UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UN COMPILATEUR DE TRAITEMENT DU PAQUET POUR LES RÉSEAUX MULTI-TENANTS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

SALAHEDDINE HAMADI

JANVIER 2017

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.10-2015). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

TABLE DES MATIÈRES

LISTE DES TABLEAUX vii

LISTE DES FIGURES ix

LEXIQUE xi

RÉSUMÉ xiii

INTRODUCTION 1

CHAPITRE I
VIRTUALISATION DES RÉSEAUX ET COMPILATION DES RÈGLES 5

1.1 Virtualisation dans le SDN 6

1.1.1 Principe de SDN et objectif 7

1.1.2 OpenFlow 7

1.1.3 Compilation des règles 9

1.2 Principe de l'analyse formelle des concepts 12

1.2.1 Contexte formel 14

1.2.2 Relation d'incidence entre objets et attributs 14

1.2.3 Concepts 15

1.2.4 Treillis de concepts 16

1.3 Principe du *network calculus* 16

1.3.1 Granularité du modèle de flux 16

1.3.2 Charge et délai dans un systèmes 17

1.3.3 Les courbes d'arrivée et de services 18

1.3.4 Utilisation du network calculus dans l'analyse des performances 19

CHAPITRE II
COMPILATEUR MULTI-TENANTS 23

2.1 Logique de conception 23

2.1.1 Architecture des switches 23

2.1.2 Sémantique des règles 25

2.2 Approche de conception 26

2.2.1 Terminologie et notation 27

2.2.2 Validation et différentiation des règles 29

2.2.3 Fouille des règles 32

2.2.4 Modélisation de transmission de paquets 36

2.2.5 Validation de traitement du paquet 40

2.2.6 Partage de ressources 43

CHAPITRE III
IMPLÉMENTATION 47

3.1 Le processeur réseau nfp-3200 47

3.1.1 Architecture processeur réseau nfp-3200 48

3.1.2 Programmation du processeur nfp-3200 50

3.2 Le processeur multicoeur MPPA-256 51

3.2.1 Architecture du MPPA-256 51

3.2.2 Programmation du processeur MPPA-256 52

3.3 Implémentation du compilateur 54

CHAPITRE IV
ÉVALUATION 57

4.1 Paramètres de banc de test 57

4.2 Performance de compilation 60

4.2.1 Capacité du compilateur 61

4.2.2 Utilisation des ressources 63

4.3 Performance du plan de données 64

4.4 Équité d'allocation des ressources 66

CONCLUSION 69

APPENDICE A

L'ALGORITHME DE CONSTRUCTION DU TREILLIS DES CONCEPTS 73

A.1 Algorithme Charm 73

A.2 Trouver les générateurs fréquents 75

APPENDICE B
LISTE DES CONCEPTS 77

RÉFÉRENCES 79

LISTE DES TABLEAUX

Tableau Page

1.1 Liste des champs utilisés dans Openflow 1.1 9

1.2 Exemple d'un contexte formel $\mathcal{K}$ 14

2.1 Exemple des règles de tenant et des règles communes 32

2.2 Contexte global 33

2.3 Exemple des concepts 34

4.1 Configuration de NFP-3200 58

4.2 Configuration de MPPA-256 59

4.3 Motif de trafic 60

B.1 Liste des concepts 78

LISTE DES FIGURES

Figure		Page
1.1	Différence entre les réseaux classiques et les réseaux SDN	8
1.2	Diagramme de Hasse de treillis des concepts	17
1.3	La charge et le délai dans un système	18
1.4	Borne de la courbe d'arrivée	19
2.1	Exemple de plans de données de différentes architectures	25
2.2	Vue global d'emplacement du compilateur	27
2.3	Vue générale d'un compilateur multi-tenants	28
2.4	Diagramme de Hasse de treillis des concepts	35
2.5	Modélisation de transmission de paquet	41
2.6	Validation de traitement de paquet	43
2.7	Plan d'allocation des ressources	44
3.1	Le processeur réseau nfp-3200	48
3.2	Le processeur Kalray MPPA-256	52
3.3	Implémentation du compilateur	53
4.1	Pipeline d'acheminement de trafic	58
4.2	Banc de test	59
4.3	Temps de mise à jour	62
4.4	Surchage de la virtualisation	62
4.5	Effet de seuil sur la compilation	64
4.6	Réduction de nombre des entrées	65

4.7 Switch matérielle 66

4.8 Switch logicielle 67

4.9 Équité d'allocation des ressources 68

LEXIQUE

acl : Liste de contrôle d'accès.

bid : enchère

central processing unit (cpu) : unité centrale de traitement

cloud computing : nuage informatique

cluster : groupe ou groupement

data path : circuit de données

data plane : plan de données

double data rate (ddr) : bus fonctionnant en débit de données double

deadline : délai limite

dynamic random access memory (dram) : mémoire vive dynamique

extent : Extension d'un concept

field : champ

flow : Flot de paquets

flow table : table de flux, table à plusieurs entrées où chaque entrée est associée à un flux de paquet

global scratch (gs) : mémoire interne à haut débit

header : En-tête du paquet

input/output (i/o) : Entrée/Sortie

intent : Intension d'un concept

internet protocol (ip) : protocole internet

latency : latence ou délai de traitement

mapping : association

match : Correspondance

match field : Champ de correspondance

media switch fabric (msf) : interface de connexion au fabric de commutation

micro engine (me) : micro engin ou coeur de traitement de paquets

multi-cores : architecture du processeur à plusieurs cœurs de traitement

multi-threading : architecture de processeur à multiple fils d'exécution

network on chip (noc) : système de communication entre les cœurs d'un Processeur multi-coeurs

network processor unit (npu) : processeur réseau

software defined networks (sdn) : réseau définis de manière logicielle

static random access memory (sram) : mémoire vive statique

switch : commutateur réseau

traffic generator : générateur de trafic

RÉSUMÉ

Le *Software Defined Network* (SDN) est un nouveau paradigme de design et de gestion des réseaux. Le SDN se base sur la séparation entre le plan de données et le plan de contrôle. Le réseau est géré à travers un contrôleur centralisé. Ce dernier gère la politique d'acheminement et d'aiguillage des paquets sur tout le réseau par des opérations d'ajout et de suppression des règles. Le SDN simplifie la virtualisation réseau où plusieurs tenants peuvent coexister sur la même infrastructure physique, chaque tenant a son propre contrôleur qui dicte la politique de traitement des paquets dans son réseau. Cela pose des problèmes au niveau du partage des ressources et d'isolation du trafic. Pour assurer une isolation du trafic entre les différents tenants ainsi qu'une haute performance d'acheminement du paquet sur tout le réseau, on introduit un compilateur centrale qui fera la translation, la distribution et l'application des règles sur l'infrastructure physique.

Dans ce rapport, nous allons détailler la conception d'un compilateur de règles pour un réseau multi-tenants. Ce compilateur transforme les règles et adapte les structures de classification pour tenir compte des contraintes matérielles des équipements. Une implémentation du compilateur a été développée pour les *switches* physiques et logiciels. Les tests ont montré une réduction de 20% à 30% de nombre des entrées, une capacité de mise à jour qui s'élève à 300 règles/seconde et une minimisation du délai du traitement des paquets par 2-3 us et par 50-100 us respectivement pour les *switches* physiques et logiciels.

INTRODUCTION

La virtualisation est une technologie qui permet d'avoir plusieurs entités logiques sur la même infrastructure physique. La virtualisation simplifie la gestion, réduit l'utilisation d'énergie, optimise l'utilisation des ressources et finalement offre une élasticité à l'infrastructure. Dans le réseau, elle se déploie en utilisant des protocoles de virtualisation (ex. : VLAN et VPN) ou à travers le SDN (*Software Defined Networking*). Le SDN introduit de nouveaux paradigmes d'architecture de réseau. Le SDN sépare le plan de traitement des données (plan de données) du plan de contrôle. Cette séparation centralise le contrôle en une seule entité logique ce qui permet une simplification de l'architecture des équipements ainsi qu'une optimisation des opérations sur tout le réseau.

Dans un réseau SDN, la virtualisation du plan des données est séparée de la virtualisation du plan du contrôle. La virtualisation du plan de données se fait principalement par la création des chemins de traitements des paquets (*Path Processing*) dédiés pour chaque réseau logique. Cette séparation simplifie l'isolation du trafic, elle impacte la performance du réseau car elle nécessite un pré classificateur pour déterminer le chemin du traitement de chaque paquet. Alors, d'une part, cette méthode ne permet pas de créer des structures de classifications qui exploitent la sémantique des règles de tous les réseaux virtuels. Et d'autre part, les contrôleurs des réseaux logiques n'ont pas une vue globale sur toute l'infrastructure et cela les empêchent d'optimiser la distribution des règles ainsi que l'utilisation des ressources matérielles.

La virtualisation et la gestion d'une infrastructure physique sont des tâches déli-

cates. En effet, le réseau se compose de plusieurs types d'équipements hétérogènes. Cette hétérogénéité complique les opérations (ex. : insertion des règles et mise à jour des règles) sur un réseau. On peut classifier les *switches* réseaux en trois catégories :

- *Switch* logicielle (ex : Open vSwitch) qui s'exécute sur des processeurs génériques (ex. : x86, ARM). L'architecture d'une telle *switch* est flexible, elle peut s'adapter en fonction de son contexte d'utilisation.
- *Switch* d'accès (ex. : TOR *switch*), elle se base généralement sur des processeurs réseau (Network Processor-NP), elle est plus performante qu'une *switch* logicielle mais moins flexible.
- *Switch* fédérateur, elle se caractérise par une performance élevée et utilise des processeurs ASICs. Par contre, cette *switch* n'est pas flexible.

Dans ce travail, nous allons présenter un compilateur des règles qui gère l'infrastructure physique d'un réseau et qui assure une isolation du trafic entre les différents tenants (càd les réseaux virtuels). Ce compilateur crée un plan de données unifié pour tous les tenants tout en respectant les conditions d'isolation de trafic. Le compilateur intercepte les règles des contrôleurs puis il les modifie pour assurer l'isolation. Ensuite, il exploite la relation sémantique entre les règles pour créer des structures de recherches adaptées à l'architecture physique sous-jacente. Ces structures de recherche permettent de réduire le nombre des entrées requises au niveau d'une *switch* pour enregistrer un ensemble de règles de routage ou de commutation.

Ce mémoire se compose de quatre chapitres. Le premier chapitre présente les principes de SDN et la compilation des règles. Nous donnerons également un aperçu sur les solutions de la compilation des entrées.

Le deuxième chapitre portera sur la description du compilateur. Nous allons détailler le fonctionnement du compilateur, ses composants et aussi leurs interactions.

Dans le troisième chapitre, nous détaillerons l'implémentation de la solution. D'abord, nous allons présenter les architectures matérielles utilisées, ensuite, nous allons donner le flux global de la solution.

Avant de conclure, nous allons présenter dans le quatrième chapitre les résultats de l'évaluation relatifs aux performances du compilateur en terme de réduction du nombre des entrées et le temps de la compilation. Nous allons aussi mesurer les performances du plan de données en terme d'optimisation de latence sur les deux différentes architectures.

CHAPITRE I

VIRTUALISATION DES RÉSEAUX ET COMPILATION DES RÈGLES

La virtualisation est un concept très répondu dans le monde de l'informatique. Le sens de la virtualisation dépend principalement de son contexte d'utilisation. En général, la virtualisation se réfère à l'introduction d'une couche d'abstraction entre le matériel qui exécute les instructions ou les opérations et sa représentation logique (Menken, 2008). Dans le contexte des serveurs, la virtualisation donne la possibilité au serveur d'exécuter plusieurs systèmes d'exploitation en même temps, ces systèmes partagent les ressources du serveur (Daniels, 2009; Sahoo *et al.*, 2010). Dans un contexte de stockage, la virtualisation permet de consolider plusieurs disques physiques et les représenter comme un seul disque ou bien de diviser un seul disque dur et le représenter comme un ensemble des disques. Récemment, la virtualisation du réseau a attiré l'attention de la communauté des chercheurs et de l'industrie dans le but de transformer l'architecture des réseaux et de donner plus de flexibilité et d'agilité à leur développement.

Afin de mieux comprendre la virtualisation dans le contexte SDN, nous allons détailler les concepts de ce dernier. Ensuite, nous allons survoler les principes de l'analyse formelle des concepts et du network calculus.

1.1 Virtualisation dans le SDN

L'explosion des appareils mobiles, la virtualisation des serveurs et les services de *cloud computing* sont les principaux facteurs qui poussent les entreprises à réexaminer l'architecture des réseaux traditionnels. Les réseaux sont généralement conçus par un grand nombre des équipementsd de télécommunications, chaque équipement implémente un nombre de protocoles. Pour déployer un réseau, les ingénieurs et les administrateurs réseau doivent d'abord choisir des équipements qui sont interopérables et qui roulent la même version des différents protocoles, et enfin ils doivent configurer chaque équipement séparément. Un changement ou une mise à niveau demande des jours et parfois mêmes des semaines à déployer. Ce manque de flexibilité ne répond pas au besoin des centres de données où les applications peuvent changer d'emplacements plusieurs fois et/ou les applications ont besoin d'une quantité de ressources variables. Aussi, les équipements réseaux actuels sont conçus comme des boites fermées (*black box*) où le tout est fermé et couplé dans un seul équipement. L'utilisateur ne peut pas ajouter des services ou bien supprimer des services qui ne sont pas nécessaires pour libérer des ressources. Les problèmes rencontrés lors du déploiement et la configuration des réseaux ont poussé la communauté à chercher des alternatifs. En 2007, des chercheurs ont proposé Tesseract (Yan *et al.*, 2007) qui a comme objectif de séparer la logique de routage, des protocoles de routages utilisés. Tesseract utilise un contrôleur centralisé baptisé *'decision element'* pour insérer directement les règles d'acheminement dans les tables de chaque équipement, cette approche évite les problèmes de dépendances entre les tables (Yan *et al.*, 2007). En 2006, IETF a proposé NETCONF (Enns *et al.*, 2011) comme protocole pour installer, modifier et supprimer la configuration des équipements réseaux. NetConf donne la possibilité aux équipements de fournir une API (*application programming interface*) que les applications peuvent utiliser pour mette à jour la configuration des équipements.

1.1.1 Principe de SDN et objectif

Le SDN (Nunes *et al.*, 2014) est une technologie qui se base sur les mêmes principes que les projets déjà cités. Le SDN a pour principal objectif de simplifier la gestion et le déploiement des réseaux et de faciliter l'innovation dans le réseau en se basant sur la programmabilité. Dans le SDN, le plan de donnée est découplé avec le plan contrôle (voir figure 1.1). Le SDN introduit donc une migration du contrôle qui était au début couplé avec chaque équipement réseau, vers une seule entité de supervision et de gestion. Dans un réseau SDN, les équipements deviennent plus simples puisqu'ils n'ont plus besoin d'intégrer les fonctionnalités du contrôle et les protocoles de routage (ex : OSPF et RIP). Ces équipements interprètent seulement les règles envoyées par le contrôleur. En plus de l'abstraction de réseau, l'architecture SDN supporte une API qui permet d'implémenter les services réseau tels que le routage et le filtrage. Le ONF (*Open Networking Foundation*) essaie de promouvoir une API ouverte et standard pour permettre une gestion multi fournisseurs. Cela ouvre la porte vers de nouvelles approches de design et de développement des protocoles réseau en se basant sur une API ouverte entre le plan du contrôle SDN et les couches d'applications. Les applications utilisent donc une abstraction du réseau qui exprime les services et les capacités réseau sans être liée au détail de l'implémentation. En résumé, le SDN détachent les applications du réseau, et les rend dépendantes seulement de ses services.

1.1.2 OpenFlow

La séparation de plan de donnée, du plan de contrôle nécessite un protocole de communication entre ces deux plans. ONF a choisi Openflow (McKeown *et al.*, 2008) comme protocole standard de communication entre les deux plans. La standardisation de communication permettra de simplifier le développement des applications et de les rendre indépendantes de la plateforme ou de l'infrastructure

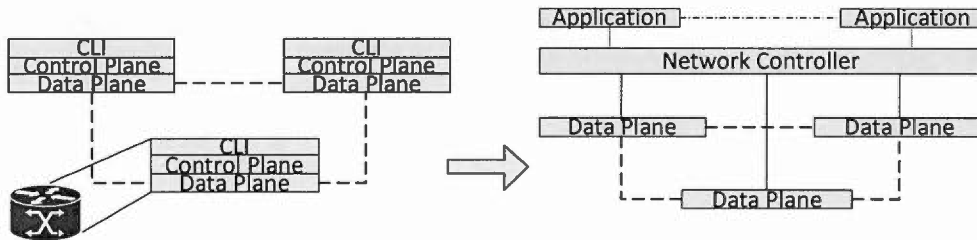


Figure 1.1: Différence entre les réseaux classiques et les réseaux SDN

physique. La description de Openflow et les différentes actions qu'il supporte est détaillée dans le document de spécification [1]. L'Openflow permet d'ajouter, supprimer et modifier des entrées dans les tables de flux dans la *switch*. Une *switch* Openflow contient au moins 3 blocs :

- **Forwarding table** : Pour implémenter Openflow, La switch doit impérativement exposer des tables de flux. Dans Openflow 1.1, une table de flux doit effectuer une recherche sur 14 champs (voir Tableau 1.1), cela permet d'avoir les fonctionnalités de commutation et routage. Dans la version Openflow 1.3, le nombre a augmenté à 41. Cela permet de supporter plusieurs fonctionnalités (ex. : filtrage, load balancing). Les tables de flux doivent supporter aussi le masque (c-à-d ignorer des champs ou des bits dans la classification) pour élargir les possibilités du réseau.
- **Tunnel de communication** : Openflow demande un tunnel sécurisé de communication entre les switches et les contrôleurs. Le tunnel peut se baser sur IPsec (Doraswamy et Harkins, 2003) ou tout autre protocole de cryptage de données.

1. https ://www.opennetworking.org/

Ingress Port	Metadata	Ether src	Ether dst	Ether type	VLAN id	VLAN priority	MPLS label	MPLS traffic class	IPv4 src	IPv4 dst	ARP opcode IPv4 Proto	IPv4 Tos bits	ICMP type Src Port	ICMP code Dst Port

Tableau 1.1: Liste des champs utilisés dans Openflow 1.1

- **Protocole Openflow** : Le protocole d'échange ouvert (Openflow) est nécessaire pour communiquer avec un contrôleur.

1.1.3 Compilation des règles

Le système Cacheflow (Katta *et al.*, 2014) a pour objectif d'améliorer la capacité de traitement des paquets dans une switch logiciel en utilisant une TCAM (*Ternary Contenant Adressable Memory*). Le principe se base sur le caching des règles les plus utilisées en les mettant sur la TCAM (Arsovski *et al.*, 2003; Salisbury, 2012). Le reste du trafic sera traité par la *switch* normale. En effet, la TCAM a une puissance de traitement plus élevée et une latence moins réduite que celles des solutions logicielles. En plus, la TCAM est plus coûteuse et elle se caractérise par une capacité réduite, donc une utilisation optimale de la TCAM est nécessaire. Le mécanisme de *caching* proposé utilise un algorithme qui calcule les relations et les dépendances entre les règles. Le problème des relations entre les règles peut se résoudre par force brute ou par la programmation linéaire. Ces deux solutions permettent de trouver la solution la plus efficace, mais ne peuvent pas être considérées à cause du temps d'exécution. La solution adoptée consiste à résoudre une version réduite du problème du voisin snapback à l'aide d'une heuristique. L'algorithme divise les chaines de dépendances entre les règles pour pouvoir cacher un petit groupe de règles afin de laisser l'espace pour des règles très sollicitées et

ainsi profiter plus de la rapidité de la TCAM. L'algorithme travaille aussi d'une manière incrémentale, un changement des règles peut être calculé facilement ce qui simplifie l'opération de la mise à jour. Le cacheflow utilise seulement la TCAM comme ressource et ne prend pas en compte le réseau multi-tenants où existent plusieurs tables. La combinaison de TCAM et switch logicielle permettra certainement d'améliorer les performances de la switch. Dans (Hamadi *et al.*, 2014), nous avons proposé un modèle d'accélération du trafic qui se base sur le traitement des flux qui consomment plus de ressources (elephant flows) à un processeur réseau, la détection de ces flux se base sur des compteurs installés sur le chemin du traitement logiciel. Comme cacheflow, le système que nous avons proposé ne tient pas compte de la virtualisation. Il vise principalement à découvrir l'intérêt de combiner le matériel et le logiciel pour les applications du traitement des paquets, comme il vise à évaluer la difficulté d'implémentation de ces solutions.

Dans (Zhang *et al.*, 2014), les auteurs ont présenté un mécanisme de placement des entrées des Firewalls et ACL sur une topologie réseau d'une manière optimale. Ce mécanisme se limite seulement aux règles d'ACL. Les règles de routages et de switching sont gérées séparément. Une règle ACL a deux types d'actions : permettre ou bloquer. Ce modèle facilite la création d'un graphe de dépendance entre les règles. Ce graphe est utilisé pour formuler un problème d'optimisation qui peut se résoudre par la programmation linéaire en nombre entiers. Le modèle d'optimisation prend des contraintes liées à la capacité de la TCAM et ne tient pas compte du délai du traitement des paquets. Dans notre travail, nous avons utilisé un modèle basé sur Network-Calculus pour dépasser cette contrainte.

La classification est une des principales causes du blocage dans le déploiement de Openflow sur un réseau, les switches doivent supporter des entrées avec des masques et une classification sur un nombre d'entêtes croissant. Ces contraintes obligent les switches à utiliser des structures de recherches matérielles telles la

TCAM ou bien des solutions logicielles compliquées et lourdes. Pour résoudre ces problèmes, Palette (Kanizo *et al.*, 2013) divise la table du flux sur tout le réseau tout en gardant la sémantique du plan de données. Pour cela, il faut résoudre deux problèmes. D'abord, comment décomposer la table des entrées en petites tables. Ensuite, comment répartir ces tables sur un réseau des switches. Ces deux problèmes sont NP-HARD. La décomposition d'une table se fait à la base de "Pivot bit decomposition" (PBD) ou bien "Cut-based decomposition" (CBD). Dans le PBD, la décomposition des tables se fait en choisissant le bit qui permet de diviser la table en sous-tables avec le même nombre d'entrées. Dans CBD, les tables sont reliés comme un graphe et la distance entre les règles est définie par le nombre des champs qu'il faut modifier, après, la décomposition se fait en utilisant le programme METIS (Karypis et Kumar, 1998). Le deuxième problème est le *mapping* des sous-tables. Pour ça, le réseau est modélisé par un graphe, où les *switches* sont les noeuds et les liens sont les arêtes. Le problème devient un problème de colorisation de graphe qui est NP-HARD. Dans une topologie fat tree (Gomez *et al.*, 2008), souvent utilisée dans les centres de données, la décomposition des tables a une solution optimale. Les auteurs ont proposé aussi des solutions heuristiques qui sont sous-optimales, mais faciles à calculer.

Le SDN, grâce au principe de séparation entre le plan de contrôle et le plan de données a donner la possibilité aux chercheurs de tester de nouveaux protocoles et approches pour contrôler les réseaux sans se soucier de l'hétérogénéité des équipements ou de leurs programmations. La majorité de l'effort se concentre sur la décomposition et l'emplacement des règles en utilisant la programmation linéaire. En outre, ces travaux considèrent un seul tenant dans le réseau et ne prennent pas en compte des dépendances entre les règles des différents tenants. Dans notre travail, nous allons aborder le problème d'un nouvel angle et on va prendre l'aspect multi-tenants en considération.

1.2 Principe de l'analyse formelle des concepts

L'analyse formelle des concepts (AFC) est une méthode pour la fouille de données, la représentation de la connaissance et la gestion de l'information. L'AFC (Ganter et Wille, 1999) est une branche de la théorie des treillis qui étudie comment on peut organiser des groupes des objets hiérarchiquement en fonction de leurs caractéristiques communes. L'AFC a pour objectif de restructurer la théorie des treillis afin de faciliter son utilisation par une large audience. L'AFC est utilisée dans plusieurs domaines dont la réseautique, l'ingénierie logicielle, l'intelligence artificielle et le traitement du texte.

Dans la réseautique, la mesure du trafic est une tâche importante. Elle donne une visibilité aux opérateurs du réseau sur le fonctionnement et les performances de chaque noeud sur un réseau. L'ingénierie de trafic et le déploiement de services demande des mesures précises sur la réponse du réseau. Le problème est que cette tâche consomme beaucoup de ressources, ce qui impacte la performance du réseau. En effet, pour chaque paquet reçu, il faut mettre à jour son compteur. Le problème de mesure du flux se résume à trouver un ensemble minimal des compteurs nécessaires pour répondre aux requêtes d'un opérateur réseau. Des approches comme NetFlow (Claise, 2004) se base sur un échantillonnage du trafic, ainsi les mesures ne sont pas exactes.

Dans le travail (Valtchev *et al.*, 2012), les auteurs ont proposés d'utiliser l'AFC pour trouver l'ensemble minimal de flux à surveiller pour répondre exactement aux requêtes d'un opérateur réseau. L'AFC permet d'exploiter les associations entre la sémantique des requêtes et les flux, et de les exprimer par un treillis. Ce treillis sera analysé afin de déterminer l'ensemble de flux minimal où il faut placer les compteurs. Cette méthode réduit l'utilisation de la mémoire de 90% et garantit une mise à jour d'emplacement des compteurs dans $30ms$. Ce travail montre l'intérêt

d'utiliser les treillis dans des problèmes reliés au réseau et à l'optimisation des ressources.

Dans le travail (Blaiech *et al.*, 2015), nous avons proposé une approche basée sur l'AFC pour adapter les règles OpenFlow aux tables d'une *switch*. D'abord, un schéma de classification adapté à l'architecture matérielle est calculé. Ce schéma de classification répond aux contraintes liées à la complexité de la recherche, la mise à jour et aux limites des tables physiques. Le travail (Hamadi *et al.*, 2015) est aussi relié à l'application de l'ACF dans l'adaptation des règles Openflow aux architectures physiques. Ce travail se concentre sur les *switches* ASICs qui utilisent le modèle de traitement proposé dans (Corp., 2014). Ces deux travaux montrent l'intérêt de l'utilisation de l'AFC et l'analyse sémantique des règles pour optimiser le plan de donnée Openflow dans les *switches*.

Dans le domaine de développement logiciel, l'ACF est utilisable dans la maintenance et le support des logiciels ainsi que dans le modification du code et l'identification des objets dans un modèle de programmation orienté objet. L'AFC est utilisé dans les différentes phases de développement d'un logiciel.

Dans les recherches (Düwel et Hesse, 1998; Düwel et Hesse, 2000), les auteurs ont utilisé l'AFC pour extraire les informations et les classes potentielles à partir d'une description d'un cas d'utilisation souvent donnée comme un texte. L'AFC est utilisée aussi dans la phase de la spécification formelle (Fischer, 2000) et la visualisation de spécification (Spivey, 1989). Au niveau de développement, la FCA est utilisable dans la maintenance (Swanson, 1976), elle permet de développer des outils qui suggèrent des corrections aux logiciels pour les améliorer et pour ajouter des fonctionnalités aux systèmes. Enfin, un aspect important de l'application de la FCA est de la réorganisation de diagramme de classe. Dans les recherches (Snelting, 2000; Snelting et Tip, 1998), les auteurs ont construit des treillis pour

représenter la relation entre les variables et les méthodes si la méthode a un accès à la variable.

1.2.1 Contexte formel

Une des notions de base de l'AFC est le contexte formel, il représente les données à analyser par les algorithmes de l'AFC. Le contexte formel $\mathcal{K} = (\mathcal{R}, \mathcal{H}, \delta)$ se compose de deux ensembles $\mathcal{R}$ et $\mathcal{H}$ et d'une relation binaire δ. $\mathcal{R}$ présente les objets et $\mathcal{H}$ présente les attributs. δ est une relation binaire définie dans $\mathcal{R} \times \mathcal{H}$. On dit que le couple (r, h) où $r \in \mathcal{R}$ et $h \in \mathcal{H}$ satisfait la relation δ si l'objet r satisfait l'attribut (propriété) h. Le couple (r, h) peut être représenté par $r\delta h$. La façon la plus simple pour représenter un contexte formel $\mathcal{K}$ est d'utiliser une matrice de 2 dimensions où les lignes sont les objets et les colonnes sont les attributs (voir Tableau 1.2). Avoir une (x) dans une cellule de l'intersection d'une ligne r et une colonne h signifie que $r\delta h$.

	a	b	c	d	e
1	x		x		x
2		x		x	
3			x		x
4	x	x			x

Tableau 1.2: Exemple d'un contexte formel $\mathcal{K}$

1.2.2 Relation d'incidence entre objets et attributs

À partir de la relation δ du contexte $\mathcal{K}$. On peut définir deux relations pour relier entre les objets et les attributs du contexte.

- $f : \wp(\mathcal{R}) \rightarrow \wp(\mathcal{H})$, $f(R) = \{h \in \mathcal{H} \mid \forall r \in R, r\delta h\}$.

- $g : \wp(\mathcal{H}) \rightarrow \wp(\mathcal{R})$, $g(h) = \{r \in \mathcal{R} \mid \forall h \in H, r\delta h\}$.

Chaque fonction projette un ensemble tiré d'une dimension de la table vers l'autre. L'image par f d'un ensemble R est l'ensemble de tous les attributs communs à tous les éléments de R. Intuitivement l'image d'un ensemble par f est calculée par l'intersection des lignes par rapport aux arguments. L'image d'un ensemble par g peut se calculer de la même manière. Par exemple, dans la table (voir Tableau 1.2), $f((1,2)) = (a,e)$ et $g((c,e)) = (1,3)$. Cela signifie que les objets $(1,2)$ partagent les attributs (a,e). De la même manière, les attributs (c,e) ont en commun les objets $(1,3)$. Le couple (f,g) représente une connexion de Galois (Denecke *et al.*, 2013).

1.2.3 Concepts

Définition 1.1. *Un concept $C = (R,H)$ est un couple qui appartient à $\mathcal{R} \times \mathcal{H}$ et qui satisfait la relation $f(R) = H$ et $g(H) = R$.*

Soit $C = (R,H)$ est un concept, alors R et H sont des ensembles fermées, en effet, $f(g(R)) = R$ et $g(f(H)) = H$. R et H sont appelés respectivement l'intention et l'extension du concept. Par exemple, $\{(1,4),(a,e)\}$ est un concept formel. Donc $f(1,4) = (a,e)$ et $g(a,e) = (1,4)$. Les concepts ont une propriété utile. Un concept est maximal sur les deux dimensions : cela est compréhensible facilement par la bijection qui existe entre les concepts et les bi-cliques maximales (c-à-d un graphe biparti complet) du graphe biparti induit par δ sur $(\mathcal{R},\mathcal{H})$. Le concept (R,H) se caractérise par un ensemble des générateurs minimaux. Ces générateurs minimaux sont des sous-ensembles de H et ont la même image de H par g. Aussi, aucun sous-ensemble d'un générateur minimal ne doit avoir la même image que H(ex. : dans le concept $(\{(4);(a,b,e)\}$, $\{(a,b);(b,e)\}$ sont des générateurs minimaux puisqu'ils ont comme image $\{4\}$ et aucun de leurs sous-ensembles n'a la même image).

1.2.4 Treillis de concepts

L'ensemble des concepts du contexte $\mathcal{K}$ constitue un ensemble partiellement ordonné par la relation d'inclusion des intentions/extensions :

$$R_1, H_1 \leq_K (R_2, H_2) \Leftrightarrow R_1 \subseteq R_2, H_2 \subseteq H_1$$

En effet, la relation $\leq_K$ est réflexive, antisymétrique et transitive.

Définition 1.2. *Un treillis est un couple ($\mathcal{E}$,$\leqslant$) où $\leqslant$ est une relation d'ordre partielle et chaque paire d'élément (x, y) admet une borne supérieure et une borne inférieure.*

Le couple $(\mathcal{C}, \leq_K)$ où $\mathcal{C}$ est l'ensemble des concepts d'un contexte $\mathcal{K}$ et $\leq_K$ est la relation d'inclusion des intentions/extensions est un treillis. L'ensemble des concepts $\mathcal{C}$ peut être représenté par le diagramme de Hasse (voir Figure 1.2).

1.3 Principe du *network calculus*

Le *network calculus* est une théorie qui s'intéresse aux problèmes de flux dans les réseaux. Cette théorie donne un ensemble d'outils pour évaluer les performances d'un réseau de communication. Elle permet d'établir des limites minimales sur la latence et la bande passante. La théorie se concentre sur l'analyse des métriques de performance dans le pire des cas. Le *network calculus* se base sur la théorie des dioïdes, et particulièrement le dioïde (min,+) souvent appelé l'algèbre (min,+) (Gondran et Minoux, 2000).

1.3.1 Granularité du modèle de flux

Dans un réseau de communication, le flux se représente par une fonction cumulative $R(t)$ qui définit le nombre de bits ou des paquets reçus dans un intervalle de temps $[0, t]$. Par définition la fonction R est croissante sur son domaine de

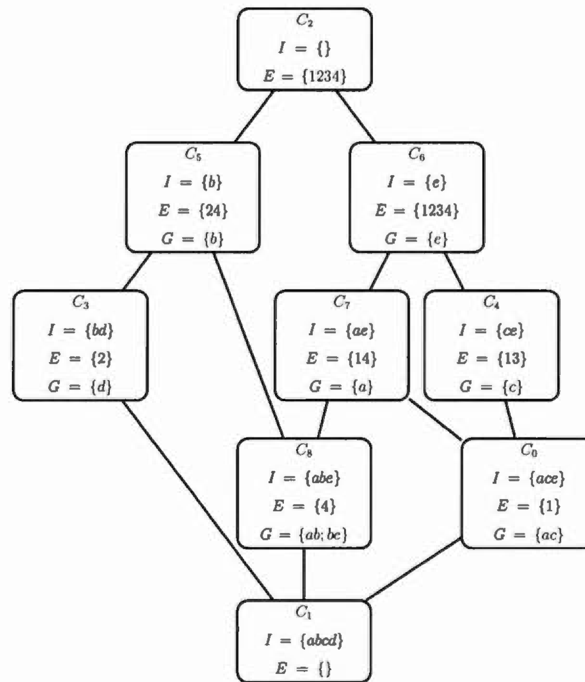


Figure 1.2: Diagramme de Hasse de treillis des concepts

définition $[0, +\infty[$. Aussi on considère en général que $R(0) = 0$ (ex. : le système commence la réception des données à l'instant $t = 0$). Le flux se modélise par un modèle fluide ou bien discret. Cependant, il est possible de passer d'un modèle fluide à un modèle discret en utilisant l'échantillonnage sur chaque intervalle de δ donc on peut prendre $S(n) = R(n\delta)$. De même on peut passer d'un modèle discret à un modèle fluide en utilisant la fonction $R(t) = S(\lceil \frac{t}{\delta} \rceil)$. Dans la suite on considère que le modèle est fluide.

1.3.2 Charge et délai dans un systèmes

On considère un système $\mathcal{S}$ et $R(t)$ la fonction d'arriver des paquets. On note $R^*(t)$ la courbe de sortie des paquets, $R^*(t)$ est cumulative et croissante sur l'intervalle $[0, +\infty[$. On définit ainsi la charge et le délai de traitement par :

Définition 1.3. *La charge (backlog) à un instant t est définit par le nombre des*

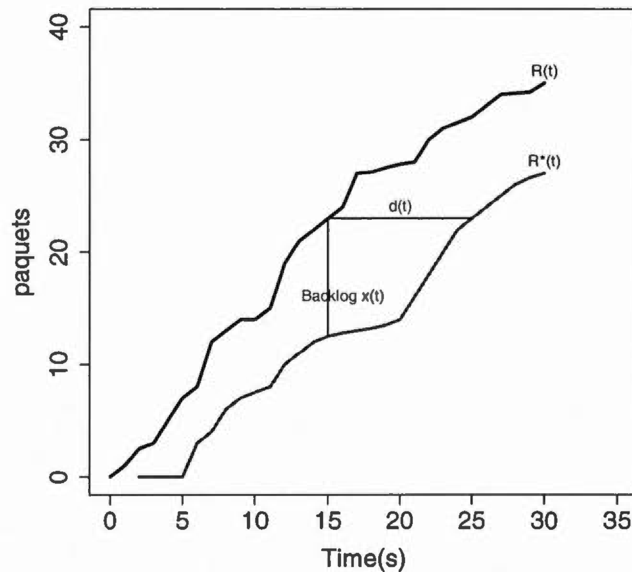


Figure 1.3: La charge et le délai dans un système

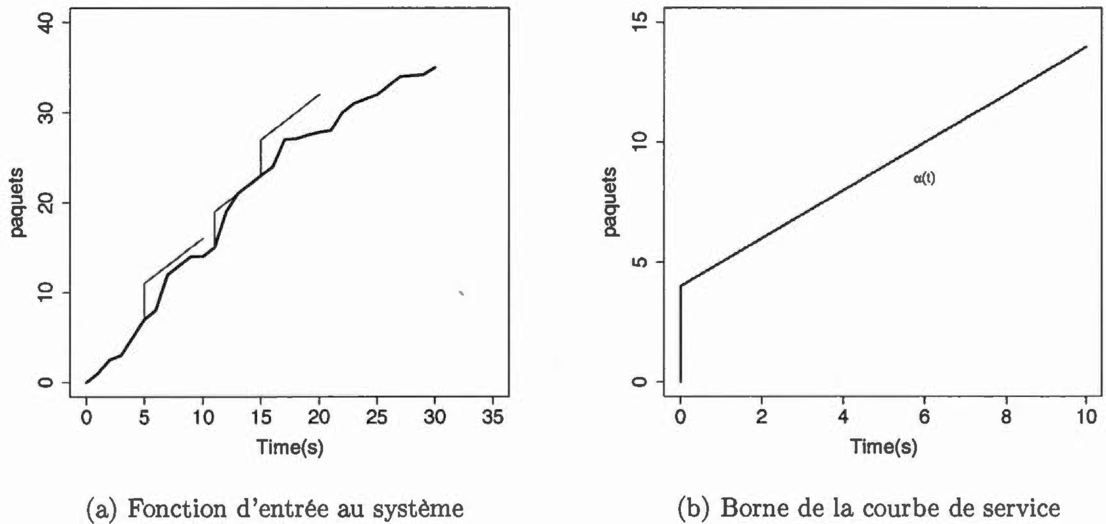
paquets qui attendent d'être traités dans le système. Formellement il est définit par :
$R(t) - R^*(t)$.

Définition 1.4. *Le délai à un instant t définit le temps d'attente d'un paquet arrivant à l'instant t. Formellement, le délai* $d(t) = inf\{\tau : R(t) \leq R^*(t+\tau)\}$.

1.3.3 Les courbes d'arrivée et de services

Dans un système réel, avoir une charge infini n'est pas possible. Aussi, le délai de traitement d'un paquet ne doit pas dépasser certaine limite. Pour respecter ces contraintes, il est nécessaire d'évaluer le système pour borner les fonctions d'arriver.

Définition 1.5. *On dit qu'une fonction* α *définit sur* $t \geq 0$ *est une courbe d'arrivée*



(a) Fonction d'entrée au système

(b) Borne de la courbe de service

Figure 1.4: Borne de la courbe d'arrivée

pour R si :

$$\forall (s,t) \in [0,+\infty[\times[0,+\infty[. \quad s \leq t \rightarrow R(t) - R(s) \leq \alpha(t-s)$$

Une des courbes d'arrivées les plus utilisées est la fonction affine $\alpha_{\sigma,\rho}(t) = t\sigma + \rho$. Cette courbe permet à la source d'envoyer ρ paquets simultanément, après il limite le trafic à σ paquets par seconde.

Définition 1.6. *Soit S un système et R et R^* sont ses courbes d'arrivée et de sortie. On dit que S assure une courbe de service β au flux si et seulement si β est croissante, $\beta(0) = 0$ et $R^* \geq R \otimes \beta$*

1.3.4 Utilisation du network calculus dans l'analyse des performances

Le network calculus est utilisé dans l'analyse des performances des réseaux SDN (Bozakov et Rizk, 2013; Gangwal *et al.*, 2005) et aussi dans l'évaluation des performances des *switches* (Azodolmolky *et al.*, 2013).

Dans (Azodolmolky *et al.*, 2013), les auteurs ont effectué une analyse sur des *switches* Openflow. L'étude modélise la *switch* par trois entités : une queue d'entrée, une table de recherche et une queue de sortie. De même le contrôleur est représenté par les mêmes entités. L'étude analyse les contraintes reliées aux queues et donne leurs bornes en terme de capacité. Aussi, elle analyse la courbe de service de la *switch* et donne une borne maximale de la latence de la *switch*. Ce modèle simpliste de la *switch* simplifie le calcul et l'utilisation de network calculus, cependant, il ne reflète pas la nature des applications développées et déployées en production. En effet, une *switch* se compose de plusieurs tables de recherches et de plusieurs queues d'entrées/sorties.

Dans (Mahmood *et al.*, 2014), les auteurs ont proposé un modèle d'évaluation d'un réseau SDN. Le modèle considère la communication avec le contrôleur et traite une seule *switch* dans le plan de donnée. les queues dans le réseau sont représentées par un modèle de Jackson (Jarschel *et al.*, 2011). L'étude traite le cas de plusieurs nœuds dans le réseau et montre que les résultats trouvés en simulation sont similaires au cas réel. Cette étude se limite à un seul motif de trafic, par conséquent, ces résultats sont valides seulement pour ce motif, un changement de trafic demandera une nouvelle vérification des bornes maximales de la latence et maximale de la bande passante. Dans (Osgouei *et al.*, 2015), les auteurs ont créé un modèle pour calculer les bornes maximales de latence et de la bande passante dans un réseau SDN virtuel en utilisant le *network calculus*. Ces bornes sont calculées pour chaque réseau virtuel. Dans (Guck et Kellerer, 2014), une couche d'abstraction entre les applications temps réel et le réseau basée sur le network calculus été présentée. L'objectif de cette abstraction est d'assurer une qualité de service de bout-en-bout. Dans ce modèle, les applications n'ont pas une connaissance sur le matériel, chaque application spécifie ses besoins en terme de latence et bande passante, la couche d'abstraction fait la correspondance nécessaire sur le

matériel pour assurer les contraintes des applications.

CHAPITRE II

COMPILATEUR MULTI-TENANTS

Le modèle proposé pour la gestion et la compilation des règles sur un réseau multi-tenants se base principalement sur la création d'un plan de données unifié qui traite le trafic de tous les tenants. La première partie de ce chapitre détaille les contraintes du design et de la conception d'un compilateur des règles. Après, il présente le design du compilateur et ses différentes composantes.

2.1 Logique de conception

Optimiser le transit d'un paquet dans le réseau nécessite en premier la compréhension des contraintes et des limitations des *switches* matérielles. Et aussi des conditions du traitement du trafic.

2.1.1 Architecture des switches

Pour insérer une règle dans une *switch*, la règle se décompose en plusieurs entrées. Chaque entrée contient une partie de la règle en plus de quelques *metadata* pour relier les entrées. Les entrées sont ajoutées séparément au niveau des tables de flux déterminées à l'avance. Une table peut être basée sur plusieurs types de mémoire (ex. : TCAM, SRAM, DRAM). Chacune des mémoires se caractérise par plusieurs caractéristiques (ex. : nombre d'entrées supportées, latence, type de recherche). Les tables utilisées et leurs dispositions influencent les fonctionnalités et les per-

formances d'un équipement réseau et permettent de déterminer sa flexibilité et sa programmabilité.

L'architecture des *switches* se divise en trois catégories. La première est une *switch* logicielle (Pfaff *et al.*, 2015). Elle utilise les mémoires (SRAM ou DRAM) et s'exécute sur des processeurs génériques (ex. : x86). Puisque les structures de recherches sont logicielles, elles sont flexibles et ne limitent ni les champs à utiliser ni la longueur de la clé. Cette flexibilité permet d'adapter les tables et les pipelines en fonction de motif des règles. Par exemple, si toutes les applications réseau n'utilisent pas un champ d'entête (ex. : port source), alors on ne réservera pas un espace dans une table de flux pour ce champ. Cela aide à optimiser les ressources. Un NPU (*Network Processor Unit*) (Thiele *et al.*, 2002b) utilise souvent des accélérateurs (ex. : TCAM et HASH), ces accélérateurs aident les applications à améliorer la performance du traitement du paquet. Les applications peuvent spécifier parfois les champs à utiliser dans les tables ou bien la longueur de la clé, mais ils ne peuvent pas réorganiser les tables ou bien créer des nouvelles structures de recherches. Le NPU (ex. : nfp-3200) offre une flexibilité considérable, mais présente plus de contraintes par rapport à une solution logicielle. Dans le cas d'une *switch* ASIC (ex. : Broadcom trident II), les tables sont configurées à l'avance avec des champs d'entêtes spécifiques. Aussi, le pipeline de traitement est fixe. L'ASIC (*Application-specific integrated circuit*) ne donne pas cette flexibilité, mais offre une meilleure performance, surtout en terme de latence. L'implémentation d'une application du plan de données sur ces architectures doit prendre en compte des tables disponibles et des contraintes de l'architecture matérielle (voir figure 2.1). Dans les 3 catégories qu'on présente, le NPU présente le meilleur compromis entre performance et flexibilité. À partir de la, on conclut que l'architecture d'une *switch* joue un rôle important dans la création et la définition des pipelines.

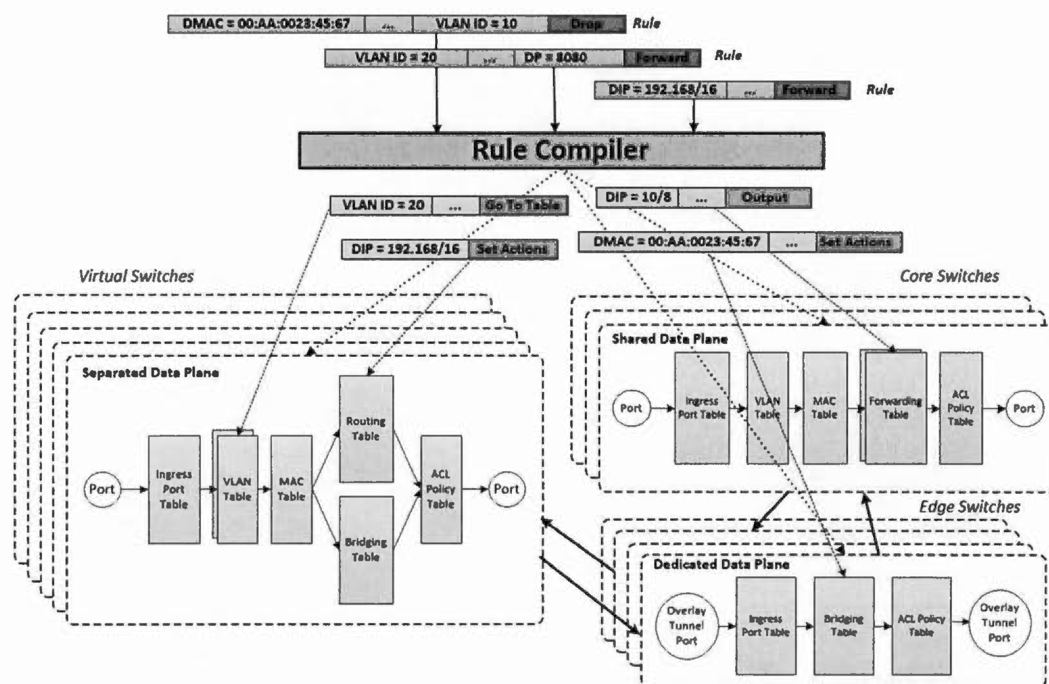


Figure 2.1: Exemple de plans de données de différentes architectures

2.1.2 Sémantique des règles

Une règle est un ensemble d'entrées où chacune est composée de plusieurs champs (*matchfields*). Une entrée reflète ainsi la description d'un paquet par une combinaison des matchfields comme une combinaison des adresses MAC, des adresses IP, port source et destination. Aussi, une entrée peut contenir des metadata qui ne sont pas reliés à l'entête du paquet. Le port d'entrée d'une *switch* est un exemple de metadata. Les règles peuvent partager plusieurs champs entre eux. Par exemple, si une règle R_i est définie par une adresse IP=132.208/16 et une autre règle R_j est définie par la même adresse IP, et si ces deux règles spécifient la même action (ex. : envoyé vers la même table ou bien par le même port), alors les deux règles peuvent être enregistrées par une seule entrée. En conséquence, on avance que les régularités combinatoires dans la distribution des entrées peuvent être exploitées

pour améliorer la transmission des paquets.

Un prétraitement des règles suivant leurs sémantiques permet de détecter les relations cachées entre les règles et l'ensemble des matchfields. Ces relations peuvent être réarrangées et disposées différemment pour révéler des sous-ensembles de règles homogènes par rapport aux matchfields partagés, et pour créer une structure de classification plus compacte (ex. : une optimisation globale de l'ensemble des entrées sur le pipeline des tables).

2.2 Approche de conception

Pour répondre aux différentes limitations de la transmission des paquets dans les réseaux multi-tenants, nous avons conçu et mis en œuvre un compilateur qui utilise une abstraction du modèle de transmission et qui considère les contraintes des *switches* pour générer un schéma de traitement des paquets.

Le compilateur réside entre les contrôleurs des tenants et l'infrastructure physique (voir figure 2.2). Ayant une vue globale du réseau, ceci lui permet d'effectuer des vérifications sur la consistance des règles et les mises à jour demandées par les contrôleurs. Le compilateur peut avoir une architecture distribuée ou bien centralisée. Dans une architecture centralisée, toutes les tâches sont exécutées au niveau du contrôleur. Chaque *switch* utilise un simple client pour installer les règles reçues. Dans une architecture centralisée, le compilateur central s'assure de la consistance et distribue les règles sur les *switches*. Ensuite, chaque *switch* compile ses propres règles suivants ses propres contraintes.

Pour prévenir chevauchement des règles, le module *rule validation* (1) communique avec les contrôleurs et les applications réseau (voir figure 2.3), il communique aussi avec l'administrateur qui spécifie la politique de virtualisation utilisée. Si un chevauchement se présente, une requête est envoyée au module du *rule differentiation*

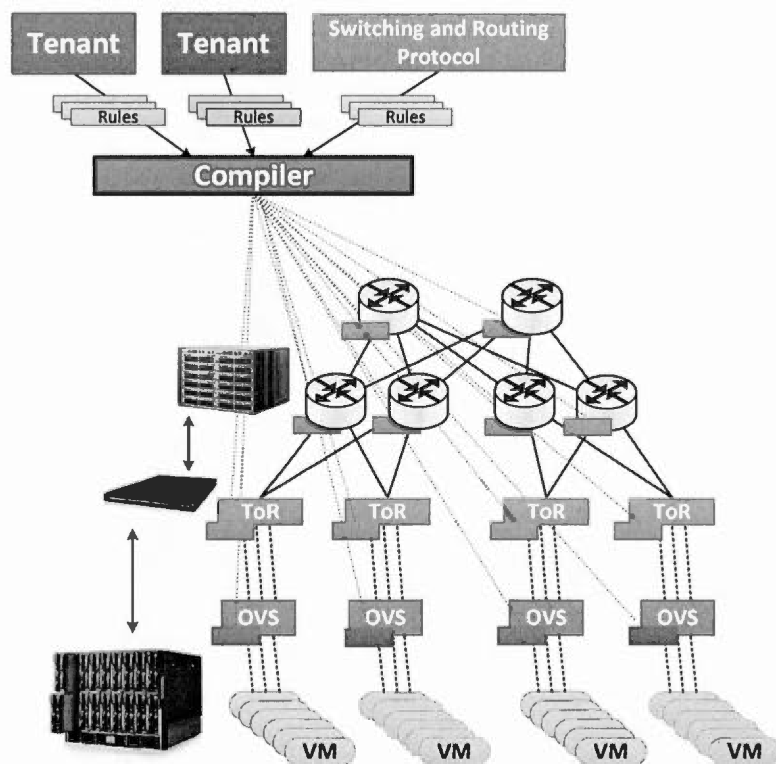


Figure 2.2: Vue global d'emplacement du compilateur

qui étend la règle en ajoutant des champs pour l'enlever. Une fois que toutes les règles sont validées, le *rule miner* calcule les motifs fréquents fermés (Pasquier, 2009) en se basant sur les concepts de l'AFC (Ganter *et al.*, 2005). Ces motifs seront utilisés pour la modélisation et la construction du schéma de classification dans les *switches* (4). Enfin, ce schéma de classification est évalué par rapport aux contraintes des *switches*.

2.2.1 Terminologie et notation

Dans cette section, on définit un ensemble de termes qu'on utilise :

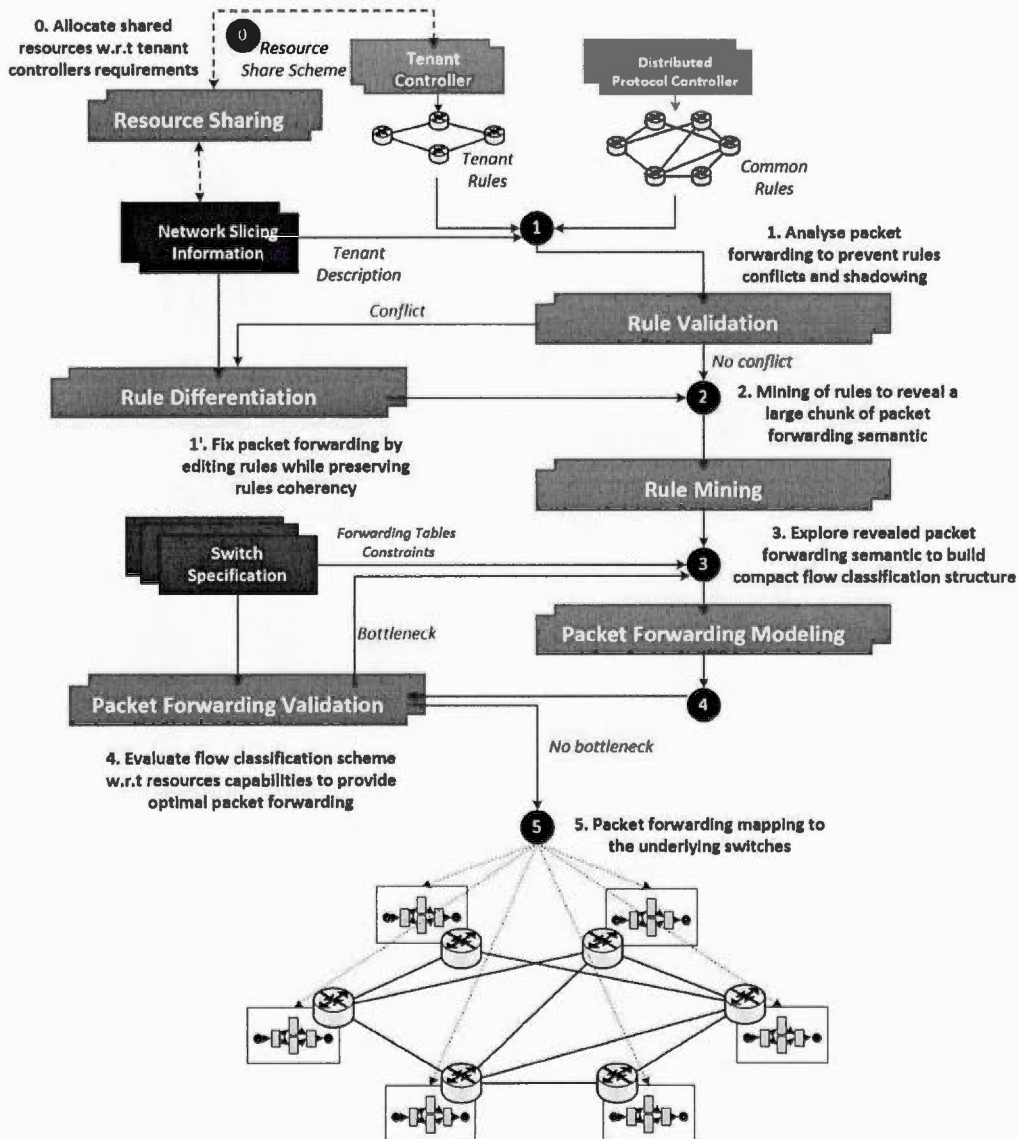


Figure 2.3: Vue générale d'un compilateur multi-tenants

— **Matchfield.** Présente un couple de champ d'entête (*field*) et sa valeur (ex. : IP source =10.1.2.0/24).

— **Règle.** Une règle se compose de plusieurs *matchfields* $H = h_1, ..., h_n$ et une action à appliquer. Une règle peut être générée par le contrôleur d'un tenant, par une application réseau ou par une configuration manuelle.

— **Différenciateur.** Présente un ensemble des descripteurs de paquets $D = h_1, ..., h_m$. Si $i \neq j$ alors $D_i \neq D_j$. Les différenciateurs sont définis préalablement par la politique de virtualisation et sont exclusifs pour chaque réseau logique (ex. :{VLAN=10, SIP=10/8}, {VLAN=20, SIP=192.168.1/24, SIP=10/8}). Aucun flux ne peut faire partie de deux différenciateurs à la fois.

— **Tenant.** caractérisé par un ensemble de différenciateurs $\{D_1, ..., D_l\}$.

— **Switch.** se compose d'un pipeline de tables $S = t_1, ..., t_p$. Les tables peuvent être logicielles (ex. : SRAM et DRAM) ou bien matérielles (ex. : TCAM). Chaque table se caractérise par le nombre d'entrées supportées, la longueur maximale de chaque entrée et le type de recherche utilisé (meilleure correspondance, exacte, ternaire).

— **Réseau.** décrit la topologie et la liaison entre les switches. Un réseau peut être représenté par une seule switch (c-à-d une règle définie ainsi la liaison de bout en bout d'un paquet)

2.2.2 Validation et différentiation des règles

Le compilateur définit une couche d'abstraction entre les entités de contrôles et l'architecture physique. L'objectif de cette abstraction est d'empêcher une entité de contrôle de perturber le fonctionnement d'une autre entité. Sans validation et différentiation, une entité peut insérer des règles qui modifient le trafic des autres entités. En conséquence, les règles doivent être validées et differenciées pour assurer une isolation du contrôle.

D'abord, les domaines de définition des tenants doivent être mutuellement disjoints. Pour vérifier cette condition, le compilateur compare entre leurs ensemble de définition. Puis, il construit une structure de classification séparément pour chaque tenant. Chaque structure contient des sous-structures pour gérer chaque type de champs. Une sous structure supporte un des trois types de recherches

suivantes :

- **Meilleure correspondance** : (*longest prefix match*) est une recherche pour la meilleure correspondance, elle est utilisée pour les champs qui ont un préfixe (ex. : IP source, IP destination).
- **recherche ternaire** : Cette recherche permet d'ignorer des bits dans l'opération de la recherche en introduisant le concept de masque.
- **Recherche exacte** : Donne un résultat qui correspond exactement à la clé de la recherche. Dans notre contexte, on l'utilise pour tous les autres champs qui ne sont pas mentionnés avant.

Après la construction de toutes les structures de recherches nécessaires. L'algorithme de détection du chevauchement entre les tenants en utilisant le principe de diviser pour régner (voir Algorithme 1). L'algorithme prend chaque différenciateur et il le compare avec les différenciateurs des autres tenants. L'algorithme extrait d'abord les champs du différenciateur. Puis effectue une recherche de chaque champ séparément dans la sous-structure des autres tenants relatifs à ce champ. À la fin, le résultat est l'intersection de toutes les recherches. Cet algorithme s'exécute une seule fois ou bien après la modification des caractéristiques de tenants.

Si les différenciateurs des tenants sont mutuellement disjoints, alors il suffit de vérifier qu'une règle appartient à son tenant pour s'assurer qu'elle ne va pas affecter le trafic des autres tenants. Ce processus de vérification des règles s'exécute chaque fois que le compilateur reçoit une règle d'un contrôleur. Si une règle est spécifiée sur un champ plus large que son tenant, le compilateur va détecter un chevauchement et il va activer le module de différentiation qui permettra de spécifier la règle en ajoutant d'autres *matchfields* ou bien en modifiant les *matchfields* de la règle pour qu'elle soit conforme à son tenant. Par exemple, si une règle R est définie par $\{IPdestination = 10/8, Action = Bloquer\}$ et son tenant T_i est défini par $\{IPdestination = 10.0/16\}$ alors l'entête de la règle sera transformée en $\overline{R} =$

Algorithm 1: Validation des différenciateurs des tenants

Objectif : Détecter les conflits entre les différenciateurs des tenants.

S Le nombre des *swtiches.*

T L'ensemble des tenants.

SC Un ensemble pour enregistrer les différenciateurs en chevauchement

$TD_{t,s}$ L'ensemble des différenciateurs du tenant t au niveau de la *switch* s

---

for $s \in S$ **do**
  for $t \in T$ **do**
    for $H \in TD_{t,s}$ **do**
      $SC = \{ALL\}$
      for $h_i \in H$ **do**
        $SC = SC \cap lookup(h_i, TD_{f_i})$
      if $SC \neq \emptyset$ **then**
        return $\{s, t, H, SC\}$

$T_i \cap R$=$\{IPdestination = 10.0/16\}$. Autrement, si le compilateur ne change pas la règle et un autre tenant est défini par $\{IPdestination = 10.1/16\}$, R va affecter le trafic de T_j (c-à-d tout son trafic sera bloqué). Dans le cas où les tenants sont définis par des règles de *exact match* comme les VLAN, si une règle R appartient à un tenant T_t défini par $\{D_1 = \{VLAN = 10\}, D_2 = \{VLAN = 20\}\}$, donc après pré-compilation la règle se transforme en $\overline{R}_1^t = \{SIP = 10/8, VLAN = 10\}$ et $\overline{R}_2^t = \{SIP = 10/8, VLAN = 20\}$. Notre méthode de détection des conflits se base sur la même approche que la validation des tenants. L'article (Natarajan *et al.*, 2012) présente plus de détails sur une approche similaire.

Le résultat de la phase de validation et de différentiation est un ensemble des règles à compiler. La préparation des règles est illustrée comme suit : On suppose

Tableau 2.1: Exemple des règles de tenant et des règles communes

Ensemble des règles de tenant 1				
Rule	DIP	VLAN	InPort	Protocol
R_1^1	10.1/16	5	*	tcp
R_2^1	10.2/16	6	2	udp
R_3^1	10.3/16	6	1	8

Ensemble des règles de tenant 2				
Rule	DIP	VLAN	InPort	Protocol
R_1^2	10.1/16	6	3	*
R_2^2	10.2/16	9	4	*
R_3^2	30.3/16	7	1	tcp

Ensemble des règles communes					
Rule	SIP	DIP	VLAN	InPort	Protocol
R_1^c	172.168/16	10.1/16	5	*	*
R_2^c	192.18.0/24	*	9	5	*

que $T_1 = \{SIP = 172.168.1/24, InPort = \{1,2\}\}$ et $T_2 = \{SIP = 172.168.2/24\}$, et les règles qui correspondent à chaque tenant sont représentées dans la table 2.1. Dans notre exemple, le compilateur détecte le chevauchement entre R_1^1 et R_1^c et il transforme R_1^1 en conséquence. La phase finale est la construction d'un contexte global qui contient toutes les règles. Ce contexte présente la relation entre les règles et les *matchfields* (voir Table 2.2).

2.2.3 Fouille des règles

À ce stade de traitement, nous avons détecté et résolu les problèmes de chevauchement des règles. Donc, il est maintenant possible de procéder à la création des chemins de traitement unifiés tout en assurant une isolation du trafic. L'étape sui-

Tableau 2.2: Contexte global

Global context																			
Rule	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}	h_{11}	h_{12}	h_{13}	h_{14}	h_{15}	h_{16}	h_{17}	h_{18}	h_{19}
R_1^1	x				x				x				x					x	
$\overline{R}_1^1$	x				x				x					x				x	
R_2^1	x					x				x				x					x
R_3^1	x						x			x			x						
R_1^2		x	x		x					x					x				
R_2^2		x	x			x						x				x			
R_3^2		x	x					x			x		x					x	
R_1^c			x		x				x										
R_2^c				x								x					x		

	SIP		DIP		VLAN		InPort		InPort		Protocol
h_1	172.168.1/24	h_5	10.1/16	h_9	5	h_{13}	1	h_{17}	5	h_{18}	tcp
h_2	172.168.2/24	h_6	10.2/16	h_{10}	6	h_{14}	2			h_{19}	udp
h_3	172.168/16	h_7	10.3/16	h_{11}	7	h_{15}	3				
h_4	192.18.0/24	h_8	30.1/16	h_{12}	9	h_{16}	4				

vante est l'analyse de l'ensemble des règles pour calculer et extraire des groupes ou des sous-ensembles des règles qui partagent et qui se caractérisent par un ensemble des. *matchfields*. Ces *matchfields* seront ainsi une clé d'identification dans le processus de la classification pour ces règles. La fouille des règles est basée sur l'analyse formelle des concepts (AFC) (Ganter et Wille, 1999). Cette fouille permet d'extraire l'ensemble de tous les groupes des règles potentiellement utiles. Ces groupes partagent les mêmes descripteurs des règles. Chaque groupe a aussi une ou plusieurs clés minimales (Un ensemble minimal des descripteurs discriminatoires) qui seront exploitées dans la construction de l'arbre.

La fouille des règles se base sur la théorie des fouilles de données (AFC) et les treillis de Galois. Les sous-ensembles qu'on cherche sont des concepts formels. Le contexte formel dans notre est $\mathcal{K} = (\mathcal{R}, \mathcal{H}, \delta)$ où $\mathcal{R}$ est l'ensemble des règles

(objets) et $\mathcal{H}$ l'ensemble des *matchfields* (attributs), et δ une relation binaire entre $\mathcal{R}$ et $\mathcal{H}$ appelée relation d'incidence de $\mathcal{K}$. $\delta \subset \mathcal{R} \times \mathcal{H}$ est basée sur l'association du paquet : un couple $(r, h) \in \delta$ (noté aussi $r\delta h$) signifie que tous les paquets qui correspond à r vérifient h.

Le calcul de tous les concepts n'est pas un problème polynomial puisqu'il peut avoir un nombre exponentiel des concepts. Par contre, dans des conditions réalistes, le nombre des concepts reste dans un ordre polynomial, ainsi le cout de calcul des concepts est rarement prohibitif. De plus, contrairement à la majorité des algorithmes de ACF qui calculent tous les concepts, il existe des algorithmes (Stumme *et al.*, 2000) dans l'ACF qui sont paramétrables et calculent seulement les concepts qui respectent quelques conditions (ex. : le support). Dans notre cas, nous avons utilisé l'algorithme (voir Annexe A).

L'ensemble des concepts du contexte $\mathcal{K}$ constitue un ensemble partiellement or-

Tableau 2.3: Exemple des concepts

Id	**Intension**	**Extension**	**Générateur**
C_{11}	h_{10}	R_2^1,R_1^2,R_3^2	h_{10}
C_{15}	h_6	R_2^c,R_2^1,R_2^2	h_6
C_{18}	h_{18}	$\overline{R}_1^1$,R_1^1,R_3^2	h_{18}
C_{22}	h_5, h_9	$\overline{R}_1^1$,R_1^c,R_1^1	h_9
C_8	h_1,h_{10}	R_2^1,R_3^1	h_1 ,h_{10}
C_{13}	h_2,h_3,h_5 h_{10},h_{15}	R_1^2	h_2,h_5 ; h_2,h_{10} h_5,h_{10} ; h_{15}
C_2	h_1,h_5 h_9,h_{18}	R_1^1,$\overline{R}_1^1$	h_1,h_5 ; h_1,h_{18} ; h_5,h_{18} h_1,h_9 ; h_9,h_{18} ; h_5,h_9

donné par la relation d'inclusion des intentions/extensions :

$$R_1, H_1 \leq_K (R_2, H_2) \Leftrightarrow R_1 \subseteq R_2, H_2 \subseteq H_1$$

En effet, la relation $<_K$ est réflexive, antisymétrique et transitive. En plus de cette relation, chaque ensemble admet une borne supérieure et une borne inférieure. Donc l'ensemble des concepts est un treillis complet (Sankappanavar et Burris, 1981). L'ensemble des concepts organisés suivant la relation d'ordre partielle $\leq_K$ (voir figure 2.4). Les informations reliées à chaque concept sont détaillées (voir Tableau B.1) dans (voir Annexe B).

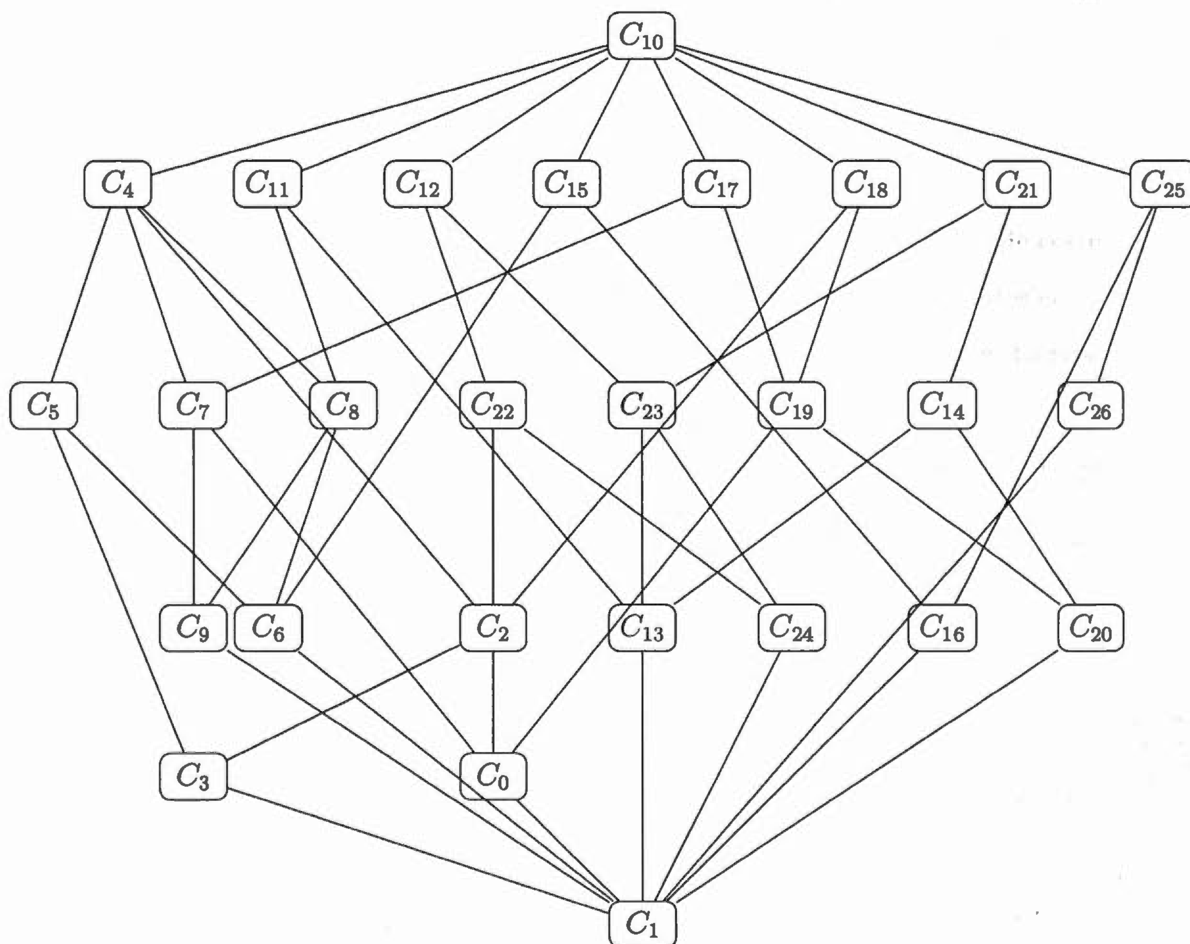


Figure 2.4: Diagramme de Hasse de treillis des concepts

2.2.4 Modélisation de transmission de paquets

Le classificateur de paquets analyse l'ensemble des concepts calculés par les algorithmes des fouilles de données. Le classificateur sélectionne un nombre de concepts qui satisfait des critères comme la couverture de toutes les règles. Ce processus de sélection est non trivial et sera détaillé par la suite. Les concepts choisis seront organisés dans une forme d'arbre d'index (voir Figure 2.5a). La construction d'un arbre d'index équilibré représente certainement un challenge. Mais sa maintenance après les mises à jour due au changement d'état du réseau est difficile.

Construction de l'arbre de classification

Organiser l'ensemble des règles dans une structure d'arbre compacte réduit le nombre de tests nécessaires pour trouver la règle correspondante à un paquet et ainsi l'action à appliquer. L'utilisation d'un arbre compact réduit l'utilisation des ressources de la *switch*. Un paquet arrivant traverse l'arbre des index pour trouver un sous-ensemble qui lui correspond. La construction de l'arbre d'index est une opération directe et dérive de la sélection des concepts. La sélection des concepts formels qui représentent les nœuds de l'arbre d'index dépend de plusieurs critères indépendants ainsi que des paramètres reliés aux matériels. La sélection optimise le choix suivant ces contraintes. Premièrement, l'union des extensions des concepts choisis doit couvrir toutes les règles du contexte formel. Le fait de ne pas couvrir les règles amène une structure de recherche incomplète et qui ne sera pas fonctionnellement valide, ainsi, La couverture des règles est une condition nécessaire. Et deuxièmement, pour optimiser la structure de classification en terme de rapidité et d'espace consommé, la taille des clés des indexes doit être minimale en terme de nombre et de longueur (la somme des longueurs des *matchfields*). Finalement, il faut que les extensions des concepts sélectionnés aient un chevauchement minimal. En effet un chevauchement induit une répétition d'une règle sur deux sous

ensembles des règles, ce qui va induire une perte des ressources.

Les architectures flexibles (ex. : x86) n'imposent pas plus de contraintes, donc la création d'un pipeline à partir des concepts choisis est simple et plus efficace puisqu'elle ne demande pas une adaptation à l'environnement. Par contre, dans les architectures rigides (ex. : processeur réseau, ASIC), la solution nécessite une adaptation aux contraintes de l'architecture. Ces contraintes limitent les bénéfices de notre schéma de classification. Par exemple, on doit s'assurer que les *matchfields* des générateurs minimales des concepts sont supportées par les tables physiques en terme de longueur et de type de recherche. Si le problème est lié à la taille des clés, le choix des concepts prendra en compte la taille maximale d'une clé et favorise les concepts avec des générateurs minimaux respectant la contrainte. Généralement, le processus de choix des concepts tente à utiliser un nombre minimal des champs, les concepts seront évalués par rapport aux champs utilisés dans leurs générateurs. Dans le cas d'une *switch* où les tables physiques sont arrangées suivant un ordre spécifique et supportent des champs déterminés à l'avance. Les indexes seront adaptés suivant l'organisation des tables, par exemple si le pipeline contient 4 étages –Metadata (ex. : port d'entrée, id d'un tunnel), L2,L3 et L4– la construction d'index ainsi que son *mapping* ne va jamais inclure un champ L3 avant un champ L4, ainsi l'ordre du pipeline reste respecté. Dans l'exemple (voir Tableau 2.3), les deux ensembles des concepts C_{11}, C_{15}, C_3,C_{18}, C_{22} et C_{15}, C_{18}, C_{22}, C_8, C_{13} couvrent toutes les règles avec un chevauchement moyen entre les extensions. Dans ce cas, l'ensemble C_{11}, C_{15}, C_3,C_{18}, C_{22} est plus favorable puisque toutes ses clés utilisent un seul champ. Pour assurer une structure d'arbre compacte, homogène et équilibrée (ex. : nombre d'entrées dans chaque sous-ensemble), on utilise un seuil pour la fouille de donnée. Incrémenter le seuil réduit le nombre des concepts, ce qui augmente la probabilité de chevauchement entre les sous-ensembles. D'un autre côté, réduire le seuil augmente le nombre des concepts ce qui induit un

nombre des index élevé et donc une structure d'arbre large et non performante. Le choix du seuil de fouille de donnée est un compromis entre la compacité de l'arbre et le chevauchement permis entre les sous-ensembles. Dans notre exemple, le seuil est de 3 règles par sous-ensemble. Pour des raisons de simplicité, on va considérer que la plateforme physique ne présente pas des contraintes pour les champs utilisés (ex. : x86, les processeurs réseaux). Donc, les champs utilisés dans la construction d'arbre d'index sont : VLAN, IP de destination et Protocole. Puisque la taille des générateurs minimaux est toujours 1. Donc, on a besoin d'une seule table de flux pour classifier le paquet. Si la taille des générateurs minimale est supérieur à 1, alors, une optimisation est possible en regroupant des champs sur la même table physique (si les contraintes le permettent) parce qu'on a besoin de nombre minimal des recherches pour identifier le sous-ensemble de la règle. Enfin, le *mapping* des règles sur les tables de flux se fait en fonction des clés pour les différents sous-ensembles des règles. Dans la classification des paquets, les entêtes du paquet seront comparés avec l'arbre des indexes pour trouver le sous ensemble de la règle, après une recherche exhaustive est effectuée sur toutes les règles.

Mise à jour de l'arbre

L'état du réseau change rapidement. De nouveaux flux peuvent circuler sur les réseaux, ou bien de nouveaux tenants peuvent être ajoutés à la volée. Ces changements demandent une maintenance et une mise à jour des entrées dans les tables de flux des *switches*. Dans notre cas, les mises à jour ne doivent pas impacter la compacité de l'arbre d'index en terme de regroupement des règles. Pour assurer l'efficacité des mises à jour et un changement minimal à apporter sur la structure d'index, on propose un algorithme de mise à jour simple et qui ne demande pas une restructuration majeure. Le principe de l'algorithme se base sur : chaque nouvelle règle est placée dans la structure existante, si ce n'est pas possible, une restructuration locale est appliquée. Le cas de base arrive lorsque la règle cor-

respond a la clé d'un index, donc la règle sera assignée à ce sous-ensemble. Si la taille de ce sous-ensemble ne dépasse pas la limite d'un nombre d'entrées permis par sous-ensemble (c-à-d le seuil de construction du treillis). A ce moment là, une insertion simple est effectuée. Dans le cas contraire, l'arbre équilibre le nombre d'entrées en divisant le sous-ensemble sur deux. Finalement, si la règle ne correspond à aucun nœuds feuille, alors une branche se crée et contient seulement la règle mise à jour. Le nouveau nœud a comme clé un des *matchfields* de la règle. L'algorithme utilise une approche récursive pour mettre à jour la structure de recherche. La procédure *UpdateIndex(node, rule)* est invoquée en premier sur la racine de l'arbre. Après l'algorithme traverse l'arbre verticalement du haut en bas en cherchant le meilleur nœud pour placer la règle, la vérification au niveau de chaque nœud utilise la primitive $match(rule, key)$ pour choisir la branche. Cette primitive vérifie si une règle $rule$ est définie aussi par key. Le premier cas de mise à jour arrive quand les appels récursifs se terminent sur une feuille. Dans ce cas, la règle est ajoutée aux sous-ensembles des règles liés à ce nœud. Si la taille ne dépasse pas le nombre maximal des nombres par sous-ensemble, alors on arrête le traitement. Sinon, la routine $BuildIndex()$ est appelée avec le sous ensemble des règles comme paramètres. Cette routine va construire un arbre qui sera à la fin relié avec l'arbre principal. Seulement les règles de sous ensemble du nœud seront touchées par le changement. Cette restructuration locale garde les caractéristiques de la structure de recherche. Dans le cas où la recherche renverrait un nœud qui n'est pas feuille. Alors, un nœud isolé (*outlier*) serait créé ($newNode()$), ce nœud contient seulement la règle $rule$ et a comme clé un de ses *matchfields* ($getNewKey()$).

Comme une illustration de la mise à jour. On considère trois règles. (1) La règle $R_4^2 = \{tcp\}$ qui appartient au tenant 2, Cette règle est plus générale que les règles installées, donc son insertion directe causera une inconsistance dans le réseau. Le

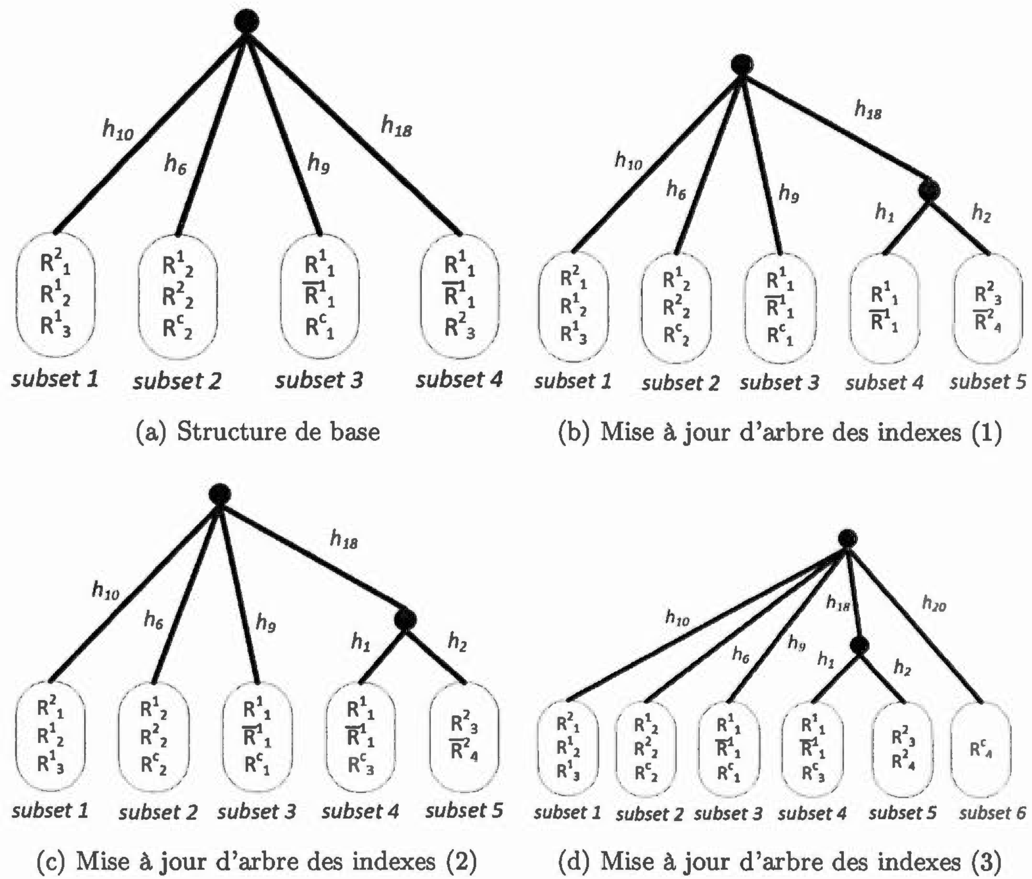
Algorithm 2: Algorithme de mise à jour

```
Procedure UpdateIndex(n : node, r : rule)
    if n.isLeaf() then
        n.getSubset().add(r)
        if n.getSubset().size > max then
            n.setSucc(BuildIndex(n.getSubset()))
    else
        for nextNode ∈ n.getSucc() do
            if match(r, nextNode.getKey()) then
                UpdateIndex(nextNode, r)
                return
        n.succ(newNode(r), n.getNewKey(r))
```

compilateur ajoute d'abord des différenciateurs. La règle R_4^2 se transforme alors en $\overline{R}_1^1 = \{tcp, SIP = 172.168.2.0/24\}$ qui correspond à la clé du sous-ensemble 5 (voir Figure 2.5b). (2) $R_3^c = \{h_1, h_{18}, Vlan = 500\}$ présente le cas le plus simple, en effet R_3^c correspond à la clé de sous ensemble 4 (voir Figure 2.5b) donc il sera ajouté directement dans le sous-ensemble. Enfin, la règle $R_4^c = \{h_{20} = \{VLAN = 100, EtherType = 0x0800, InPort = 20\}$ ne partage aucun *matchfield* avec la structure de recherche, ainsi on crée un nœud isolé (*outlier*) et dont le sous-ensemble correspondant contient seulement la règle R_4^c (voir Figure 2.5d)

2.2.5 Validation de traitement du paquet

La modélisation de traitement du paquet détermine une structure de classification optimale et décrit les règles à installer sur les tables de la *switch*. Avant de mapper cette structure sur l'architecture matérielle des nœuds. Une validation du schéma de classification par rapport aux contraintes matérielles des *switches* est néces-



(a) Structure de base

(b) Mise à jour d'arbre des indexes (1)

(c) Mise à jour d'arbre des indexes (2)

(d) Mise à jour d'arbre des indexes (3)

Figure 2.5: Modélisation de transmission de paquet

saire. La validation permet d'adapter le pipeline aux exigences de traitement de paquet. Pour cela, on a implémenté un module basé sur la théorie de network calculus. Ce module évalue la latence des paquets suivant le modèle de classification proposé. Il utilise la capacité des tables de chaque *switch* et les courbes d'arrivées de trafic qu'on détermine par des mesures de trafic (Thiele *et al.*, 2002a). On définit f par l'ensemble de trafic qui subira le même traitement (mêmes opérations pour la classification). On définit $A(t)$ comme le nombre de paquets arrivés dans l'intervalle $[0, t]$ et qui appartient à f. Les bornes minimales et maximales de la courbe d'arrivée satisfont la relation :

$$\alpha^l(t-s) \leqslant A(t) - A(s) \leqslant \alpha^u(t-s), \forall 0 \leqslant s \leqslant t$$

$\alpha^l_f(\triangle)$ définis une borne inférieure sur le nombre de paquets qui appartiennent au flux f et qui puissent arriver pendant toute période de durée $\triangle$. De même, $\alpha^u_f(\triangle)$ définit une borne supérieure sur le nombre de paquets qui appartiennent au flux f et qui puissent arriver pendant toute période de durée $\triangle$. Donc on aura la relation :

$$\alpha^l_f(\triangle) \leqslant \alpha^u_f(\triangle) \text{ et } \alpha^l_f(0) = \alpha^u_f(0) = 0$$

Le *network calculus* permet d'évaluer les courbes de services sur un système avec plusieurs entités inter-connectées. À partir du théorème (voir Théorème 2.1) du *network calculus*, on peut calculer le délai sur une *switch* par une convolution des courbes d'arrivées de trafic et des courbes de services de ses tables. Cette convolution établit une plateforme générale pour l'analyse de l'ensemble des tables de la *switch* (voir Figure 2.6).

Théorème 2.1. *Soit T_1 et T_2 deux tables ayant une courbe de service β_1 et β_2. La concaténation des deux tables offre une courbe de service $\beta_1 \otimes \beta_2$*

Dans notre contexte, les paramètres des tables d'une *switch* comme la fréquence, le temps d'accès et le type de recherche sont utilisés pour déterminer la courbe de service de chaque table. Pour estimer la capacité de la *switch* pour un flux de trafic, on a besoin des bornes sur le délai de traitement de paquet et des bornes sur les capacités de traitements des différentes ressources. On peut définir une borne supérieur sur le délai de traitement d'un flux f par l'inégalité suivante :

$$delay \leqslant \sup_{u \geqslant 0}\{inf\{\tau \geqslant 0 : \alpha^u(u) \leqslant \beta^l(u+\tau)\}\}$$

On peut définir aussi une borne supérieure sur la capacité d'une table. Cette borne reflète la capacité requise par une table pour traiter les flux :

$$capacity \leqslant \sup_{u \geqslant 0}\{\alpha^u(u) - \beta^l(u)\}$$

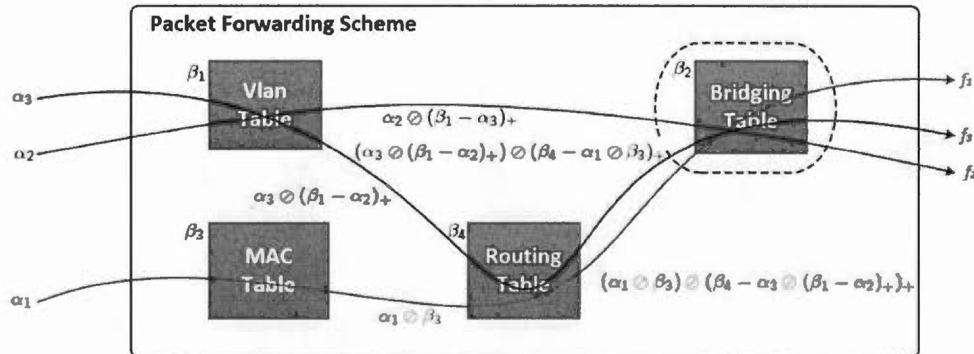


Figure 2.6: Validation de traitement de paquet

Pour analyser l'ensemble des flux afin de trouver la meilleure configuration possible, on utilise l'outil DISCO (Schmitt et Zdarsky, 2006). L'outil permet de calculer la courbe du service de bout-en-bout et la charge (*backlog*) sur chaque nœud et il permet aussi de faire une analyse sur chaque flux en particulier ou bien sur tous les flux. Le résultat de cette analyse indique les ressources nécessaires en terme de capacité au niveau de chaque table.

2.2.6 Partage de ressources

Dans le réseau, le trafic et la charge de chaque tenant changent au fil du temps. Cependant, la capacité de traitement du réseau est constante. On doit ainsi gérer les ressources matérielles pour répondre aux besoins des différents tenants tout en tenant compte des limites physiques. L'approche de partage des ressources résoud le problème de comment allouer les ressources partagées équitablement et efficacement. Par exemple, si on compile les règles d'un tenant par rapport aux autres tenants, il est nécessaire de savoir comment allouer les ressources à ce tenant suivant ses besoins ainsi que les ressources disponibles. Dans ce contexte, les *switches* et les tables de flux sont considérées comme les ressources que les tenants

peuvent partager. Pour résoudre le problème, on modélise le problème suivant la théorie des jeux. On considère les tenants comme des joueurs en compétition pour un ensemble des ressources (ex. : table de flux). La théorie des jeux donne à chaque joueur une stratégie afin de maximiser ses propres intérêts. Notre système

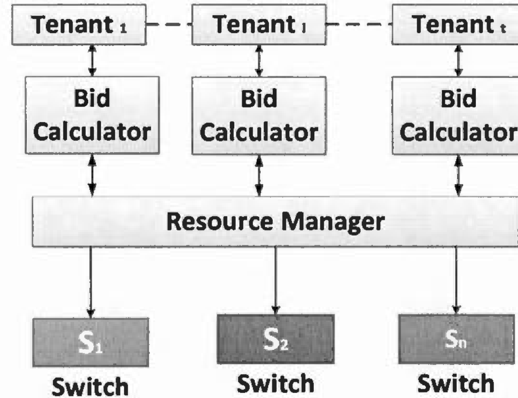


Figure 2.7: Plan d'allocation des ressources

se modélise par un jeu non coopératif (ex. : chaque joueur cherche seulement ses propres intérêts). On considère un mécanisme d'anticipation du prix (*price-anticaption*) où chaque tenant enchère (*bid*) sur chaque ressource séparément et reçoit le ratio de la valeur de son *bid* sur l'ensemble des *bids* des autres tenants sur cette ressource (voir Figure 2.7). Cette approche d'allocation proportionnelle est utilisée pour allouer une capacité de traitement de paquet d'une *switch* à un ensemble de flux d'un tenant. Chaque tenant a un budget fixe, On note le budget d'un tenant i par Θ_i et $\Phi_i = (\Phi_{i,1}, \Phi_{i,2}, ..., \Phi_{i,s})$ est son vecteur *bid*, s est le nombre des ressources pour lesquelles un tenant *bid*. Puisque le budget est fixe donc on a $\sum_{j=1}^{j=s} \Phi_{i,j} = \Theta_i$. L'allocation est proportionnelle, donc l'allocation de la ressource j pour le tenant i est $g_{i,j} = \frac{\Phi_{i,j}}{P_j}$ où $P_j = \Sigma_{k=1}^{k=t} \Phi_(k, j)$ et t le nombre des tenants. Le bid d'un tenant sur un nombre des entrées dans une *switch* est calculé suivant l'algorithme de *best-response*. Pour calculer son bid, un

tenant a une fonction d'utilité U_i qui reflète le bénéfice d'un tenant en utilisant sa stratégie du bid. Les tenants ont des intérêts différents pour chaque *switch* (c-à-d puissance, latence, coût). Le vecteur $\varphi_i = (\varphi_{i,1}, ..., \varphi_{i,s})$ représente la préférence d'un tenant i. Sans perte de généralité, on utilise la fonction linéaire $U_i = \varphi_{i,1} \times g_{i,1} + ... + \varphi_{i,s} \times g_{i,s}$. Le calcul de la stratégie se fait à l'aide de best reponse 3 (Feldman *et al.*, 2005). Le tenant i calcule sa stratégie comme la meilleure réponse aux stratégies des autres tenants. On représente le *bid* total des autres tenants par $y_1, ..., y_s$. La sortie de l'algorithme est le vecteur du *bid* du tenant i : $\Psi_i = \{\psi_{i,1}, \psi_{i,2}..., \psi_{i,s}\}$.

Algorithm 3: Algorithme de Best response

Maximiser $U = \sum_{j=1}^{j=s} \varphi_{i,j} \times \frac{g_{i,j}}{g_{i,j}+y_j}$

En respectant $\Sigma_{j=1}^{j=s}\Phi_{i,j} = \Phi_i$ $\quad \triangleright\ \forall i \in [1,t], j \in [1,s] \Phi_{i,j} \geq 0$

D'abord, on ordonnance $\frac{\varphi_{i,j}}{y_j}$

On suppose que $\frac{\varphi_{i,1}}{y_1} \geq \frac{\varphi_{i,2}}{y_2} \geq .... \geq \frac{\varphi_{i,s}}{y_s}$

Calculer le plus grand k tel que

$\frac{\sqrt{\varphi_{i,k} y_k}}{\Sigma_{j=1}^{j=s}\sqrt{\varphi_{i,j} y_j}} \times (\Theta_i + \Sigma_{j=1}^{j=N} y_j) - y_k \geq 0$

for $j = 0$ *to* N **do**

  if $j > k$ **then**

    $\psi_{i,j} = 0$

    $\psi_{i,j} = \frac{\sqrt{\varphi_{i,j} y_j}}{\Sigma_{j=1}^{j=s}\sqrt{\varphi_{i,j} y_j}} \times (\Theta_i + \Sigma_{j=1}^{j=s} y_j) - y_k$

CHAPITRE III

IMPLÉMENTATION

Dans l'évaluation de notre compilateur, nous nous intéressons à deux architectures matérielles différentes : le processeur réseau nfp-3200 de Netronome [1] et MPPA-256 de Kalray [2]. Nous avons implémenté le plan de donnée sur les deux architectures afin d'évaluer le gain de la compilation dans les deux cas. Dans ce chapitre, nous allons d'abord présenter les deux différentes plateformes ainsi que leurs modèles de programmation.

3.1 Le processeur réseau nfp-3200

Un processeur réseau est un microprocesseur programmable optimisé pour le traitement des paquets. Il est conçu pour gérer les opérations associées aux 7 couches du modèle OSI (Stallings, 1987). Ces opérations sont principalement l'extraction des champs d'un paquet, la manipulation des bits de ces champs, les recherches dans les structures matérielles et logicielles, la modification du paquet, et finalement la transmission vers le réseau. Dans les modèles de traitement des paquets, chaque paquet se traite indépendamment des autres, ce qui permet de paralléliser les chemins du traitement. Les performances des processeurs réseau peuvent aller

1. www.netronome.com

2. www.kalrayinc.com

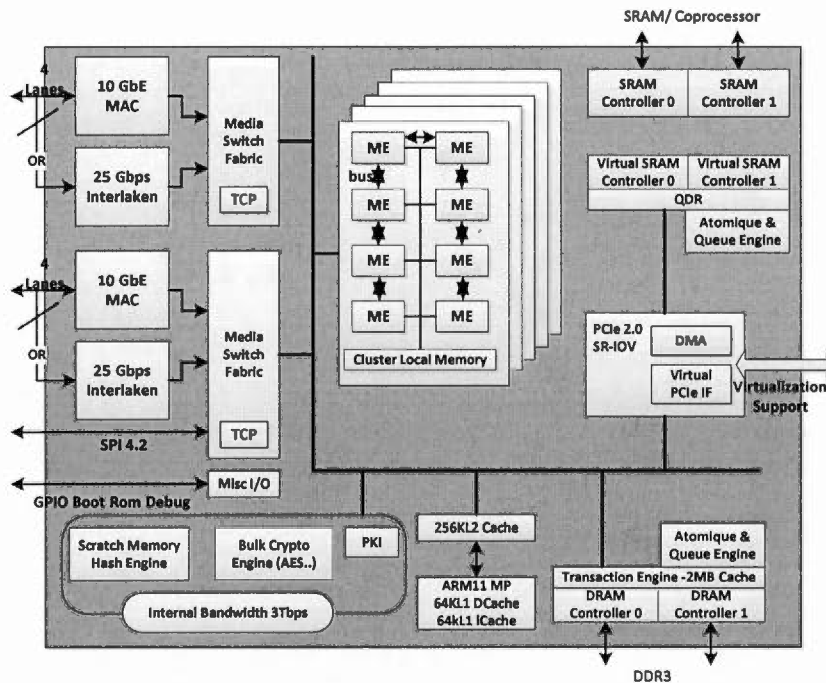


Figure 3.1: Le processeur réseau nfp-3200

jusqu'à 400 Gbps.

3.1.1 Architecture processeur réseau nfp-3200

Le processeur réseau nfp-3200 est un processeur de *20Gbps* qui a 40 MEs (*Micro Engine*) de traitement des paquets distribués sur 5 *clusters*. L'ensemble des MEs offre 2000 instructions et 50 opérations sur les flux à raison de 30 millions de paquets par seconde.

Une ME supporte un contrôle logiciel de 8 fils d'éxécution *threads*. Lors de l'exécution, un *thread* se bloque souvent en attendant la terminaison des opérations de lecture et d'écriture au niveau des mémoires (ex. : SRAM et TCAM), donc avoir plusieurs *threads* permet aux *threads* d'intercaler les opérations. Il existe toujours

un *thread* qui est prête pour s'exécuter. La ME a une mémoire RAM qui enregistre les instructions des programmes, elle supporte 8192 lignes de code, cette mémoire s'initialise extérieurement par l'interface PCIe ou bien par le processeur ARM11. Pour doubler le nombre d'instructions qu'une ME peut enregistrer, il est possible de partager la mémoire d'instruction de deux MEs voisines à condition qu'elles exécutent le même programme. Aussi, une ME a 8 contextes d'exécution, chaque contexte a ses propres registres, si un programme crée un registre, alors 8 copies de ce registre seront créées, chacune dans un contexte. Chaque contexte est associé avec un seul *thread*, donc, changer le *thread* en exécution ne nécessite pas une copie des données d'une mémoire partagée aux registres. Les MEs sont toutes connectées avec un système de gestion des événements, ce système envoie des signaux aux MEs pour les informer de l'état des opérations. Par exemple, lorsqu'un thread demande l'écriture dans une mémoire externe, elle doit attendre que le système des gestions des événements lui indique que l'opération d'écriture est terminée avant qu'elle puisse continuer l'exécution. La communication entre les MEs se fait à travers des *rings*, un ring se construit en allouant la mémoire sur le *Global scratch* (un type de mémoire). L'échange de donnée sur le ring s'effectue par un mécanisme de queue FIFO (premier entré, premier sorti).

Le nfp-3200 dispose de plusieurs types de mémoires et d'accélérateurs. Pour les mémoires, le processeur contient une SRAM d'une capacité de 32Mbp et une DRAM de 4Gbp, la SRAM a un temps d'accès moins élevé que celui de la DRAM. Elle est utile pour implémenter des compteurs et des structures de recherche. La DRAM est utilisée pour enregistrer les paquets reçus du réseau. Elle peut être utilisée aussi pour enregistrer des compteurs ou bien pour implémenter des structures de recherche logicielles, mais son temps d'accès élevé va limiter les bénéfices de ces utilisations. Les accélérateurs matériels sont des coprocesseurs ASIC qui se caractérisent par une rapidité d'exécution. Le bloc de cryptographie offre 5 algo-

rithmes différents (AES, 3DES, ARC4, SHA-1, MD5) qui ont respectivement des performances de (6 ; 2.6 ; 1.8 ; 4.3 ; 2.5)Gbs.

Les accélérateurs qui nous intéressent beaucoup sont la TCAM (Ternary Content Adressable Memory) et le HASH. La TCAM exécute des recherches en $O(1)$, la TCAM a une capacité de $0x7FFFF$ entrées. La taille de la clé peut être 72 bits, 144 bits, 288 bits et 576 bits. Le bloc de hachage permet de hacher des clés d'une longueur variable avec un temps de $O(1)$, le hachage peut être utilisé comme structure de recherche ou bien pour développer des applications de *load balancing, firewall...*

Pour recevoir et envoyer le trafic du réseau, le nfp-3200 utilise deux MSF (*Media Switch Fabric*). Les deux MSF connectent les entrées sorties du réseau au bus CPP du processeur. Dans la direction *ingress* (ex. : du réseau ver le NPU), les paquets qui arrivent sont enregistrés sur la MSF RBUF (Received Buffer). Dans la direction *egress* (c-à-d du NPU vers le réseau) les paquets sont enregistrés sur la TBUF (Transmission Buffer) ou bien la DRAM. Ensuite, la MSF les place dans le réseau.

3.1.2 Programmation du processeur nfp-3200

Le nfp-3200 a deux modes de programmation, le premier est un langage assembleur prioritaire, le deuxième est une variété du langage C appelé *microC*.

- **Programmation par assembleur** : L'assembleur offre un niveau bas de contrôle et de programmation, le code offre ainsi un grand niveau de flexibilité en terme de programmation et de gestion des ressources. Il est possible de contrôler les *threads* et d'ajouter plus de granularité le traitement des paquets. En général, le code assembleur offre une meilleure performance, mais il n'est pas portable et nécessite souvent plus de temps pour le maîtriser et pour développer les applications.

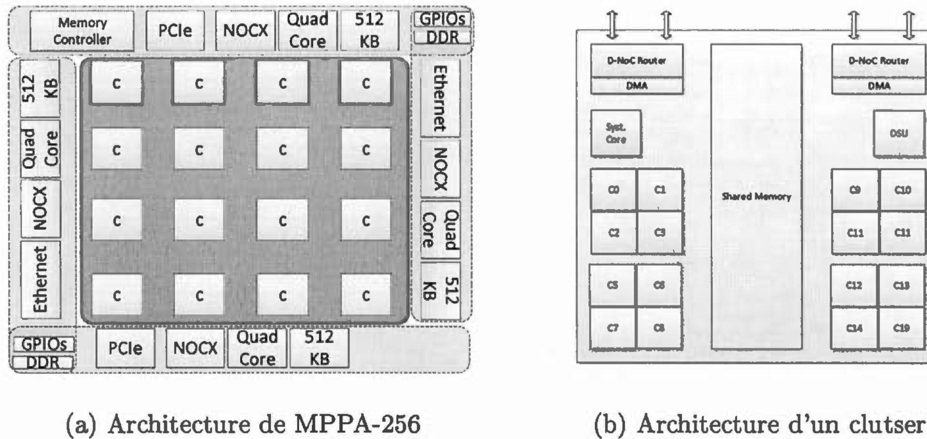
- **Programmation par *MicroC*** : Il est possible d'utiliser un langage qui s'approche de C pour programmer le processeur réseau. Cela accélère le cycle du développement des applications et simplifie la portabilité du code vers d'autres plateformes. Cependant, ce mode ne donne pas les meilleures performances comme un code assembleur.

3.2 Le processeur multicoeur MPPA-256

3.2.1 Architecture du MPPA-256

Le processeur MPPA-256 se compose de 16 *clusters* et de 4 sous-systèmes d'entrées sorties (IOs) (voir Figure 3.2a). Les IOS sont connectés par un NoC (*Network On Chip*). Le NoC a deux réseaux parallèles : le premier est optimisé pour le transfert des données avec une bande passante élevée, le deuxième est pour le transfert des messages de contrôle. Il transfère des messages courts rapidement. Les 16 *clusters* sont regroupés dans des ensembles de 4 *clusters*. Un *cluster* (voir Figure 3.2b) est un multi-core qui a une mémoire SRAM partagée de 2MB. Le cluster a 17 *cores*, 16 entre eux sont des *cores* de traitements de données, ils sont appelés PEs (*Processing Element*). Le dernier *core* est utilisé pour le contrôle et le management, il est appelé RM (*Resources Manager*). Chaque *core* a sa propre mémoire cache pour enregistrer les données et les instructions, la cache a une capacité de 8KB.

Chacun des IOS se compose de 4 interfaces réseau et 4 core RM placés suivant une configuration multiprocesseur symétrique et qui sont connectés à une mémoire de 512 kB. Chaque RM a sa propre mémoire de 32 kB et partage avec les autres RM une cache de 128 kb, le partage de la cache assure une cohérence de données entre les RMs. Pour les interfaces réseau, deux sont connectées avec les mémoires DDR (ex. : Double Data Rate) et deux sont connectées a des interfaces Ethernet de 10 Gb/s.



(a) Architecture de MPPA-256

(b) Architecture d'un clutser

Figure 3.2: Le processeur Kalray MPPA-256

3.2.2 Programmation du processeur MPPA-256

Le MPPA-256 a deux modes de programmation, une programmation orientée donnée (*dataflow programming*) et une programmation POSIX où les *threads* sont réparties sur les différents cores pour une exécution rapide.

- **Programmation Dataflow** : Dans ce modèle, les programmeurs ne gèrent pas la synchronisation des informations entre les cores. Les cores sont connectés suivant des canaux bien définis, les conflits au niveau des variables partagées ne sont pas possible. Pour travailler dans ce mode, il faut utiliser un langage appelé SigmaC (ΣC), ce langage hérite des bibliothèques standard du langage C. Il assure une portabilité du code entre les différentes architectures offertes par kalray. En plus, le compilateur utilise des heuristiques pour adapter le code aux caractéristiques de l'architecture d'exécution (ex. : nombres des cores, mémoire disponible).
- **Programmation POSIX** : Dans ce modèle, les processus sur les systèmes d'opérateurs créent des sous-processus sur les autres *cores*. L'utilisation du compilateur *GNU GCC* permet d'étendre les possibilités d'un programme à

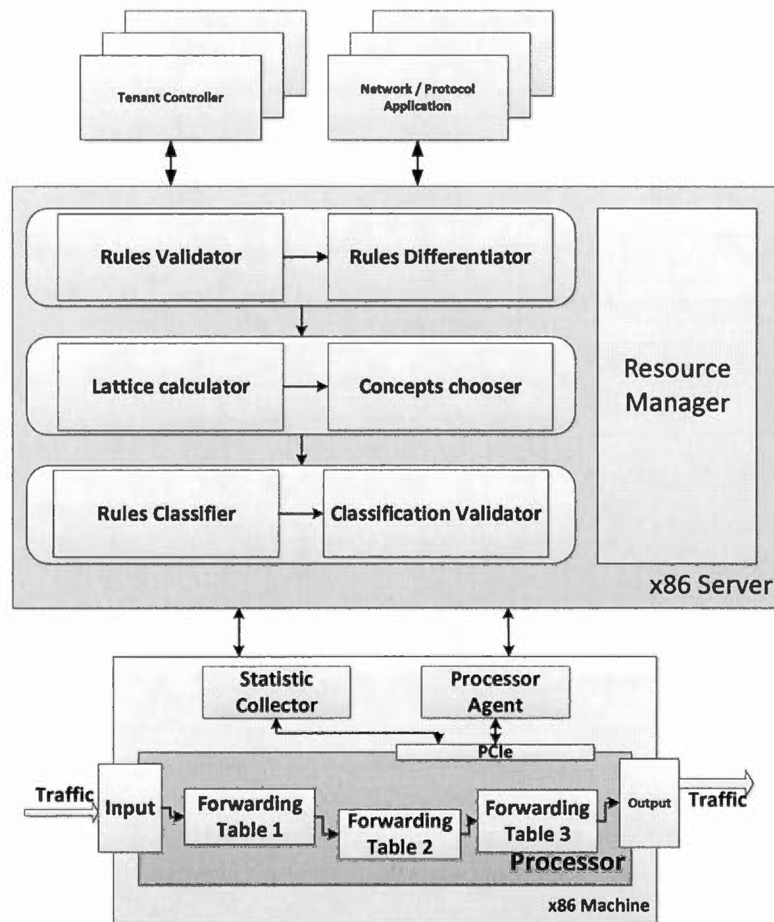


Figure 3.3: Implémentation du compilateur

s'exécuter sur plusieurs cores à la fois. Ce modèle se différencie par rapport aux POSIX classiques seulement par le mécanisme de communication entre les cores, dans un POSIX classique, la communication peut se faire avec IPC (*Inter Process Communication*), ici, il faut utiliser des fichiers spéciaux gérés par le NoC.

3.3 Implémentation du compilateur

Pour l'implémentation, nous avons utilisé 2 machines physiques différentes : la première est connectée avec le processeur (MPPA-256 ou nfp-3200), la deuxième héberge le compilateur. Les contrôleurs sont des applications qui roulent sur des machines virtuelles. Ces machines virtuelles roulent sur le même serveur du compilateur. Les contrôleurs sont connectés au compilateur suivant un modèle client/serveur. Les messages échangés entre les contrôleurs et le compilateur respectent une forme standard bien définie. Cela ne limite pas ce que le compilateur peut supporter.

Comme expliqué dans le chapitre de design, le compilateur se compose de plusieurs entités qui interagissent entre elles. Le *rules validator* joue le rôle d'interface avec les contrôleurs, il reçoit les règles et vérifie s'il y'a un chevauchement entre les règles. S'il en trouve, il va envoyer les règles vers le module de *Rules Differentiator* pour les étendre, si aucun conflit n'est détecté il va envoyer la règle vers le *Rule Miner*. Le *Rule Miner* est le bloc le plus intéressant dans l'implémentation, d'abord il construit le contexte global, après il calcule le treillis. Le *Rules Classifier* construit la structure de recherche à partir des concepts choisis par le *Rule Miner*. Le *Classification Validator* supervise les performances de la structure de classification en terme de compacité et de distribution des règles. Aussi, il exécute un module basé sur le Network Calculus pour détecter des blocages au niveau du processeur réseau. Le *Classification Validator* peut déclencher une reconstruction globale de structure de classification si les performances se dégradent considérablement.

Le processeur réseau est branché avec une machine hôte par une interface PCIe. La machine exécute un programme pour les statistiques et un autre pour le contrôle du processeur. Le *Static Collector* récupère du processeur les statistiques du plan

de donnée, il utilise des compteurs pour calculer le taux des paquets traités et le taux d'utilisation des tables physiques, ces données seront utilisées par le *Classification Validator*. Le *Processor Agent* configure les tables du processeur, effectue des opérations d'insertion et de suppression des entrées. Le *Processor Agent* et le Static Collector dépendent de chaque processeur. Donc pour que notre compilateur supporte un nouveau processeur, il suffit de développer ces deux modules.

CHAPITRE IV

ÉVALUATION

Dans le chapitre d'implémentation, nous avons présenté les architectures physiques utilisées et leurs modèles du traitements de paquets. Dans ce chapitre, nous allons nous concentrer sur l'évaluation du compilateur et du plan de données pour montrer l'intérêt de la compilation des règles sur les réseaux multi tenants. À travers cette évaluation, nous voulons déterminer les avantages et les inconvénients de la solution et trouver les configurations optimales des différents paramètres pour une meilleure performance. L'évaluation sera divisée principalement en deux parties, la première partie consiste à évaluer les performances de compilation qui sont liées seulement au compilateur, alors que la deuxième partie traitera les performances du plan de données.

4.1 Paramètres de banc de test

Pour l'évaluation, nous avons utilisé le programme d'acheminement des paquets (voir Figure 4.1) décrit dans (Corp., 2014). Ce programme est utilisé dans les centres de données pour répondre aux exigences de la bande passante et latence. Le programme supporte 3 types d'acheminement : 1) Traitement L2 2) L3 *unicast* 3) L3 *multicast*. Les deux processeurs choisis pour l'évaluation supportent plusieurs configurations pour les tables physiques. On a choisi une configuration qui est mieux adaptée à notre contexte d'évaluation. Pour le nfp-3200, nous avons

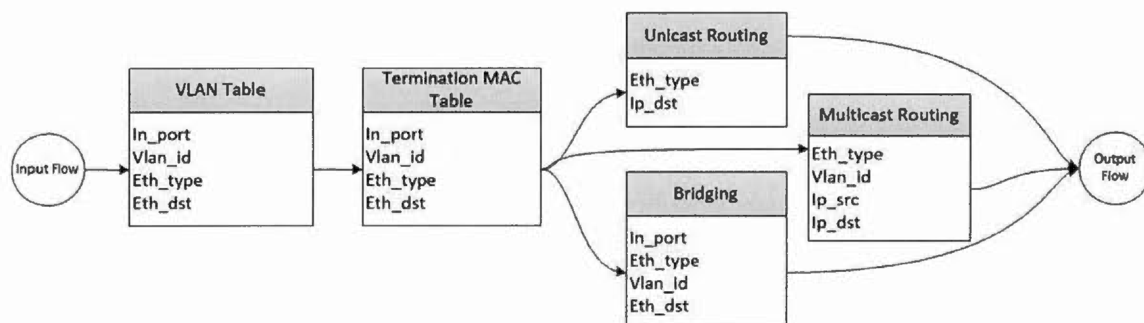


Figure 4.1: Pipeline d'acheminement de trafic

Type de recherche	Type de mémoire	Clé (byte)	Taille (entrées)
Exact	Hash	16	262144
Exact	Hash	48	159288
Ternary	TCAM	16	65536
Ternary	TCAM	32	32768
Ternary	TCAM	64	32768
Exact	SRAM	4	16384

Tableau 4.1: Configuration de NFP-3200

adopté la configuration (voir Tableau 4.1). Pour mppa-256, nous avons choisi la configuration (voir Tableau 4.2).

Pour récupérer les performances de plan de données, nous avons utilisé un générateur de trafic nommé Blaster (Netronome, 2013). Le Blaster se connecte avec le processeur réseau suivant l'architecture (voir Figure 4.2)

Le Blaster se connecte aux machines à travers une interface PCIe. Le Blaster est pourvu d'un logiciel qui permet de le paramétrer. Il peut spécifier le profil du trafic, créer des fichiers (.pcap), modifier la distribution du trafic, spécifier le

Search type	Algorithm	Key (byte)	Size (entries)
Exact	Hash	11	2048
Prefix	Longest Prefix Match	4	1024
Exact	Hash	31	2048

Tableau 4.2: Configuration de MPPA-256

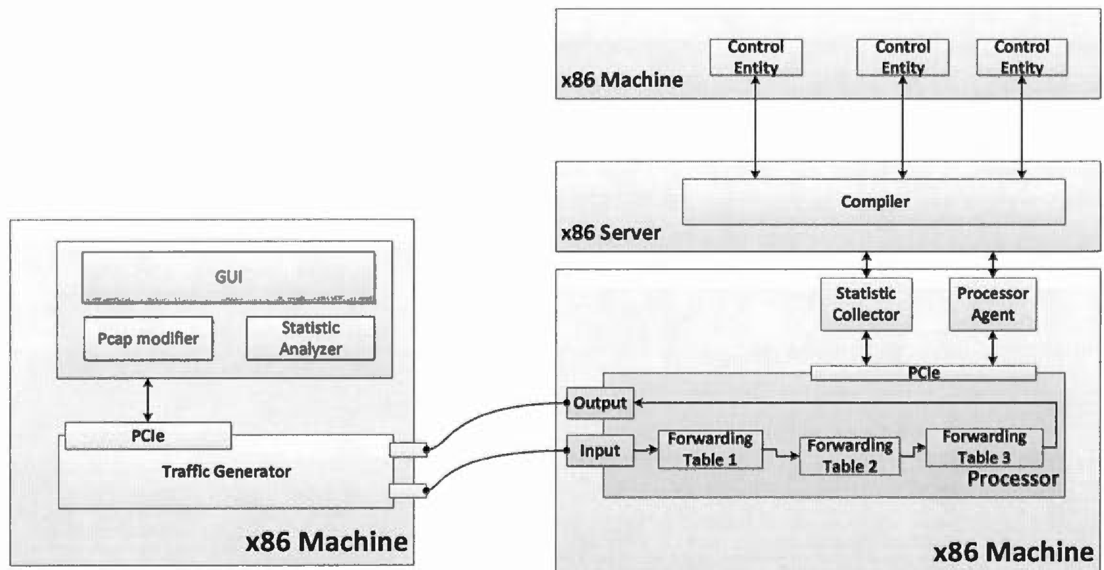


Figure 4.2: Banc de test

Motif de trafic	Flux total	Débit (Mbps)
1	80	51
2	160	103
3	240	154
4	3000	549
5	4000	961

Tableau 4.3: Motif de trafic

nombre de flux/seconde et évidemment désigner le port de sortie du trafic. Le logiciel prépare les nouveaux fichiers pcap pour refléter les changements et les caractéristiques indiqués par l'utilisateur. Le logiciel passe après ces fichiers pcap au Blaster à travers l'interface PCIe. Le Blaster a deux interfaces Ethernet de 10Gbps et permet de saturer un lien de 10Gbps. Pour les mesures, le Blaster calcule le taux de trafic perdu en divisant le nombre de paquets reçus par le nombre de paquets envoyés. Aussi, il calcule la latence et le maximum de la bande passante supportée. En utilisant le Blaster nous avons spécifié le motif du trafic décrit dans le tableau (voir Tableau 4.3).

4.2 Performance de compilation

Les performances de compilation dépendent de plusieurs paramètres. Ainsi, notre objectif sera de trouver les paramètres optimaux pour la compilation. On va varier le nombre de règles pendant la première phase de compilation et pendant la phase de mise à jour. On va aussi varier le seuil de fouille des règles.

4.2.1 Capacité du compilateur

Le compilateur compile les règles et crée une structure de classification, il effectue des mises à jour sur cette structure. L'opération de mise à jour est essentielle dans le réseau et doit être la plus rapide possible. Nous avons observé et évalué le coût des opérations de compilation et de mise à jour afin de déterminer quand une reconstruction de la structure de recherche est mieux que la mise à jour.

On a calculé le temps de compilation de 10000 règles pour des contextes multi tenants et mono-tenant. Le temps de la compilation est le même pour les deux contextes (voir Figure 4.3). Ce temps dépend fortement de l'implémentation du compilateur et la puissance de la machine qui l'exécute, l'utilisation des meilleurs processeurs (ex. : Intel Xeon) permettra d'améliorer les performances. On remarque que le temps de mise à jour croît avec le nombre des règles à mettre à jour. Pour 100 à 200 règles, le temps est de 1 à 1.5 seconde. Ce temps croît jusqu'à 3 secondes pour 1100 règles pour un contexte multi-tenant et 1900 pour un contexte mono-tenant. Donc 1900 et 1100 représentent la limite où une mise à jour est meilleure qu'une reconstruction globale. Si le nombre de mises à jour dépasse cette limite, alors c'est mieux de procéder avec une reconstruction globale. En effet, les mises à jour affectent la compacité et l'équilibre de la structure de classification, donc une reconstruction permettra de rééquilibrer la structure de classification.

Le temps de mise à jour dans le contexte de multi tenants est supérieur à celui de mono-tenant. En effet, l'opération de mise à jour des règles dans un contexte multi tenants nécessite la validation des règles par rapport aux définitions des tenants. En plus, elle demande une modification et extension des règles dans le cas de conflit. Ces opérations ajoutent une surcharge qu'on appelle surcharge de virtualisation (voir Figure 4.4).

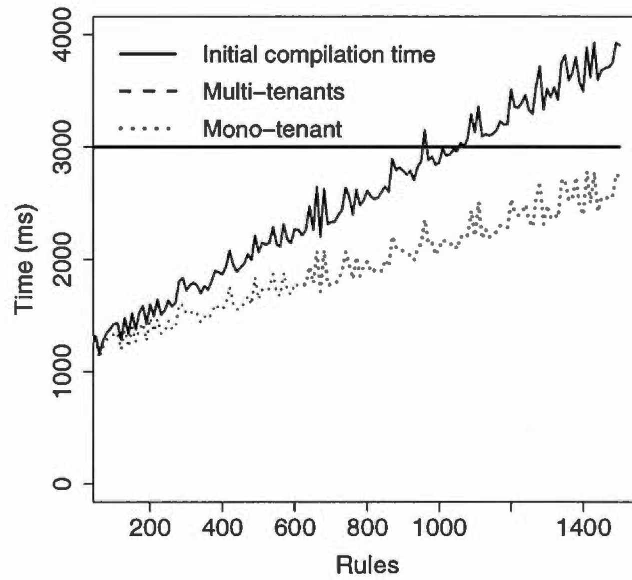


Figure 4.3: Temps de mise à jour

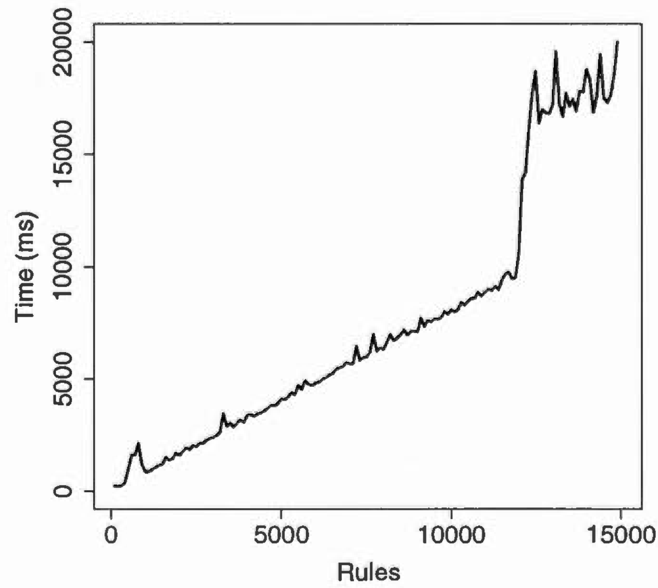


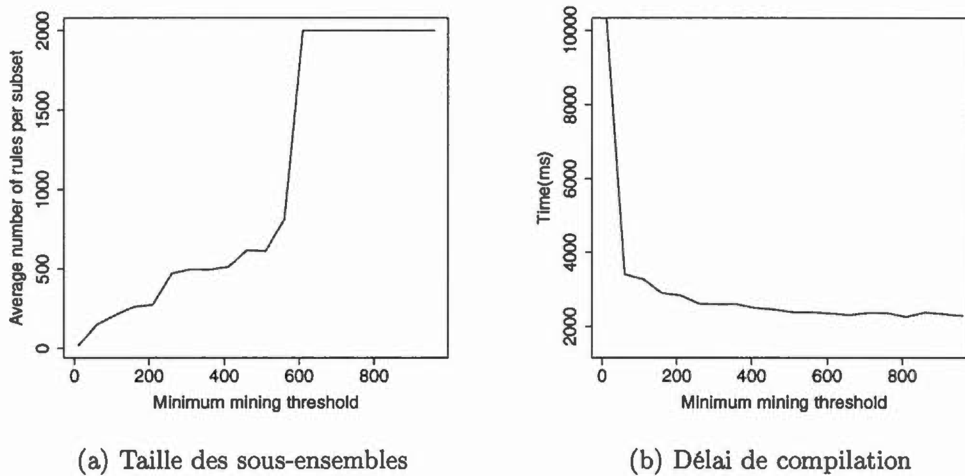
Figure 4.4: Surchage de la virtualisation

Un autre paramètre qui influence la performance de la compilation est le seuil utilisé dans la fouille des règles. Le suil est utilisé pour déterminer le nombre des règles dans chaque sous-ensemble. Pour évaluer son effet sur le temps de compilation, nous avons fixé le nombre des règles à 10000 règles et nous avons varié la valeur du seuil (voir Figure 4.5b)

On remarque que le fait d'augmenter le seuil diminue le temps de compilation de 3.5 secondes à 2.4 secondes. En effet, un seuil élevé permet au compilateur de calculer moins de concepts et ainsi générer moins de sous-ensembles, le compilateur utilise un nombre minimal des *matchfields* pour séparer les règles en des sous-ensembles. Par conséquent, augmenter le seuil induit une structure de classification qui se caractérise par un nombre limité des index. Par contre, le nombre des règles par sous-ensemble augmente, ce qui limite le gain de la structure de classification (voir Figure 4.5a). Le choix du seuil présente un compromis entre le temps de compilation et la distribution des règles sur les sous-ensembles. Pour assurer une meilleure performance du compilateur, il faut enregistrer la valeur du seuil à utiliser dans les paramètres du compilateur. Suivant nos résultats, le meilleur rang pour le seuil est $[\frac{N}{30}, \frac{N}{25}]$.

4.2.2 Utilisation des ressources

Un des objectifs principaux de notre compilateur est l'optimisation des ressources au niveau des tables de flux des *switches* du réseau. En prenant les mêmes scénarios que les tests d'avant, on compare les ressources consommées par un programme compilé (ex. : les entrées compilées) avec un programme non compilé. On calcule le nombre d'entrées à installer dans les tables de flux dans les deux cas par rapport au nombre des règles envoyées par les contrôleurs (voir Figure 4.6). On remarque que le compilateur réduit le nombre des entrées à installer de 30% à 50% pour un contexte multi-tenant et mono-tenant respectivement. En effet, indexer des sous-



(a) Taille des sous-ensembles

(b) Délai de compilation

Figure 4.5: Effet de seuil sur la compilation

ensembles des règles par des *matchfields* partagés réduit le nombre des entrées requises au niveau des tables de flux. En plus, il est intéressant de noter que si les *switches* ont des tables de flux larges (ex. : en terme de longueur de clé), une meilleure optimisation de la structure de classification est possible, en effet des tables de flux larges permettent de combiner des *matchfields* dans la même table et ainsi exécuter une seule recherche au lieu d'une séquence de recherche. Le facteur d'optimisation des entrées pour un contexte multi-tenant est moins élevé (voir Figure 4.6). Donc, l'opération de validation et différenciation des règles peut générer plusieurs règles à partir de chaque règle. Cela augmente le nombre de règles dans le contexte global.

4.3 Performance du plan de données

L'évaluation de la performance de plan de données est cruciale dans l'évaluation de performance de notre compilateur. La latence des paquets au niveau d'un équipement est une métrique importante pour évaluer cet équipement. Dans notre cas, on a comparé un programme compilé avec un programme non compilé sur deux

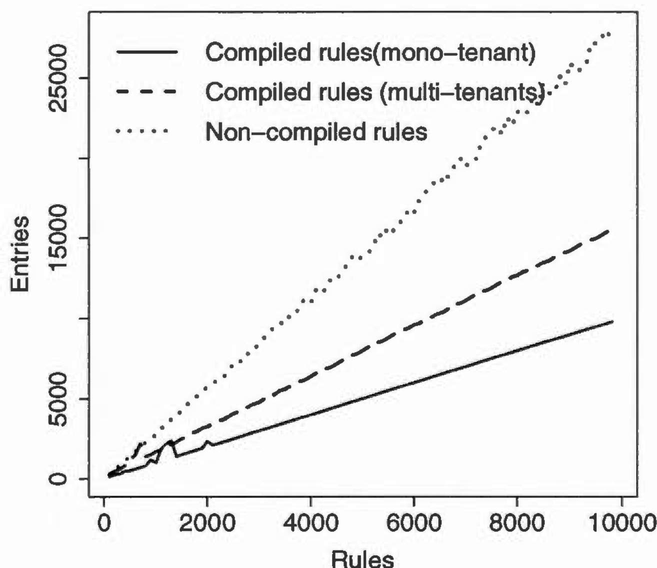


Figure 4.6: Réduction de nombre des entrées

architectures différentes (ex. : switch logicielle et switch matérielle). Les deux architectures présentent des solutions de hautes performances pour la commutation et le routage des paquets dans différents équipements dans le marché. Nous avons utilisé le même motif de trafic pour évaluer les performances de plan de données (voir Tableau 4.3).

Dans le cas d'une architecture matérielle, un programme compilé offre des meilleures performances, la latence est réduite de 30% ce qui revient à 2-4us de moins (voir Figure 4.7). Dans le cas d'une architecture logicielle, un programme compilé a une latence de 50-70% de moins par rapport à un programme non compilé, aussi la réduction de la latence est de 50-100us (voir Figure 4.8).

L'optimisation sur une switch logicielle est plus considérable par rapport à une switche matérielle. Sur une switch matérielle, le traitement de paquet se fait par

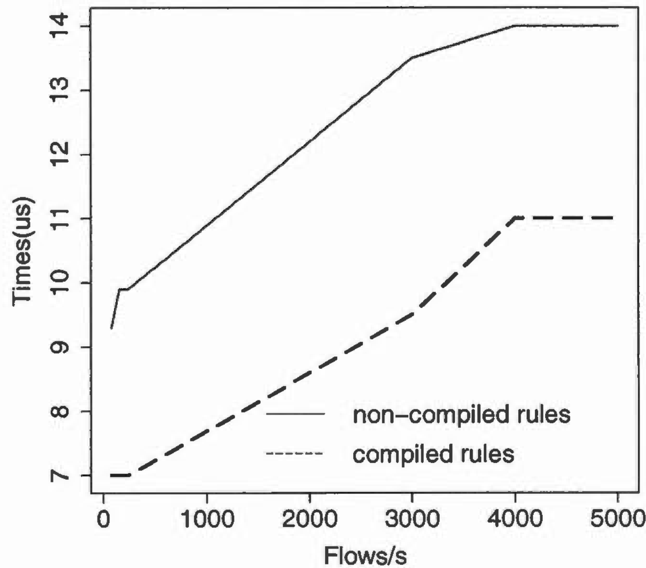


Figure 4.7: Switch matérielle

des accélérateurs matériels qui sont rapides et donc la compilation n'agit pas beaucoup sur la classification des paquets, la compilation du programme améliore les opérations liées aux *parsing* (ex. : extraction des champs) et réorganise l'opération de la recherche. Pour une *switch* logicielle, le compilateur a une grande liberté pour la création des tables et des pipelines, cela permet ainsi d'avoir un gain significatif en terme de latence.

4.4 Équité d'allocation des ressources

Suivant le modèle des enchères qu'on a présenté dans la partie design du compilateur, les tenants enchérissent sur un nombre des règles au niveau de chaque *switch*. Le mécanisme d'allocation des ressources analyse les enchères des tenants et décide du nombre d'entrées à allouer pour chaque tenant. Le tenant détermine la préférence de chaque *switch* suivant ses intérêts (ex. : performance, fonctionnalité,

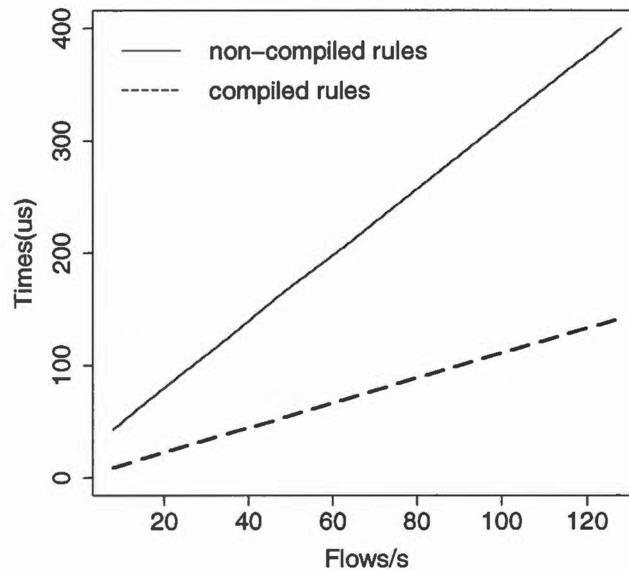


Figure 4.8: Switch logicielle

localisation).

Pour évaluer l'équité en terme d'allocation des ressources, nous avons évalué la fonction d'uniformité d'équité des tenants définie par :

$$\tau = \frac{min_i(U_i)}{max_i(U_i)}$$

L'utilité d'un tenant U_i présente ce que le tenant a reçu à partir de son vecteur de *bid*. Une uniformité d'utilité de 1 est le maximum théorique. Après un changement des motifs de trafic ou bien des applications au niveau des contrôleurs, un contrôleur peut choisir de changer sa stratégie de *bid*. Ces changements affectent l'allocation des ressources et nécessitent un calcul d'une nouvelle allocation. Dans l'évaluation (voir Figure 4.9), nous avons perturbé le système de 50% (Un tenant a changé son *bid* de 50% suite à un changement au niveau de trafic ou des applications) et nous avons évalué le changement d'uniformité d'utilité. On remarque que le système commence à réagir instantanément après la perturbation. L'équi-

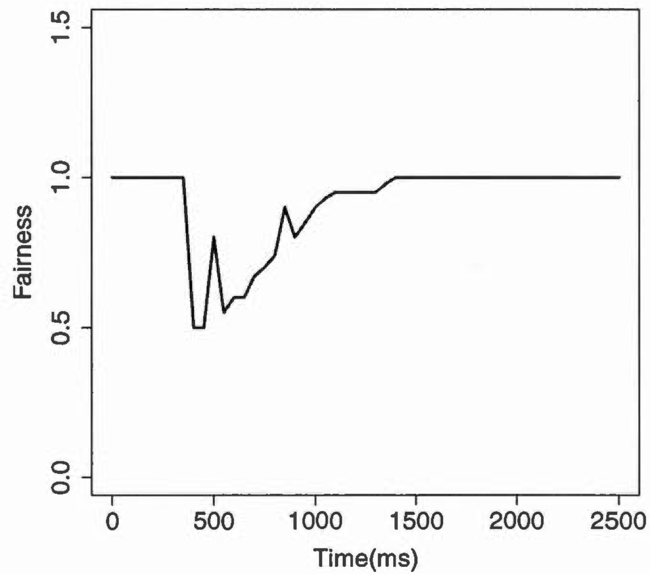


Figure 4.9: Équité d'allocation des ressources

libre est atteint après 1.5s. Vu que ces changements ne sont pas fréquents, alors un délai de 1.5s n'affectera pas la performance du système et permet d'adapter l'allocation des ressources au changement des applications et/ou des contrôleurs.

CONCLUSION

En se basant sur la fouille des règles et les concepts de l'AFC, ainsi que sur la théorie des jeux et le *network calculus*. Nous avons conçu et implémenté un compilateur qui permet aux tenants de gérer efficacement leurs réseaux, tout en assurant un bon fonctionnement de l'architecture physique sous-jacente. Le compilateur propose une stratégie de compilation des règles qui consiste à trouver les relations combinatoires entre les règles et de créer des modèles de traitements optimaux suivant l'architecture physique. Le compilateur utilise aussi le *network calculus* pour évaluer les délais du traitement des paquets sur tout le réseau et sur un nœud particulier pour éviter les blocages. La théorie de jeux est aussi utilisée suivant un modèle d'enchères pour assurer une allocation des ressources équitables entre les différents tenants. Le compilateur dispose de cinq fonctionnalités principales :(1) différencier le trafic des tenants pour garder un bon fonctionnement de tous les réseaux logiques; (2) utiliser une structure de classification flexible qui supporte des mises à jour rapides; (3) minimiser le nombre des règles à installer sur chaque *switch* individuellement; (4) assurer un placement des règles qui élimine le blocage et (5) assurer une équité au niveau de l'allocation des ressources entre les différents tenants. Dans le but d'évaluer les différentes composantes de notre solution, nous avons implémenté le compilateur sur un serveur d'architecture x86. L'implémentation du compilateur dans notre cas est centralisée dans la même machine, mais peut être facilement distribuée sur plusieurs machines pour améliorer les performances et la disponibilité. Le plan de données a été implémenté et testé sur deux architectures différentes. La première est basée sur un processeur réseau de Netronome (nfp-3200), l'autre est basée sur une machine x86. Ces deux

implémentations permettent d'évaluer notre compilateur dans deux contextes différents. Le premier est matériel où il y'a des contraintes au niveau du nombre des entrées supportées dans les tables, ainsi qu'un modèle de traitement rigide. Le deuxième est purement logiciel, il impose moins de contraintes. Cependant, il est limité en terme de puissance du traitement.

Nos expériences ont permis de mettre en évidence les problèmes et les défis relatifs au développement des compilateurs pour les règles. Ces expériences ont montré que notre compilateur réduit le nombre des entrées à installer au niveau de chaque commutateur par 20% à 30% par rapport à des règles non compilées. Aussi, d'après les tests, le compilateur supporte une mise à jour de 300 règles/s. En plus, le délai de transmission des paquets a été réduit par 30% à 50% selon le type de l'équipement. L'optimisation du délai ainsi que le nombre d'entrées montre l'importance de combiner la fouille des règles et le *network calculus* dans une seule entité. L'exploitation des relations combinatoires entre les règles crée une structure de recherche à la fois compacte et rapide. En plus, puisque la structure est indexée, les mises à jour deviennent rapides et faciles à effectuer avec un minimum de changement au niveau de la structure de recherche. Comme toute solution de virtualisation, notre compilateur ajoute une surcharge au niveau du temps de la compilation des entrées par rapport à une solution avec un seul tenant. D'après les résultats, cette surcharge se limite à 10%. Les résultats montrent aussi que la stratégie de gestion des ressources évite le blocage des réseaux et assure une utilité de 0.9 pour les contrôleurs lors de la compilation des règles. L'isolation du trafic introduit aussi une surcharge puisqu'elle demande une analyse des règles pour assurer qu'aucun tenant ne perturbe ni le fonctionnement des autres tenants ni le fonctionnement des services spéciaux. En général, les résultats ont montré la convivialité et la faisabilité d'un compilateur des entrées pour les réseaux multi tenants. Il est important de noter que le compilateur s'exécute sur un serveur

externe dédié, ceci dit, la performance du compilateur peut facilement s'améliorer en allouant plus de puissance de calcul (ex. : des processeurs plus rapides) au serveur central. Aussi, une architecture distribuée est possible pour une meilleure performance.

APPENDICE A

L'ALGORITHME DE CONSTRUCTION DU TREILLIS DES CONCEPTS

A.1 Algorithme Charm

L'espace d'énumération des ensembles fréquents est 2^m. L'éxploration de toutes les possibilités consomme beaucoup de ressource de calcul ce qui rend l'opération lente. L'algorithme se base sur un résultat fondamental pour trouver les ensembles fréquents fermés :
Le support de chaque élément x est supérieur ou égal au support(c-à-d le seuil) de sa fermeture, formellement :

$$\forall H \in \mathcal{H}, \sigma(H) = \sigma(g(f(H)))$$

$\sigma(H)$ est le support de H. Il représente le nombre des règles qui contiennent H dans leurs descripteurs. Charm permet de trouver les FFIs d'une manière efficace. Durant l'exploration des données, Charm parcourt l'ensemble des matchfields et l'ensemble des règles en même temps.

Pour comparer si un élément existe déjà ou non dans l'ensemble des fréquents fermés, on garde une table de hashage des fréquents fermés avec leurs supports.

Algorithm 4: Algorithme Charm

function CHARM($\delta \subseteq I \times T$, *minsup*)

$P = \{h_i \times g(h_i) : h_i \in \mathcal{H} \wedge \sigma(h_i) \geq minsup\}$ Charm-extend(Noeuds=$\oslash$,

$C = \oslash$)

return C

end function

---

procedure CHARM-EXTEND([P], C)

forall $h_i \times g(h_i)$ *dans [P]* **do**

$[P_i] = \oslash$ et $P_i = P \cup h_i$

forall $h_j \times g(h_j) \in [P]$ *et* $j > i$ **do**

$H = H \cup h_j$ et $R = g(h_i) \cap g(h_j)$

$Charm - Property(H \times R, h_i, h_j, P_i, [P_i], [p])$

$C = C \cup P_i$ ▷ si P_i n'est pas inclu

$Charm - extend([P_i], C)$

end procedure

procedure $Charm - Property(H \times R, h_i, h_j, P_i, [P_i], [p])$

if $\sigma(H) > minsup$ **then**

if $g(h_i) = g(h_j)$ **then**

Supprimer h_j de [P]

$P_i = P_i \cup h_j$

else if $g(h_i) \subset g(h_j)$ **then**

$P_i = P_i \cup h_j$

else if $g(h_j) \subset g(h_i)$ **then**

Supprimer X_j des [P]

Ajouter $H \times R$ à $[P_i]$

else if $g(h_i)\, != g(h_j)$ **then**

Ajouter $H \times R$ à $[P_i]$

end procedure

A.2 Trouver les générateurs fréquents

L'algorithme de Talky-G est utilisé pour trouver les générateurs fréquents, il parcourt les données d'une manière inverse (droite à gauche) à Charm et il garde comme résultat les générateurs fréquents.

Algorithm 5: Talky-G pseudocode

```
procedure TALKY-G() racine.element ← ∅ racite.relation ← δ for tous
element do
    ▷ un parcours vertical est obligatoire if (element.support ≥ minsupp) et
    (element.support < M) then
        ▷ M = nombre total des objets racine.AjouterEnfant(element)

    for tous enfant do
        ▷ parcourir droite a gauche enregistrer(enfant) explorer(enfant)
end procedure

procedure EXPLORER(NOEUD) for tous autre do
                                        ▷ voisin de noeud gauche à droite
    generateur ← RecupererProchainGenerateur(noeud, element) if
    generateur ≠ null then
        noeud.ajouterenfant(generateur)

    for tous autre do
        ▷ voisin de noeud droite à gauche enregistrer(autre) explorer(autre)
end procedure


function RECUPERERPROCHAINGENERATEUR(NOEUD,AUTRE)
candidat.relation = noeud.relation ∩ autre.relation if
|candidat.relation| < minsupp then
    return null
    else if (candidat.relation = neoud.relation ) ou (candidat.relation =
autre.relation ) then
    return null
    else
        candidat.element ← noeud.element ∪ autre.element
    if candidat a un sous ensemble avec le meme support dans le table de
hashage then
    return null
return candidat

end function
```

APPENDICE B

LISTE DES CONCEPTS

Id	Intension	Extension	Générateur
C_2	h_1,h_5 h_9,h_{18}	R_1^1,$\overline{R}_1^1$	h_1,h_5 ; h_1,h_{18} ; h_5,h_{18} h_1,h_9 ; h_9,h_{18} ; h_5,h_9
C_3	h_1,h_{14}, h_{18}, h_5, h_9	$\overline{R}_1^1$	h_{14}, h_{18} ;h_{14}, h_5 ;h_{14}, h_9
C_4	h_1	$\overline{R}_1^1$,R_1^1,R_2^1,R_3^1	h_1
C_5	h_1,h_{14}	$\overline{R}_1^1$,R_2^1,	h_{14}
C_6	h_1,h_6,h_{14},h_{10},h_{19}	R_2^1	h_{10}, h_{14} ;h_1, h_6 ;h_6,h_{10} ;h_{19}
C_7	h_1, h_{13}	R_1^1,R_3^1	h_1, h_{13}
C_8	h_1, h_{10}	R_2^1,R_3^1	h_1, h_{10}
C_9	h_1, h_{10}, h_7, h_{13}	R_3^1	h_7 ;h_{10}, h_{13}
C_{11}	h_{10}	R_2^1,R_1^2,R_3^2	h_{10}
C_{12}	h_5	$\overline{R}_1^1$,R_1^1,R_1^2,R_1^c	h_5
C_{13}	h_2,h_3,h_5,h_{10},h_{15}	R_1^2	h_2,h_5 ; h_2,h_{10} ;h_5,h_{10} ; h_{15}
C_{14}	h_2, h_3	R_1^2,R_2^2,R_3^2	h_2
C_{15}	h_6	R_2^1,R_2^2	h_6
C_{16}	h_2, h_3, h_6, h_{12}, h_{16}	R_2^2	h_{16} ;h_2, h_6 ;h_3, h_6 ;h_{12}, h_2 ;h_{12}, h_3 ;h_{12}, h_6
C_{17}	h_{13}	R_2^1, R_3^1, R_3^2	h_{13}
C_{18}	h_{18}	$\overline{R}_1^1$,R_1^1,R_3^2	h_{18}
C_{19}	h_{13}, h_{18}	R_2^1, R_3^2	h_{13}, h_{18}

C_{20}	$h_{11}, h_{13}, h_{18}, h_2, h_3, h_8$	R_3^2	h_{11} ;h_8 ;h_2, h_{13} ;h_{18}, h_2 ;h_3, h_{13} ;h_3, h_{18}
C_{21}	h_3	R_1^2,R_2^2,R_3^2,R_1^c	h_3
C_{22}	h_5, h_9	$\overline{R}_1^1$,R_1^c,R_1^1	h_9
C_{23}	h_3, h_5	R_1^2,R_1^c	h_3, h_5
C_{24}	h_3, h_5, h_9	R_1^c	h_3, h_5, h_9
C_{25}	h_{12}	R_2^2,R_2^c	h_{12}
C_{26}	h_{12}, h_{17}, h_4	R_2^c	h_{17}, h_4
C_{27}	h_{12}, h_{17}, h_4	R_2^c	h_{17} ;h_4

Tableau B.1: Liste des concepts

RÉFÉRENCES

Arsovski, I., Chandler, T. et Sheikholeslami, A. (2003). A ternary content-addressable memory (tcam) based on 4t static storage and including a current-race sensing scheme. *IEEE Journal of Solid-State Circuits*, *38*(1), 155–158.

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

Blaiech, K., Hamadi, S., Valtchev, P., Cherkaoui, O. et Beliveau, A. (2015). Toward a semantic-based packet forwarding model for openflow. Dans *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, 1–6. IEEE.

Bozakov, Z. et Rizk, A. (2013). Taming sdn controllers in heterogeneous hardware environments. Dans *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, 50–55. IEEE.

Claise, B. (2004). Cisco systems netflow services export version 9.

Corp., B. (2014). *Openflow Data Plane Abstraction (OF-DPA)*. Rapport technique, Broadcom Corp.

Daniels, J. (2009). Server virtualization architecture and implementation. *Crossroads*, *16*(1), 8–12.

Denecke, K. *et al.* (2013). *Galois connections and applications*, volume 565. Springer Science & Business Media.

Doraswamy, N. et Harkins, D. (2003). *IPSec : the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional.

Düwel, S. et Hesse, W. (1998). Identifying candidate objects during system analysis. Dans *Third CAISE*, volume 98, 8–9. Citeseer.

Düwel, S. et Hesse, W. (2000). Bridging the gap between use case analysis and class structure design by formal concept analysis. Dans *Proceedings of Modellierung*, 27–40.

Enns, R., Bjorklund, M. et Schoenwaelder, J. (2011). Netconf configuration protocol. *Network.*

Feldman, M., Lai, K. et Zhang, L. (2005). A price-anticipating resource allocation mechanism for distributed shared clusters. Dans *Proceedings of the 6th ACM conference on Electronic commerce*, 127–136. ACM.

Fischer, B. (2000). Specification-based browsing of software component libraries. *Automated Software Engineering*, *7*(2), 179–200.

Gangwal, O. P., Rădulescu, A., Goossens, K., Pestana, S. G. et Rijpkema, E. (2005). Building predictable systems on chip : An analysis of guaranteed communication in the æthereal network on chip. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices* 1–36. Springer.

Ganter, B., Stumme, G. et Wille, R. (2005). *Formal concept analysis : foundations and applications*, volume 3626. springer.

Ganter, B. et Wille, R. (1999). *Formal concept analysis : mathematical foundations.* Springer.

Gomez, C., Gilabert, F., Gomez, M. E., Lopez, P. et Duato, J. (2008). Ruft : Simplifying the fat-tree topology. Dans *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, 153–160. IEEE.

Gondran, M. et Minoux, M. (2000). Dioïdes et semi-anneaux : algèbres et analyses pour le xxie siècle ? *TSI. Technique et science informatiques*, *19*(1-3), 253–262.

Guck, J. W. et Kellerer, W. (2014). Achieving end-to-end real-time quality of service with software defined networking. Dans *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, 70–76. IEEE.

Hamadi, S. *et al.* (2014). Fast path acceleration for open vswitch in overlay networks. Dans *Global Information Infrastructure and Networking Symposium, 2014*, 1–5. IEEE.

Hamadi, S., Blaiech, K. et Cherkaoui, O. (2015). Semantic-based forwarding model for network devices. Dans *Protocol Engineering (ICPE) and International Conference on New Technologies of Distributed Systems (NTDS), 2015 International Conference on*, 1–7. IEEE.

Jarschel, M., Oechsner, S., Schlosser, D., Pries, R., Goll, S. et Tran-Gia, P. (2011). Modeling and performance evaluation of an openflow architecture. Dans *Proceedings of the 23rd international teletraffic congress*, 1–7. International Teletraffic Congress.

Kanizo, Y., Hay, D. et Keslassy, I. (2013). Palette : Distributing tables in software-defined networks. Dans *INFOCOM, 2013 Proceedings IEEE*, 545–549. IEEE.

Karypis, G. et Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, *20*(1), 359–392.

Katta, N., Alipourfard, O., Rexford, J. et Walker, D. (2014). Infinite cacheflow in software-defined networks. Dans *Proceedings of the third workshop on Hot topics in software defined networking*, 175–180. ACM.

Mahmood, K., Chilwan, A., Østerbø, O. N. et Jarschel, M. (2014). On the modeling of openflow-based sdns : The single node case. *arXiv preprint arXiv :1411.4733.*

McKeown, N. *et al.* (2008). Openflow : enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, *38*(2), 69–74.

Menken, I. (2008). *Virtualization Architecture, Adoption and Monetization of Virtualization Projects using Best Practice Service Strategy, Service Design, Service Transition,... and Continual Service Improvement Processes.* Emereo Pty Ltd.

Natarajan, S. *et al.* (2012). Efficient conflict detection in flow-based virtualized networks. Dans *Computing, Networking and Communications, 2012 Intl Conf on*, 690–696. IEEE.

Netronome (2013). Argon blaster flow simulator and traffic generator. http ://www.netronome.com/pages/Flow-Simulation-and-Traffic-Generation-Solutions/. Récupéré de `http://www.netronome.com/`

Nunes, B. A., Mendonca, M., Nguyen, X.-N., Obraczka, K. et Turletti, T. (2014). A survey of software-defined networking : Past, present, and future of programmable networks. *Communications Surveys & Tutorials, IEEE*, *16*(3), 1617–1634.

Osgouei, A. G., Koohanestani, A. K., Saidi, H. et Fanian, A. (2015). Analytical performance model of virtualized sdns using network calculus. Dans *Electrical Engineering (ICEE), 2015 23rd Iranian Conference on*, 770–774. IEEE.

Pasquier, N. (2009). Frequent closed itemsets based condensed representations for association rules. *Post-Mining of Association Rules : Techniques for Effective Knowledge Extraction*, 246–271.

Pfaff, B. *et al.* (2015). The design and implementation of open vswitch. Dans

12th USENIX Symp on Networked Systems Design and Implementation.

Sahoo, J., Mohapatra, S. et Lath, R. (2010). Virtualization : A survey on concepts, taxonomy and associated security issues. Dans *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, 222–226. IEEE.

Salisbury, B. (2012). Tcams and openflow : What every sdn practitioner must know. *See http ://tinyurl. com/kjy99uw.*

Sankappanavar, H. P. et Burris, S. (1981). A course in universal algebra. *Graduate Texts Math, 78.*

Schmitt, J. B. et Zdarsky, F. A. (2006). The disco network calculator : a toolbox for worst case analysis. Dans *Proc of the 1st intl conf on Performance evaluation methodolgies and tools*, p. 8. ACM.

Snelting, G. (2000). Software reengineering based on concept lattices. Dans *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*, 3–10. IEEE.

Snelting, G. et Tip, F. (1998). *Reengineering class hierarchies using concept analysis*, volume 23. ACM.

Spivey, J. M. (1989). An introduction to z and formal specifications. *Software Engineering Journal*, *4*(1), 40–50.

Stallings, W. (1987). *Handbook of computer-communications standards ; Vol. 1 : the open systems interconnection (OSI) model and OSI-related standards.* Macmillan Publishing Co., Inc.

Stumme, G. *et al.* (2000). Fast computation of concept lattices using data mining techniques. Dans *KRDB*, 129–139.

Swanson, E. B. (1976). The dimensions of maintenance. Dans *Proceedings of the 2nd international conference on Software engineering*, 492–497. IEEE Computer Society Press.

Thiele, L. *et al.* (2002a). Design space exploration of network processor architectures. *Network Processor Design : Issues and Practices*, *1*, 55–89.

Thiele, L., Chakraborty, S., Gries, M. et Kunzli, S. (2002b). 4 design space exploration of network processor. *Network Processor Design : Issues and Practices, 1.*

Valtchev, P., Mounaouar, O., Cherkaoui, O., Dimitrov, A. et Marchand, L. (2012). Flowme : Lattice-based traffic measurement. *arXiv preprint arXiv :1211.2501.*

Yan, H., Maltz, D. A., Ng, T. E., Gogineni, H., Zhang, H. et Cai, Z. (2007). Tesseract : A 4d network control plane. Dans *NSDI*, volume 7, 27–27.

Zhang, S., Ivancic, F., Lumezanu, C., Yuan, Y., Gupta, A. et Malik, S. (2014). An adaptable rule placement for software-defined networks. Dans *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, 88–99. IEEE.