry in the FDB tells the switch which port should be used to forward a packet to its destination. It also includes other important information, such as whether the destination is local or distant, and the VLAN ID associated with the port. Each entry is associated with a timer which causes its deletion when expired. Hence, any change to a given FDB table will trigger one of the events presented in Table 7.1 and, consequently, will be recorded by the LTTng tracer. For instance, the event *br_fdb_update* will be fired when a new entry is inserted into a switch FDB, following its self-learning process. Also, when an entry is deleted due to the expiration of its timer, the event *fdb_delete* is fired. We use those events as input data for the algorithm published in [154] to discover the links between the virtual switches.

The bridge implementation in recent Linux kernels is VLAN-aware. Thus, users can configure the ports of a Linux bridge according to the IEEE 802.1q standard, like any physical bridge. For instance, they can configure a bridge port to tag packets with a specific VLAN ID or to untag received packets. Based on the data available in the FDB table, the proposed method can detect which ports of a given switch have been assigned to a specific VLAN. Figure 7.4 presents a simple VN topology that has been discovered based on this method.

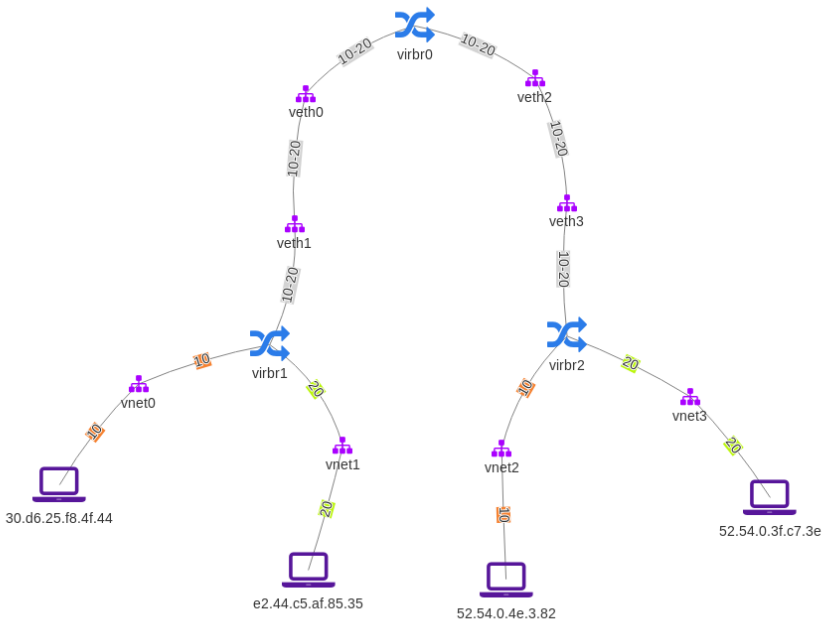

Figure 7.4 Graphical representation of a VN composed of four containers connected through three switches

Second Method

The second method infers the VN topology by combining all the individual paths taken by packets to reach their destinations. We can define the path as the set of nodes through which a packet has traversed in the VN. It mainly consists of three elements; the sender NIC, the switches that forwarded the packet and, finally, the recipient NIC. The tracepoints required for this method are described in Table 7.2. Let us consider two use cases to gain a better understanding of this method.

Table 7.2 A subset of the tracepoints leveraged by the second method

Tracepoint	**Description**
net_if_receive_skb	Fired when a new packet is received by a network interface and is about to go through the kernel TCP/IP stack.
net_dev_xmit	Fired when a packet is sent by the network interface.
skb_kfree	Fired when the skb data structure of a packet is freed by the kernel after a packet has been dropped.
skb_consume	Fired when a packet is consumed by a user space application, for instance.
*br_forward_skb_entry**	Fired when a packet is forwarded by a Linux bridge.

The VN is deployed within a single host This method considers a network as a directed graph, where nodes represent NICs, ports, or switches. It relies on firing events *net_if_receive_skb*, *net_dev_xmit*, and *br_forward_skb_entry* once a packet is received, sent by a NIC, or forwarded by a switch, respectively. Our algorithm uses the memory address of the associated kernel data structure (*skbaddr*) to uniquely identify a packet. This algorithm records the *skbaddr* of every traversed NIC in the path to track packets.

Hence, the algorithm stores the transmitting NICs, forwarding nodes, and the receiving NICs as a chain list for each sent packet. It also stores the size of sent packets to calculate the volume of the traffic exchanged between nodes. Our algorithm combines the chain lists (the traversed paths) to create a network topology graph. The chain lists are terminated and dumped on the graph as either *skb_kfree* or *skb_consume* is fired. This method presents the advantage of discovering the network topology and the traffic volume of the data exchange between network nodes. Figure 7.5 represents a VN topology built using this strategy.

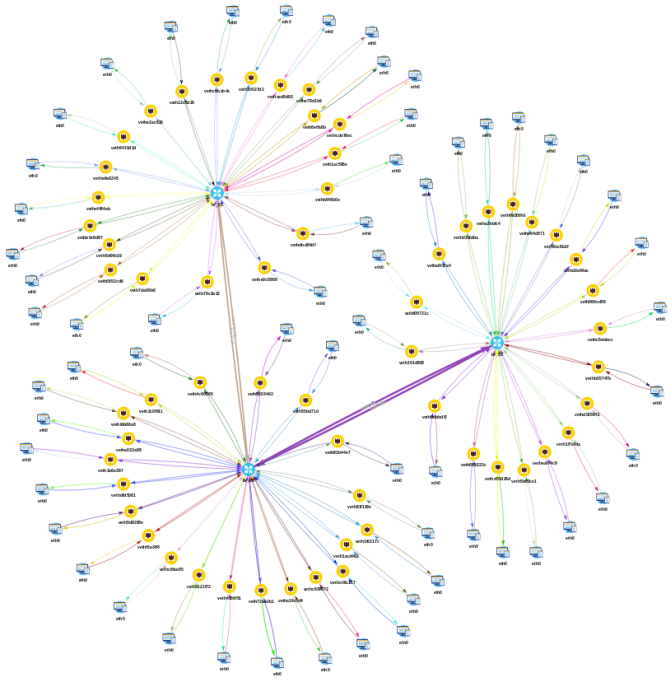

Figure 7.5 Graphical representation of a VN composed of dozens of containers connected through three switches

The VN is deployed over multiple hosts If the VN is distributed across many physical machines, one question might arise: how to track a packet when its receiver is not located within the same host machine as its sender? To answer this question, we need to uniquely identify the packets, that leave the host machine where the sender runs. As we explained earlier, if the sender and the receiver are on the same host, we leverage the *skbaddr* field to uniquely identify the packet. However, when the sender and the recipient are on different host machines, the value of this field will not obviously be the same in the sender and receiver host machines.

Hence, to trace the packets path in a physical LAN, one might think of leveraging the value of the host address, port and *Sequence Number* fields in the TCP header as a unique packet identifier. Unfortunately, this approach is too restrictive, as we cannot use it to track packets transported by protocols other than TCP (e.g., UDP). Therefore, we adopted another idea which consists of tagging the packets transmitted by physical NICs using the *identification* field of the IP header. The IP protocol uses this field to uniquely identify the fragments of a fragmented IP packet. Hence, based on the *kretprobe* mechanism, we wrote a kernel module that hooks into the function responsible for transmitting packets. This module tags the packet by setting a number in the *identification* field of its IP header, if the last sending

NIC is not a virtual NIC. As our target network does not contain routers, there is no risk that the *identification* field will be updated.

Algorithm 1 Algorithm for building the network topology graph from packets' paths

Input: *lttng_trace*, The synchronized LTTng trace
Output: G, The graph representing the VN topology

```
Initilize G, chain_dict
for each event in lttng_trace do
    if event.name = "net_if_receive_skb" or event.name = "net_dev_xmit" or
event.name = "br_forward_skb" then
        dev_name ← event["name"]
        size ← event["size"]
        id ← event["skbaddr"]
        new_id ← event["ipv4"].identification
        obj ← (dev_name, size)
        if new_id ≠ 0 then
            change_dict_key(chain_dict, id, new_id)
            add_node_chain(chain_dict, obj, new_id)
        else
            add_node_chain(chain_dict, obj, id)
        end if
    end if
    if event.name = "skb_kfree" or event.name = "skb_consume" then
        id ← event["skbaddr"]
        chain ← chain_dict[id]
        dump_chain_to_graph(G, chain)
        delete_chain(G, chain)
    end if
end for
for each id in chain_dict.keys() do
    chain ← chain_dict[id]
    dump_chain_to_graph(G, chain)
    delete_chain(G, chain)
end for
```

Example: Figure 7.6 shows the path taken by a packet sent by a container in Host1 to another container in Host2. Being transmitted by the container vNIC (*eth0*), the packet goes down through many network interfaces: the bridge port (*vethe127*), VXLAN interface (*vxlan10*), and Host1 NIC (*ens2*). Within Host1, the packet can be tracked using the memory address of its data structure *skbaddr* (*0xffff8eb1b6d14300*). When Host2 receives this packet, it goes up through many network interfaces: Host2 pNIC (*ens3*), VXLAN interface (*vxlan44*), the bridge port (*vethae036*), and, finally, the vNIC of the recipient container (*eth0*). The

received packet is saved in memory under a new *skbaddr* (*0xffffa02377a97d00*), but it can be matched with the packet sent by Host1 using the identification field of the IP header (*0x52*).

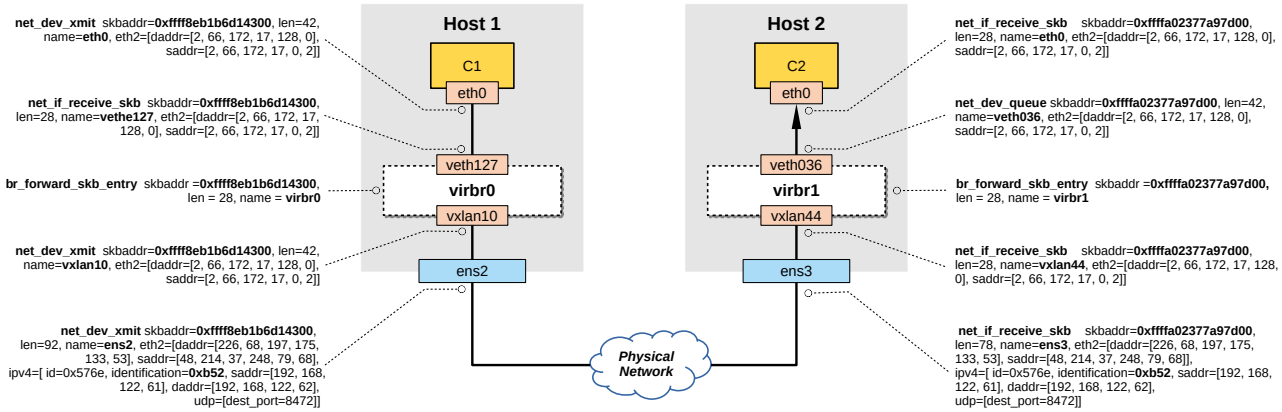

Figure 7.6 An excerpt of a synchronized trace indicating the path taken by a packet sent by a container in Host1 to another container in Host2

7.5 Cost Analysis

In this section, we assess the effectiveness of our approach, by analyzing the overhead induced by the analysis conducted to generate the required topologies. Often, the overhead on a given system is the main threat to its functionalities. Hence, we calculate the overhead of the proposed framework in terms of the disk space required for the data collection, in addition to the analysis time. The latter represents the time required for our analysis to read the traces and generate the associated topology of the VN.

Table 7.3 Hardware and software experimental configurations

Hardware Environment		**Software Environment**	
CPU	Intel Core i7-7500U CPU @2.70GHz	OS	Ubuntu 18.04
# Cores	4	Kernel	version 4.19.5
RAM	16 GB	LTTng	version 2.12.0

To compare the performance of our analysis methods, we run a set of experiments using the configuration setup shown in Table 4.3. In each experiment, we created a VN deployed in a LAN composed of three host machines. We varied the number of virtual switches to create

VNs with different sizes and topologies. To each virtual switch, we connected one container. In our experiments, we configured the containers to ping each other randomly, at the rate of one ping per second. Our goal is to generate a sufficiently large amount of traffic, required for our analysis. We ran and traced each experiment for a period of 300 seconds.

Table 7.4 presents the results of our experiments. It shows how the size of the traces varies according to the number of virtual switches in the VN. In addition, it shows the time required for recovering the link-layer topology from the traces. As a first observation, we can notice that the generated traces are very large. For instance, tracing the activity of a VN composed of 10k virtual switches for only 5 minutes results in traces larger than 8 GB in size (second method). Hence, the considerable size of trace files is a common limitation for all tracing techniques. Luckily, the literature reports many useful techniques that can be used to compress traces. For example, the authors in [121] propose a taxonomy of data abstraction techniques that use pattern matching to replace several low-level events with a more meaningful higher-level event. Another observation is about the analysis time, which closely correlates with the trace size. This makes sense since the larger the trace size, the more events it contains, the longer is the time required to process them.

Furthermore, we can observe that the size of the traces increases as a function of the number of virtual switches that compose the virtual network. Moreover, it is clear that the rate at which the trace size increases is higher for the second method, compared to the first one. We can explain this observation for the second method by the fact that the paths taken by packets are longer in large VNs than in small ones. Hence, since the events are fired when packets traverse the transient nodes, it is normal that the trace size increases once the VN becomes larger. As for the first method, the number of entries in the FDB increases with the number of virtual switches discovered in the network (see section 7.4.2 for more details). Therefore, the rate of issued events increases with the frequency of adding entries to the FDBs. Table 7.4 also shows that the second method requires to use larger traces than the first method. This was to be expected, since the second method relies on many more tracing events to discover the paths and, consequently, derive the VN topology.

In summary, although our approach for recovering the link-layer topologies of VNs from traces is effective, the disk space cost remains a limitation, especially for the second method. Fortunately, there are many solutions to reduce this cost. Hence, one interesting solution consists in using sampling to lower the frequency of events issued. For instance, it is possible to configure the tracing session in a way to trigger events only for the N^{th} packets whose senders and receivers are the same. Another solution is to leverage the abstraction techniques, we mentioned earlier, to compress the trace files when the data collection phase is completed.

Therefore, we believe that the overhead incurred by our approach is acceptably low.

Table 7.4 Overhead incurred by the collection and analysis of the tracing data

Topology Discovery Method	Number of Virtual Switches	Trace Size (MB)	Analysis Time (s)
Method #1	100	12	<1
	1000	151	15
	10000	1088	89
Method #2	100	138	14
	1000	1702	97
	10000	8604	904

7.6 Conclusion

The automatic discovery of the link-layer topology of a network is a crucial process for many protocols and applications. For instance, information about the link-layer topologies of VNs is mandatory for the VNE, which consists of finding the optimal embedding of VNs on a physical substrate network. Unfortunately, the problem of discovering the topologies of VNs did not attract much interest from the research communities.

This paper addresses this issue and proposes a framework capable of building the link-layer topology of a VN spanning over many physical hosts. Our framework implements two different methods to explore the topology. The first method leverages the data available in FDB tables, to discover the connections between switches. The second method tracks the paths taken by network flows, to find the interconnections between network nodes. The data required by these two methods is collected through kernel tracing. Our approach presents the advantage of being non-invasive, as it does not require any access to deployed containers. We developed a prototype framework to demonstrate the feasibility of our approach. This framework is successful in determining complex VNs topologies in LANs. Our future work will be devoted to adapt our approach to Wide Area Networks (WANs).

Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena and EfficiOS.

CHAPITRE 8 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous revenons sur les objectifs de recherche, que nous avons définis précédemment, et examinons dans quelle mesure les travaux présentés dans les chapitres 4-7 ont permis de les atteindre. Rappelons que l'objectif de cette recherche est de proposer des outils et des méthodes permettant l'analyse de performance de certains logiciels, qui sont utilisés dans la mise en œuvre des réseaux programmables. De cet objectif principal découlent les objectifs spécifiques liés à la collecte des données, l'analyse des traces, et la diminution des surcoûts générés.

Collecte des données

Pour élaborer nos analyses de performance, nous nous sommes basés sur l'instrumentation des solutions logicielles étudiées, en plus de l'exploitation des points de trace fournis par le noyau Linux. En effet, dans la plupart de nos travaux, nous avons eu recours à une instrumentation statique de ces solutions, dans le but d'extraire des données permettant de caractériser leurs performances. À titre d'illustration, dans le chapitre 5, nous avons effectué une instrumentation manuelle du code source de DPDK, la solution d'accélération de traitement des paquets réseau. De même, dans le chapitre 6, nous avons instrumenté le code source de KVMGT, la solution de virtualisation des processeurs graphiques. Parfois, nous avons opté pour l'ajout des points de trace dynamiques, au niveau du noyau Linux. Il est vrai que l'instrumentation dynamique génère un surcoût relativement élevé, mais elle évite à l'utilisateur la modification et la recompilation du noyau.

Quelquefois, nous avons eu besoin de mettre en correspondance des évènements de traçage pour élaborer certaines analyses. À titre d'exemple, dans le chapitre 4, il était nécessaire de correspondre l'évènement qui dénote l'envoi d'un *upcall*, par le module noyau d'Open vSwitch, avec celui qui dénote sa réception par son module de l'espace utilisateur. De même, dans le chapitre 7, il était nécessaire de faire correspondre l'évènement qui indique l'envoi d'un paquet IP donné, avec celui qui indique sa réception, lorsque les deux évènements ont été enregistrés sur des machines différentes. À ce titre, nous avons développé des modules noyau, qui s'appuient sur le mécanisme *Kretprobe*, pour s'attacher à des fonctions précises dans le noyau. Bien que cette technique génère un surcoût supplémentaire, elle évite à l'utilisateur de devoir modifier le code source du noyau.

Il est à noter que LTTng, notre outil de traçage principal, a été d'une grande utilité, surtout

quand les analyses nécessitaient la collecte de données de performance, à partir de plusieurs composants logiciels. À titre d'exemple, l'utilisation de ce traceur a facilité la surveillance des interactions du module noyau d'Open vSwitch avec les fils d'exécution au niveau de l'espace utilisateur (voir chapitre 4). Les analyses de performance multi-niveaux, qui ont été rendues possibles grâce à la diversité des sources de données, ont permis de découvrir des scénarios où le fonctionnement de ce logiciel n'est pas optimal.

Analyse des traces d'exécution

L'application des techniques d'abstraction variées sur les données de traçage nous a aidés à élaborer des analyses pertinentes, et à diagnostiquer des bogues de performance difficiles. À ce titre, les techniques d'abstraction nous ont été particulièrement utiles dans la modélisation des mécanismes et des composantes des solutions logicielles étudiées. À titre d'exemples, elles nous ont permis de modéliser des fils d'exécution, des pipelines, des files d'attente, ainsi que des ressources physiques et virtuelles variées (p. ex., processeurs graphiques, cartes réseau, etc.). En effet, notre approche pour identifier des anomalies dans le comportement du logiciel supervisé repose sur deux piliers. Premièrement, le calcul des états de ses composants, à partir d'une trace d'exécution, et la recherche, par la suite, d'une combinaison des états incohérents. Deuxièmement, le calcul des métriques de performance génériques, comme le débit de transmission de trafic par interface réseau, et spécifiques, comme le pourcentage de scrutation réussie par fil d'exécution (*Percentage of RX Spins*, voir section 5.5.2). À travers l'inspection de ces métriques, l'utilisateur peut identifier une dégradation de performance ou la présence d'un goulot d'étranglement.

Il convient de noter que l'algorithme de synchronisation de traces a été d'une grande utilité, en permettant d'appliquer des analyses sur une trace unifiée, construite à partir de plusieurs traces enregistrées sur des machines différentes. De même, l'utilisation des structures de données basées sur la *State History Tree*, a rendu possible le développement des analyses capables de traiter efficacement une très grande quantité de données de traçage. Au niveau de la visualisation des traces, nous avons utilisé des abstractions variées pour illustrer les résultats de nos analyses. À ce titre, les nombreuses facilités offertes par le cadriciel Trace Compass nous ont été utiles dans la mise en œuvre de ces abstractions. De même, nous nous sommes servis des représentations graphiques diverses (p. ex., courbe, tableau, histogramme, etc.) dans l'affichage des métriques de performance calculées et des états des composants du logiciel tracé. Le résultat était des interfaces graphiques interactives permettant de naviguer facilement dans la trace, de suivre les indicateurs de performance de l'application surveillée, et d'identifier ses bogues de performance.

Évaluation des surcoûts

Dans le but d'estimer l'efficacité de nos analyses de performance, nous avons évalué séparément les surcoûts relatifs aux phases de collecte et d'analyse de données. En effet, nous avons effectué une série de tests pour chaque analyse fournie, afin de mesurer le surcoût induit par le traçage. Dans la plupart de ces tests, nous avons utilisé un outil d'étalonnage pour créer des charges de travail différentes, et nous avons mesuré la performance de la solution étudiée, lorsque les points de trace sont activés et désactivés. Nos mesures montrent que le surcoût généré par le traçage est très faible, ce qui prouve l'efficacité de cette méthode, et l'adéquation de notre approche pour l'analyse des applications qui s'exécutent dans un environnement de production.

Nous avons également évalué le surcoût de traçage pour ce qui est de l'espace disque requis pour la sauvegarde des traces. En effet, étant donné que le traçage s'intéresse à l'enregistrement des données de bas niveau, la fréquence de génération des évènements est, dans la plupart des cas, assez élevée. En conséquence, les sessions de traçage donnent souvent lieu à des traces volumineuses, dont l'analyse requiert beaucoup des ressources de calcul et de mémoire. Afin de résoudre ce problème, nous avons proposé dans le chapitre 4, une méthode permettant de réduire la taille des traces générées.

La méthode proposée consiste à définir, au niveau du traceur, deux sessions de traçage : une session *légère* qui active un ensemble restreint des points de trace, et une autre session *exhaustive* qui active un ensemble plus grand des points de trace. La première session de traçage fournit les évènements nécessaires à la détection des problèmes de performance potentiels. Ces évènements sont en effet analysés à la volée par un module rattaché au traceur, et qui implémente des heuristiques variées. Quant à la deuxième session, elle fournit des évènements servant de matière pour l'investigation des causes des problèmes détectés, et qui sont destinés pour des analyses a posteriori. En effet, le traceur est configuré pour sauvegarder sur disque uniquement les évènements capturés par la deuxième session, lorsque le module d'analyse, associé à la première session, le notifie de l'occurrence d'un problème de performance.

En somme, nous estimons que les objectifs de la recherche définis précédemment ont été largement atteints, et que notre approche permet d'analyser efficacement les performances des infrastructures SDN et NFV.

CHAPITRE 9 CONCLUSION

Aujourd'hui, l'industrie de la réseautique est très préoccupée par les nombreux enjeux technologiques, imposés par l'explosion du trafic réseau et les besoins en flexibilité des nouvelles applications. Au cœur de ces enjeux, on trouve surtout la complexité de gestion des réseaux traditionnels et la difficulté de déploiement des services réseau. Pour relever ces défis, les opérateurs et les fournisseurs de services sont grandement séduits par les réseaux programmables, rendus possibles grâce à des architectures innovantes, telles que la NFV et le SDN. Ces architectures promettent d'accroître la flexibilité et l'agilité des réseaux, à travers l'exécution, par des logiciels, des fonctions auparavant remplies par du matériel dédié. Néanmoins, cette "softwarisation" des réseaux favorise davantage l'occurrence des bogues de performance, et complique le diagnostic de leurs causes profondes.

Dans cette thèse, nous proposons une approche permettant d'analyser les performances des solutions logicielles (p. ex., Open vSwitch, DPDK), qui sont impliquées dans la mise en œuvre des architectures SDN et NFV. Cette approche se base sur un traçage aux niveaux noyau et espace utilisateur pour collecter les données de performance requises par les analyses. Étant donné que le traçage induit parfois un surcoût considérable en termes d'espace de stockage, nous avons proposé une technique permettant de réduire la taille des traces générées. En somme, nos tests montrent que la collecte des données génère un très faible surcoût, ce qui prouve l'efficacité de notre approche.

De plus, nous avons développé plusieurs analyses capables d'identifier et de diagnostiquer des bogues de performance dans les logiciels observés. Ces analyses reposent sur diverses techniques d'abstraction, afin de dériver les états des composants internes du logiciel surveillé, à partir des traces collectées, et calculer des métriques de performance adaptées. Pour éviter la relecture des évènements de traçage, nous sauvegardons l'historique de ces états dans une base de données locale, lors de la première lecture de la trace. L'objectif de cette étape est d'accélérer l'exécution de nos analyses, et rendre possible le traitement des traces volumineuses. Nous avons également proposé une analyse permettant de reconstituer la topologie d'un réseau virtuel, à partir d'une trace d'exécution. Le but était d'analyser les chaînes de service dans l'infrastructure NFV, et identifier les liens qui existent entre les VNF.

Il est important de souligner que tous les outils que nous avons utilisés dans le cadre de cette thèse sont libres et à source ouverte. En particulier, nous nous sommes servis de LTTng dans la génération des traces d'exécution, et de Trace Compass dans l'implémentation de nos analyses. En effet, intégrer toutes les analyses dans un même cadriciel d'analyse permet

d'avoir une vue complète et détaillée sur les différents aspects de performance du système observé. À ce titre, les cas d'utilisation que nous avons présentés démontrent l'efficacité de notre approche, et la capacité de nos analyses à diagnostiquer des bogues de performance complexes.

9.1 Limitations de la solution proposée

Comme nous l'avons mentionné précédemment, nous nous sommes basés essentiellement sur l'instrumentation statique des applications cibles, afin d'extraire leurs données de performance. Bien que cette méthode génère un surcoût minimal, et offre une grande flexibilité quant au placement des points de trace, elle présente l'inconvénient de nécessiter un accès au code source de l'application, en plus d'une étape de recompilation. En conséquence, l'utilisation de la version instrumentée des bibliothèques et des outils étudiés est nécessaire pour la mise en œuvre des analyses développées dans le cadre de cette thèse. De ce fait, la collecte des données qui sont requises pour les analyses pourrait, dans des contextes très spécifiques, être impossible, tel que lorsque l'application à surveiller est déjà en cours d'utilisation. Dans certains cas, le recours à l'instrumentation dynamique peut aider à surmonter cette limitation.

Il convient de noter aussi qu'un bon nombre des analyses fournies dans cette thèse sont destinées à des utilisateurs expérimentés, qui ont une bonne compréhension du fonctionnement du système surveillé. Il est vrai que ces analyses permettent d'avoir une bonne visibilité du fonctionnement du logiciel surveillé, ainsi que des états et performances de ses mécanismes internes. Toutefois, la découverte des bogues de performance ou des anomalies dans le comportement du logiciel repose essentiellement sur l'expertise de l'utilisateur. En effet, l'utilisateur doit avoir une connaissance approfondie de la conception et du fonctionnement du logiciel surveillé pour pouvoir diagnostiquer ses bogues. Cette limitation peut être surmontée par le développement d'un système intelligent, qui est capable d'automatiser le processus de découverte des bogues de performance.

La nécessité d'installer un outil de traçage sur la machine, au niveau de laquelle l'application à surveiller s'exécute, peut aussi être considérée comme une limitation. En effet, nous avons vu comment l'implémentation d'un traceur natif au sein de DPDK permet aux applications accélérées de produire des traces au format CTF, sans toutefois avoir à recourir à un outil de traçage externe. Il est entendu que l'utilisation d'un traceur indépendant permet de collecter des données de performance utiles, à partir du noyau et des autres applications en exécution. Toutefois, il est important de souligner que les surcoûts induits par un traceur indépendant sont généralement supérieurs à ceux d'un traceur natif (voir section 5.7). Évidemment, cette

limitation peut être surmontée par l'ajout des facilités donnant aux applications réseau la possibilité de produire, d'une façon autonome, des données de traçage.

Une autre limitation tient au fait que la validation des analyses proposées et le calcul du surcoût de traçage ont été effectués uniquement dans un environnement de test. Toutefois, pour attester de l'efficacité de notre approche, nous estimons qu'il est important de mettre à l'épreuve les capacités de nos outils à diagnostiquer les bogues de performance qui se produisent dans un environnement de production. En effet, les environnements de production sont différents de ceux de test sur plusieurs aspects, tels que le nombre des éléments à surveiller (p. ex., les commutateurs virtuels), le volume du trafic qui transite sur le réseau, la tolérance à l'impact de la collecte des données sur les activités de production, etc. Par conséquent, l'échelle et la complexité de l'environnement de production, en plus de l'impossibilité de contrôler toutes les conditions de l'analyse à effectuer, ne peuvent qu'augmenter davantage la difficulté du diagnostic des problèmes de performance.

9.2 Améliorations futures

Dans cette section, nous proposons quelques pistes à suivre pour améliorer les travaux présentés dans cette thèse.

En effet, les analyses que nous avons présentées fournissent des indicateurs de performance variés, ainsi que des informations détaillées sur le fonctionnement des mécanismes internes du logiciel observé. Néanmoins, il incombe à l'utilisateur d'interpréter ces informations et découvrir les anomalies ou bogues de performance. Parfois, cette tâche peut être ardue, surtout lorsque la période de traçage est longue, ou que l'utilisateur ne dispose pas des connaissances nécessaires à l'analyse des informations fournies. De ce fait, une piste intéressante pour améliorer cette recherche consiste à proposer des algorithmes capables d'automatiser le processus de détection des bogues de performance. Ces algorithmes peuvent être développés sur la base des règles de détection d'anomalies définies par des experts (p. ex., combinaisons des états incohérents, durées de séjour maximales dans les files d'attente, limites sur les latences de traitement des paquets, etc.), ou des techniques d'apprentissage machine.

D'un autre côté, l'approche, que nous avons adoptée pour découvrir et diagnostiquer des bogues de performance, consiste à tracer l'exécution des applications à observer, puis analyser, d'une façon hors-ligne, les traces obtenues. Malheureusement, cette approche ne permet pas de détecter les problèmes de performance en un temps réel, et n'assure pas, en conséquence, la réactivité requise pour diagnostiquer et corriger ces problèmes dans les plus brefs délais. De plus, les analyses de performance hors-ligne sont connues pour introduire une surcharge

importante au niveau du stockage de données. Bref, une amélioration possible de nos travaux consiste à modifier les algorithmes proposés de telle sorte qu'ils puissent analyser en ligne les données de performance exportées par le logiciel surveillé. À ce titre, LTTng, notre outil de traçage principal, présente deux modes de fonctionnement intéressants, qui peuvent être exploités dans ce contexte. Le premier offre la possibilité de sauvegarder en mémoire seulement les données de traçage, et le deuxième permet de diffuser ces données en continu, à travers un réseau.

RÉFÉRENCES

[1] "Intel dpdk," En ligne : https://www.dpdk.org/, août 2021.

[2] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres et N. McKeown, "Where is the debugger for my software-defined network?" ser. HotSDN '12, Proceedings of the First Workshop on Hot Topics in Software Defined Networks. ACM, 2012, p. 55–60.

[3] D. Kreutz, F. M. V. Ramos, P. E. V. andChristian Esteve Rothenberg, S. Azodolmolky et S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, n°. 1, p. 14–76, janv. 2015.

[4] N. M. K. Chowdhury et R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, n°. 5, p. 862–876, 2010.

[5] "Open vswitch," En ligne : https://www.openvswitch.org/, juill. 2021.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker et J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, n°. 2, p. 69–74, mars 2008.

[7] Open Networking Foundation, "Openflow switch specification, version 1.4.0," En ligne : https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf, October 2013.

[8] "Floodlight sdn openflow controller," En ligne : https://github.com/floodlight/floodlight, sept. 2021.

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown et S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM computer communication review*, vol. 38, n°. 3, p. 105–110, 2008.

[10] D. Erickson, "The beacon openflow controller," dans *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 13–18.

[11] O. Blial, M. Ben Mamoun et R. Benaini, "An overview on sdn architectures with multiple controllers," *Journal of Computer Networks and Communications*, vol. 2016, 2016.

[12] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu et B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," dans *2014 IEEE 22nd International Conference on Network Protocols*. IEEE, 2014, p. 569–576.

[13] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow et G. Parulkar, "Onos: Towards an open, distributed sdn os," dans *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–6.

[14] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," dans *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–132.

[15] B. Pfaff et B. Davie, "The open vswitch database management protocol," Internet Requests for Comments, RFC 7047, Dec 2013. [En ligne]. Disponible: http://www.ietf.org/rfc/rfc7047.txt

[16] L. Zhu, M. M. Karim, K. Sharif, C. Xu, F. Li, X. Du et M. Guizani, "Sdn controllers: A comprehensive analysis and performance evaluation study," *ACM Comput. Surv.*, vol. 53, nº. 6, déc. 2020.

[17] R. Sherwood et K.-K. Yap, "Cbench: an openflow controller benchmarker," En ligne : https://github.com/andi-bigswitch/oflops/tree/master/cbench, sept. 2021.

[18] M. Jarschel, F. Lehrieder, Z. Magyari et R. Pries, "A flexible openflow-controller benchmark," dans *2012 European Workshop on Software Defined Networking*, 2012, p. 48–53.

[19] M. Jarschel, C. Metter, T. Zinner, S. Gebert et P. Tran-Gia, "Ofcprobe: A platform-independent tool for openflow controller analysis," dans *2014 IEEE Fifth International Conference on Communications and Electronics (ICCE)*, 2014, p. 182–187.

[20] Z. Shang, H. Wu et K. Wolter, "An openflow controller performance evaluation tool," dans *Computer Performance Engineering*. Springer International Publishing, 2018, p. 235–249.

[21] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood et A. W. Moore, "Oflops: An open framework for openflow switch evaluation," dans *Passive and Active Measurement*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 85–95.

[22] C. Rotsos, G. Antichi, M. Bruyere, P. Owezarski et A. W. Moore, "Oflops-turbo: Testing the next-generation openflow switch," dans *2015 IEEE International Conference on Communications (ICC)*, 2015, p. 5571–5576.

[23] Y. Jiang, X. Yang, H. Chen, W. Quan, Z. Sun et G. Lv, "Ortf: Open-source reconfigurable testing framework for sdn switches," dans *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International*

Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, p. 1188–1196.

[24] H. Kang, S. Lee, C. Lee, C. Yoon et S. Shin, "Spirit: A framework for profiling sdn," ser. ICNP' 15, IEEE 23rd International Conference on Network Protocols. IEEE, 2015, p. 417–424.

[25] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières et N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," ser. NSDI '14, 11th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, 2014, p. 71–85.

[26] W. Stallings, "Snmp and snmpv2: the infrastructure for network management," *IEEE Communications Magazine*, vol. 36, n°. 3, p. 37–43, 1998.

[27] B. Claise, G. Sadasivan et V. Valluri, "Cisco systems netflow services export version 9," En ligne : https://www.ietf.org/rfc/rfc3954.txt, oct. 2004.

[28] P. Phaal, S. Panchen et N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," En ligne : https://tools.ietf.org/html/rfc3176, sept. 2001.

[29] M. Afaq, S. Rehman et W.-C. Song, "Large flows detection, marking, and mitigation based on sflow standard in sdn," *Journal of Korea Multimedia Society*, vol. 18, n°. 2, p. 189–198, 2015.

[30] T. Lin et W. Chao, "The research on network performance management system based on sdn technology," ser. ICSPCC 2015, IEEE International Conference on Signal Processing, Communications and Computing. IEEE, 2015.

[31] N. L. M. van Adrichem, C. Doerr et F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," ser. NOMS '14, IEEE Network Operations and Management Symposium. IEEE, 2014.

[32] J. Suh, T. T. Kwon, C. Dixon, W. Felter et J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," ser. ICDCS '14, 34th International Conference on Distributed Computing Systems. IEEE, 2014, p. 228–237.

[33] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek et G. Antichi, "Enabling event-triggered data plane monitoring," ser. SOSR '20, Proceedings of the Symposium on SDN Research. San Jose, CA, USA: ACM, 2020, p. 14–26.

[34] X. T. Phan et K. Fukuda, "Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks," *Journal of Information Processing*, vol. 25, p. 182–190, 2017.

[35] X. T. Phan, I. D. Martinez-Casanueva et K. Fukuda, “Adaptive and distributed monitoring mechanism in software-defined networks,” ser. CNSM 2017, 13th International Conference on Network and Service Management. IEEE, 2017.

[36] P. Shantharama, A. S. Thyagaturu et M. Reisslein, “Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies,” *IEEE Access*, vol. 8, p. 132 021–132 085, 2020.

[37] T. Zhang, H. Qiu, L. Linguaglossa, W. Cerroni et P. Giaccone, “Nfv platforms: Taxonomy, design choices and future challenges,” *IEEE Transactions on Network and Service Management*, vol. 18, nº. 1, p. 30–48, 2021.

[38] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel et G. Bianchi, “Survey of performance acceleration techniques for network function virtualization,” *Proceedings of the IEEE*, vol. 107, nº. 4, p. 746–764, 2019.

[39] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern et D. Miller, “The express data path: Fast programmable packet processing in the operating system kernel,” ser. CoNEXT ’18, Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66.

[40] L. Rizzo, “netmap: a novel framework for fast packet i/o,” ser. 21st USENIX Security Symposium (USENIX Security 12), Bellevue, WA, USA, 2012, p. 101–112.

[41] C. Zeng, F. Liu, S. Chen, W. Jiang et M. Li, “Demystifying the performance interference of co-located virtual network functions,” dans *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, p. 765–773.

[42] P. Naik, D. K. Shaw et M. Vutukuru, “Nfvperf: Online performance monitoring and bottleneck detection for nfv,” ser. IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). IEEE, 2016, p. 154–160.

[43] W. Wu, K. He et A. Akella, “Perfsight: Performance diagnosis for software dataplanes,” ser. Proceedings of the Internet Measurement Conference. ACM, 2015, p. 409–421.

[44] F. Moradi, C. Flinta, A. Johnsson et C. Meirosu, “Conmon: An automated container based network performance monitoring system,” dans *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017, p. 54–62.

[45] M. Gokan Khan, S. Bastani, J. Taheri, A. Kassler et S. Deng, “Nfv-inspector: A systematic approach to profile and analyze virtual network functions,” dans *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, 2018, p. 1–7.

[46] J. Nam, J. Seo et S. Shin, “Probius: Automated approach for vnf and service chain analysis in software-defined nfv,” dans *Proceedings of the Symposium on SDN*

Research, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018. [En ligne]. Disponible: https://doi.org/10.1145/3185467.3185495

[47] G. Piao, P. K. Nicholson et D. Lugones, "Env2vec: Accelerating vnf testing with deep learning," dans *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[48] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang et K. Park, "Apunet: Revitalizing GPU as packet processing accelerator," dans *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, mars 2017, p. 83–96.

[49] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang et V. Volkov, "Parallel computing experiences with cuda," *IEEE Micro*, vol. 28, nº. 4, p. 13–27, 2008.

[50] J. E. Stone, D. Gohara et G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, nº. 3, p. 66, 2010.

[51] J. Tseng, R. Wang, J. Tsai, S. Edupuganti, A. W. Min, S. Woo, S. Junkins et T.-Y. C. Tai, "Exploiting integrated gpus for network packet processing workloads," dans *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, p. 161–165.

[52] E. Papadogiannaki, L. Koromilas, G. Vasiliadis et S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," *IEEE/ACM Transactions on Networking*, vol. 25, nº. 3, p. 1593–1606, 2017.

[53] C. Jung, S. Kim, I. Yeom, H. Woo et Y. Kim, "Gpu-ether: Gpu-native packet i/o for gpu applications on commodity ethernet," dans *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, p. 1–10.

[54] A. Kalia, D. Zhou, M. Kaminsky et D. G. Andersen, "Raising the bar for using gpus in software packet processing," dans *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, mai 2015, p. 409–423.

[55] D. Terpstra, H. Jagode, H. You et J. Dongarra, "Collecting performance data with papi-c," dans *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz et W. E. Nagel, édit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 157–173.

[56] "ROCm Tracer," En ligne : https://github.com/ROCm-Developer-Tools/roctracer, 2021.

[57] "Cuda profiling tools interface," En ligne : https://developer.nvidia.com/cuda-profiling-tools-interface, oct 2021.

[58] C. January, J. Byrd, X. Oró et M. O'Connor, "Allinea map: Adding energy and openmp profiling without increasing overhead," dans *Tools for High Performance Computing 2014*, C. Niethammer, J. Gracia, A. Knüpfer, M. M. Resch et W. E. Nagel, édit. Cham: Springer International Publishing, 2015, p. 25–35.

[59] "TAU performance system," En ligne : http://www.paratools.com/tau, 2019.

[60] K. Zhou, M. W. Krentel et J. Mellor-Crummey, "Tools for top-down performance analysis of gpu-accelerated applications," dans *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[61] "Nvidia nsight systems," En ligne : https://developer.nvidia.com/nsight-systems, 2021.

[62] "Intel vtune amplifier," En ligne : https://software.intel.com/en-us/intel-vtune-amplifier-xe, 2019.

[63] "CodeXL," En ligne : https://gpuopen.com/compute-product/codexl/, 2019.

[64] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey et N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, nº. 6, p. 685–701, 2010.

[65] K. Zhou, L. Adhianto, J. Anderson, A. Cherian, D. Grubisic, M. Krentel, Y. Liu, X. Meng et J. Mellor-Crummey, "Measurement and analysis of gpu-accelerated applications with hpctoolkit," *Parallel Computing*, vol. 108, p. 102837, 2021.

[66] P. Margheritta et M. R. Dagenais, "Lttng-hsa: Bringing lttng tracing to hsa-based gpu runtimes," *Concurrency and Computation: Practice and Experience*, vol. 0, nº. 0, p. e5231, avr. 2019.

[67] D. Couturier et M. R. Dagenais, "Lttng clust: A system-wide unified cpu and gpu tracing tool for opencl applications," *Advances in Software Engineering*, vol. 2015, p. 2:2–2:2, janv. 2015.

[68] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar et P. Ranganathan, "Gvim: Gpu-accelerated virtual machines," dans *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, ser. HPCVirt '09. New York, NY, USA: ACM, 2009, p. 17–24.

[69] J. Duato, A. J. Peña, F. Silla, R. Mayo et E. S. Quintana-Orti, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," dans *International*

Conference on High Performance Computing Simulation, vol. International Conference on High Performance Computing Simulation. IEEE, juin 2010, p. 224–231.

[70] V. Gupta, K. Schwan, N. Tolia, V. Talwar et P. Ranganathan, "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," dans *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, p. 3–3.

[71] L. Shi, H. Chen, J. Sun et K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, nº. 6, p. 804–816, juin 2012.

[72] C. Lee, S. W. Kim et C. Yoo, "Vadi: Gpu virtualization for an automotive platform," *IEEE Transactions on Industrial Informatics*, vol. 12, nº. 1, p. 277–290, févr. 2016.

[73] K. Tian, Y. Dong et D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through," dans *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, p. 121–132.

[74] Y. Dong, M. Xue, X. Zheng, J. Wang, Z. Qi et H. Guan, "Boosting gpu virtualization performance with hybrid shadow page tables," dans *USENIX Annual Technical Conference*. Santa Clara, CA: USENIX Association, 2015, p. 517–528.

[75] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He et H. Guan, "gscale: Scaling up GPU virtualization with dynamic sharing of graphics memory space," dans *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, p. 579–590.

[76] J. Song, "Kvmgt: a full gpu virtualization solution." dans *KVM Forum. The Linux Foundation.* The Linux Foundation, 2014.

[77] J. Ma, X. Zheng, Y. Dong, W. Li, Z. Qi, B. He et H. Guan, "gmig: Efficient gpu live migration optimized by software dirty page for full virtualization," dans *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '18. ACM, 2018, p. 31–44.

[78] "AMD MxGPU," En ligne : https://www.amd.com/en/graphics/workstation-virtualization-solutions, 2019.

[79] "Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation," En ligne : http://www.nvidia.com/content/grid/vdi-whitepaper.pdf, déc. 2021.

[80] "strace: linux syscall tracer," En ligne : https://strace.io/, sept. 2021.

[81] "Lttng," En ligne : https://lttng.org, juin 2021.

[82] S. Rostedt, “ftrace - function tracer,” En ligne : https://www.kernel.org/doc/Documentation/trace/ftrace.txt, juin 2021.

[83] F. C. Eigler et R. Hat, “Problem solving with systemtap,” ser. Proceedings of the Ottawa Linux Symposium, 2006, p. 261–268.

[84] “Userspace rcu,” En ligne : https://liburcu.org/, aug 2021.

[85] Y.-C. Liao et H. Langweg, “Cost-benefit analysis of kernel tracing systems for forensic readiness,” dans *Proceedings of the 2nd International Workshop on Security and Forensics in Communication Systems*, ser. SFCS ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 25–36. [En ligne]. Disponible: https://doi.org/10.1145/2598918.2598921

[86] EfficiOS, “Babeltrace.” En ligne : https://github.com/efficios/babeltrace, févr. 2019.

[87] “Trace Compass,” En ligne : http://tracecompass.org/, juill. 2021.

[88] A. Montplaisir, N. Ezzati-Jivan, F. Wininger et M. Dagenais, “Efficient model to query and visualize the system states extracted from trace data,” ser. International Conference on Runtime Verification, A. Legay et S. Bensalem, édit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 219–234.

[89] M. Jabbarifar, M. Dagenais et A. Shameli-Sendi, “Online incremental clock synchronization,” *Journal of Network and Systems Management*, vol. 23, n°. 4, p. 1034–1066, 2015.

[90] “Trex : Realistic traffic generator,” En ligne : https://trex-tgn.cisco.com/.

[91] A. Wundsam, D. Levin, S. Seetharaman et A. Feldmann, “Ofrewind: Enabling record and replay troubleshooting for networks,” ser. USENIX ATC ’11, USENIX Annual Technical Conference. Portland, Oregon, USA: USENIX Association, 2011, p. 327–340.

[92] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira et A. Valadarsky, “Vericon: towards verifying controller programs in software-defined networks,” ser. PLDI ’14, Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2014, p. 282–293.

[93] M. Canini, D. Venzano, P. Perešíni, D. Kostić et J. Rexford, “A nice way to test openflow applications,” ser. NSDI ’12, 9th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, 2012, p. 127–140.

[94] R. Skowyra, A. Lapets, A. Bestavros et A. Kfoury, “A verification platform for sdn-enabled applications,” ser. 2014 IEEE International Conference on Cloud Engineering. IEEE, 2014, p. 337–342.

[95] W. Queiroz, M. A. Capretz et M. Dantas, “An approach for sdn traffic monitoring based on big data techniques,” *Journal of Network and Computer Applications*, vol. 131, p. 28–39, 2019.

[96] Z. Zha, A. Wang, Y. Guo, D. Montgomery et S. Chen, “Instrumenting open vswitch with monitoring capabilities: designs and challenges,” ser. SOSR ’18, Proceedings of the Symposium on SDN Research. ACM, 2018.

[97] X. T. Phan et K. Fukuda, “Toward a flexible and scalable monitoring framework in software-defined networks,” ser. AINA 2017, 31st International Conference on Advanced Information Networking and Applications Workshops. IEEE, 2017, p. 403–408.

[98] A. Wang, Y. Guo, F. Hao, T. Lakshman et S. Chen, “Umon: Flexible and fine grained traffic monitoring in open vswitch,” ser. CoNEXT ’15, Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies. ACM, 2015.

[99] A. Saghir et T. Masood, “Performance evaluation of openstack networking technologies,” ser. ICEET ’19, International Conference on Engineering and Emerging Technologies. IEEE, 2019, p. 27–32.

[100] B. Claise, S. Bryant, S. Leinen, T. Dietz et B. H. Trammell, “Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information,” En ligne : https://tools.ietf.org/html/rfc5101, jan 2008.

[101] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, “The design and implementation of open vswitch,” ser. NSDI ’15, Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation. USENIX, 2015, p. 117–130.

[102] O. Belkadi, Y. Laaziz, A. Vulpe et S. Halunga, “An integration of opendaylight and opennebula for cloud management improvement using sdn,” ser. TELFOR ’19, 27th Telecommunications Forum. IEEE, 2019.

[103] J. Stringer, “Openvswitch without Open vSwitch: The API and its users.” NetDev 2.1 – The Technical Conference on Linux Networking, 2017.

[104] “Open networking foundation,” En ligne : https://www.opennetworking.org/, aug 2021.

[105] D. Toupin, “Using tracing to diagnose or monitor systems,” *IEEE Software*, vol. 28, n°. 1, p. 87–91, 2011.

[106] “The lttng documentation,” En ligne : https://lttng.org/docs/v2.10/, janv. 2019.

[107] “Open vswitch documentation, release 2.12.90,” En ligne : https://buildmedia.readthedocs.org/media/pdf/openvswitch/latest/openvswitch.pdf, janv. 2020.

[108] S. Shanmugalingam, A. Ksentini et P. Bertin, “Dpdk open vswitch performance validation with mirroring feature,” ser. ICT ’16, 23rd International Conference on Telecommunications. IEEE, 2016, p. 1–6.

[109] A. Bianco, R. Birke, D. Bolognesi, J. Finochietto, G. Galante, M. Mellia, M. Prashant et F. Neri, “Click vs. linux: two efficient open-source ip network stacks for software routers,” ser. HPSR. 2005, Workshop on High Performance Switching and Routing, 2005, p. 18–23.

[110] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye et M. Lipshteyn, “Rdma over commodity ethernet at scale,” ser. Proceedings of the 2016 ACM SIGCOMM Conference, 2016, p. 202–215.

[111] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart et G. Carle, “Throughput and latency of virtual switching with open vswitch: A quantitative analysis,” *Journal of Network and Systems Management*, vol. 26, n°. 2, p. 314–338, 2018.

[112] Intel, “Intel vtune profiler,” En ligne : https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html, juin 2021.

[113] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone et D. Rossi, “Flowatcher-dpdk: Lightweight line-rate flow-level monitoring in software,” *IEEE Transactions on Network and Service Management*, vol. 16, n°. 3, p. 1143–1156, 2019.

[114] M. Trevisan, A. Finamore, M. Mellia, M. Munafo et D. Rossi, “Traffic analysis with off-the-shelf hardware: Challenges and lessons learned,” *IEEE Communications Magazine*, vol. 55, n°. 3, p. 163–169, 2017.

[115] A. Belkhiri et M. Dagenais, “Diagnostic and troubleshooting of openflow-enabled switches using kernel and userspace traces,” *International Journal of Communication Systems*, vol. 34, n°. 14, p. e4920, 2021.

[116] K. Yasukata, M. Honda, D. Santry et L. Eggert, “Stackmap: Low-latency networking with the OS stack and dedicated nics,” ser. 2016 USENIX Annual Technical Conference (USENIX ATC 16). Denver, CO: USENIX Association, juin 2016, p. 43–56.

[117] R. Chen et G. Sun, “A survey of kernel-bypass techniques in network stack,” dans *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, ser. CSAI ’18. New York, NY, USA: Association for Computing Machinery, 12 2018, p. 474–477.

[118] F. Wolf, F. Freitag, B. Mohr, S. Moore et B. Wylie, “Large event traces in parallel performance analysis,” ser. ARCS’06, 19th International Conference on Architecture of Computing Systems. Gesellschaft für Informatik eV, 2006, p. 264–273.

[119] I. Vurdelja, I. Blažić, D. Drašković et B. Nikolić, "Detection of linux malware using system tracers-an overview of solutions," *International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*, 2020.

[120] E. Eskin, A. Arnold, M. Prerau, L. Portnoy et S. Stolfo, "A geometric framework for unsupervised anomaly detection," dans *Applications of data mining in computer security*. Springer, 2002, p. 77–101.

[121] N. Ezzati-Jivan et M. R. Dagenais, "Multi-scale navigation of large trace data: A survey," *Concurrency and Computation: Practice and Experience*, vol. 29, n°. 10, p. e4068, 2017.

[122] "Gpus on compute engine," En ligne : https://cloud.google.com/compute/docs/gpus/, oct. 2019.

[123] G. Aceto, A. Botta, W. Donato et A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, n°. 9, p. 2093–2115, 2013.

[124] C.-H. Hong, I. Spence et D. S. Nikolopoulos, "Gpu virtualization and scheduling methods: A comprehensive survey," *ACM Computing Surveys*, vol. 50, n°. 3, p. 35:1–35:37, juin 2017.

[125] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu et J. Wiegert, "Intel virtualization technology for directed i/o," *Intel Technology Journal*, vol. 10, p. 179–192, août 2006.

[126] L. van Doorn, "Hardware virtualization trends," dans *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, ser. VEE '06. New York, NY, USA: ACM, 2006, p. 45–45.

[127] J. P. Walters, A. J. Younge, D. I. Kang, K. T. Yao, M. Kang, S. P. Crago et G. C. Fox, "Gpu passthrough performance: A comparison of kvm, xen, vmware esxi, and lxc for cuda and opencl applications," dans *IEEE 7th International Conference on Cloud Computing*. IEEE, juin 2014, p. 636–643.

[128] J. Zhang, X. Lu et D. K. Panda, "Performance characterization of hypervisor-and container-based virtualization for hpc on sr-iov enabled infiniband clusters," dans *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, mai 2016, p. 1777–1784.

[129] J. Zhang, X. Lu et D. Panda, "High-performance virtual machine migration framework for mpi applications on sr-iov enabled infiniband clusters," dans *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, mai 2017, p. 143–152.

[130] Y. Suzuki, S. Kato, H. Yamada et K. Kono, "Gpuvm: Gpu virtualization at the hypervisor," *IEEE Transactions on Computers*, vol. 65, nº. 9, p. 2752–2766, 2016.

[131] "Nouveau: Accelerated open source driver for nvidia cards," En ligne : https://nouveau.freedesktop.org, oct. 2019.

[132] "Nvidia virtual gpu management and monitoring," En ligne : https://www.nvidia.com/en-us/data-center/virtualization/it-management/, sept. 2019.

[133] "Nvidia gpu monitoring & troubleshooting," En ligne : https://goliathtechnologies.com/software/goliath-performance-monitor/infrastructure/nvidia/, sept. 2019.

[134] "Introducing gpuopen," En ligne : https://gpuopen.com/, 2019.

[135] "Kernel virtual machine," En ligne : https://www.linux-kvm.org, 2019.

[136] "Qemu," En ligne : https://www.qemu.org/, 2019.

[137] S. Rostedt, "ftrace - function tracer," En ligne : https://www.kernel.org/doc/Documentation/trace/ftrace.txt, 2017.

[138] "Perf: Linux profiling with performance counters," En ligne : https://perf.wiki.kernel.org, sept. 2015.

[139] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee et K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," dans *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*. IEEE, oct. 2009, p. 44–54.

[140] M. Masdari, S. S. Nabavi et V. Ahmadi, "An overview of virtual machine placement schemes in cloud computing," *Journal of Network and Computer Applications*, vol. 66, p. 106 – 127, 2016.

[141] H. Nemati et M. Dagenais, "Virtual cpu state detection and execution flow analysis by host tracing," dans *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. IEEE, Oct 2016, p. 7–14.

[142] S. Wu, H. Yin, H. Cao, L. Yang et H. Zhu, "Node ranking strategy in virtual network embedding: An overview," *China Communications*, vol. 18, nº. 6, p. 114–136, 2021.

[143] L. Xingtao, G. Yantao, W. Wei, Z. Sanyou et L. Jiliang, "Network virtualization by using software-defined networking controller based docker," dans *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*, 2016, p. 1112–1115.

[144] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck et R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications surveys & tutorials*, vol. 18, n°. 1, p. 236–262, 2015.

[145] B. Donnet et T. Friedman, "Internet topology discovery: a survey," *IEEE Communications Surveys Tutorials*, vol. 9, n°. 4, p. 56–69, 2007.

[146] S. I. Saheb et A. Rasool Md, "Auto-discovery and monitoring of network resources: Snmp-based network mapping and fault management," dans *Smart Computing Techniques and Applications*, S. C. Satapathy, V. Bhateja, M. N. Favorskaya et T. Adilakshmi, édit. Singapore: Springer Singapore, 2021, p. 643–653.

[147] R. Wazirali, R. Ahmad et S. Alhiyari, "Sdn-openflow topology discovery: An overview of performance issues," *Applied Sciences*, vol. 11, n°. 15, 2021.

[148] S. Khan, A. Gani, A. W. A. Wahab, M. Guizani et M. K. Khan, "Topology discovery in software defined networks: Threats, taxonomy, and state-of-the-art," *IEEE Communications Surveys & Tutorials*, vol. 19, n°. 1, p. 303–324, 2016.

[149] J. Nogueira, M. Melo, J. Carapinha et S. Sargento, "A distributed approach for virtual network discovery," dans *IEEE Globecom Workshops*, 2010, p. 277–282.

[150] L. Lahlou, N. Kara et C. Edstrom, "Davinci: Online and dynamic adaptation of evolvable virtual network services over cloud infrastructures," *Future Generation Computer Systems*, 2021.

[151] M. A. Khan, A. Safi, I. M. Qureshi et I. U. Khan, "Flying ad-hoc networks (fanets): A review of communication architectures, and routing protocols," dans *International Conference on Latest trends in Electrical Engineering and Computing Technologies (INTELLECT)*, 2017, p. 1–9.

[152] "Fd.io: Vector packet processing," En ligne : https://fd.io/vppproject/vpptech/, aug 2021.

[153] Y. Breitbart, M. Garofalakis, C. Martin, R. Rastogi, S. Seshadri et A. Silberschatz, "Topology discovery in heterogeneous ip networks," dans *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, 2000, p. 265–274.

[154] U. Uzair, H. F. Ahmad, A. Ali et H. Suguri, "An efficient algorithm for ethernet topology discovery in large multi-subnet networks," dans *IEEE International Conference on System of Systems Engineering*, 2007, p. 1–7.

[155] K. Nowicki et A. Malinowski, "Topology discovery of hierarchical ethernet lans without snmp support," dans *41st Annual Conference of the IEEE Industrial Electronics Society*, 2015, p. 5439–5443.

[156] I. Sanchez-Navarro, A. S. Mamolar, Q. Wang et J. M. Alcaraz Calero, “5gtoponet: Real-time topology discovery and management on 5g multi-tenant networks,” *Future Generation Computer Systems*, vol. 114, p. 435–447, 2021.

[157] A. Kamal, “Ethernet topology discovery: A survey,” *CoRR*, vol. abs/0907.3095, 2009. [En ligne]. Disponible: http://arxiv.org/abs/0907.3095

[158] M. Gebai et M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead,” *ACM Computing Surveys (CSUR)*, vol. 51, nº. 2, p. 1–33, 2018.

[159] “Trace Compass,” En ligne : http://tracecompass.org/, juill. 2021.

[160] B. Poirier, R. Roy et M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events,” *ACM SIGOPS Operating Systems Review*, vol. 44, nº. 3, p. 75–87, 2010.