Titre: Title:	Architecture pour la recherche exacte dans le plan des données d'un processeur réseau implémenté sur FPGA
Auteur: Author:	Patrick Richer St-Onge
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Richer St-Onge, P. (2022). Architecture pour la recherche exacte dans le plan des données d'un processeur réseau implémenté sur FPGA [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10279/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/10279/
Directeurs de recherche: Advisors:	J. M. Pierre Langlois, & Jean Pierre David
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Architecture pour la recherche exacte dans le plan des données d'un processeur réseau implémenté sur FPGA

PATRICK RICHER ST-ONGE
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Avril 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Architecture pour la recherche exacte dans le plan des données d'un processeur réseau implémenté sur FPGA

présenté par **Patrick RICHER ST-ONGE**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Sylvain MARTEL, président
Pierre LANGLOIS, membre et directeur de recherche
Jean Pierre DAVID, membre et codirecteur de recherche
François-Raymond BOYER, membre

REMERCIEMENTS

La réalisation de ce travail de maîtrise n'aurait pas été possible sans le soutien de plusieurs personnes, dont j'aimerais remercier.

Je souhaite d'abord remercier Pierre Langlois et Jean-Pierre David d'avoir accepté de superviser mon travail de maîtrise et de m'avoir appuyé jusqu'à la fin. Leur aide a été essentiel lors de toutes les étapes du projet, de la conception à la rédaction.

Je veux aussi remercier les professeurs, le personnel, les étudiants et les partenaires industriels avec lesquels j'ai pu travailler au sein de la chaire de recherche. Entre autres, merci à Vincent et Thomas pour les discussions intéressantes et pour les séances de travail.

Enfin, je dois aussi remercier ma copine, ma famille et mes amis pour leur soutien. Merci à ma mère de me pousser à compléter ce que j'entreprends. Merci à Jérémie et Olivier d'avoir partagé cette expérience en parallèle.

RÉSUMÉ

Avec le débit grandissant des liens de transmission réseau, des architectures dédiées ont été conçues pour produire des puces pouvant maintenir un niveau de performance élevé. Traditionnellement, le traitement de paquets dans le plan des données d'un hôte à l'extrémité du réseau est effectué par un processeur à usage général, ce qui offre une grande flexibilité et qui simplifie la programmation. Afin de permettre une meilleure utilisation du processeur pour des tâches de calcul, une portion de la charge de travail en lien avec le traitement de paquets est transférée vers la carte réseau. La carte réseau doit alors pouvoir être programmée pour exprimer des algorithmes variés tout en maintenant le débit de traitement du lien. Une transition similaire arrive aussi avec les commutateurs au centre du réseau. Les fonctions d'un commutateur étant normalement fixes, l'arrivée de commutateurs programmables leur permet d'être responsables d'une plus grande portion du traitement au coeur du réseau, réduisant ainsi le besoin de serveurs externes.

Ce changement vers des architectures à la fois programmables et spécifiques à un domaine présente de nouvelles contraintes de conception pour les différentes cibles d'exécution. De plus, les différentes cibles d'exécution devraient partager une abstraction commune dans l'intérêt d'être toutes programmées par un même langage de programmation. Une telle cible est le FPGA, une plateforme reconfigurable qui est intéressante pour l'implémentation de plans des données programmables. Différents composants entrent dans la réalisation d'un pipeline de traitement de paquets, ayant chacun besoin de maintenir un débit élevé de transfert et une faible latence de traitement. Parmi ces composants se trouve la mémoire associative, le sujet central de ce travail. La mémoire associative est une partie essentielle des équipements réseau afin de pouvoir effectuer la recherche d'un champ d'un en-tête d'un paquet réseau dans une table.

Ce travail propose une architecture pour la recherche exacte adaptée pour un plan des données programmable sur FPGA. La recherche d'une clé dans la table se fait toujours en un seul cycle d'horloge. De plus, l'opération d'insertion d'une règle se fait aussi en un seul cycle, et parallèlement à la recherche. Ainsi, l'insertion ne bloque pas l'opération de recherche et il est possible d'obtenir une latence d'accès fixe en pire cas. L'implémentation est basée sur l'utilisation de plusieurs tables de hachage pour imiter le comportement d'une mémoire associative. Cette structure de données offre une meilleure utilisation de la mémoire que l'implémentation d'une mémoire adressable par le contenu sur FPGA. Pour accompagner cette architecture,

un flot de compilation à partir d'un programme P4 est aussi réalisé pour montrer la possibilité d'utiliser l'implémentation proposée dans un plan des données programmable. P4 est un langage de programmation dédié qui permet d'exprimer le traitement sur des paquets.

Enfin, étant donné que l'implémentation est réalisée entièrement en matériel, il est possible aussi d'insérer directement des règles à partir du pipeline matériel. Cela ouvre la possibilité à une forme de traitement avec états, permettant l'insertion d'une règle dans une table à partir du plan de données. Un cas d'utilisation est lors d'une recherche, lorsqu'une clé n'est pas présente dans une table, il est alors possible d'insérer la règle directement à partir du pipeline matériel. Cela permet d'accélérer le traitement du paquet tout en évitant un aller-retour avec le plan de contrôle.

ABSTRACT

With the continuously increasing line rate of networking hardware, domain-specific architectures have been designed to enable such high performance. Traditionally, packet processing of a network host is done by the general-purpose processor of that computer. This offers flexibility and ease of use in programming the behaviour of the data plane. However, in order to allow a higher utilization of the processor for its intended workload, part of the packet processing has been offloaded to the network interface card. Therefore, it must be possible to program the network card to express various packet processing algorithms while maintaining the line rate. A similar transition is also occurring with switches at the core of the network. Usually, network switches haven been fixed functions, however new programmable switches have been proposed which allow programming the data plane. More tasks can be performed by the programmable switches reducing the need for external servers.

The transition towards domain specific architectures that are also programmable presents new challenges. Specific designs are needed in order to achieve high performance on different execution targets. Moreover, the many targets should share a common abstraction to be able to program them with one programming language. One target is the FPGA, a reconfigurable platform that has potential for the implementation of a programmable packet processing pipeline. Many components are involved in the creation of a complete packet pipeline with each part needing to be carefully designed in order to achieve a high throughput and a low processing latency. One of such component is the associative memory which is the main theme of this project. Associative memory is an essential part of networking hardware as it is the component that allows to lookup the field of a packet header in a table.

The work proposes an exact match architecture suitable for programmable data plane on FPGA. Lookup of a key in the table is always done in a single clock cycle. Furthermore, the insertion of a rule is also done in a single cycle and parallel to the lookup operation. Thus, the insertion operation does not block the lookup operation which means the lookup latency in worst case is constant. The implementation is based on a design that uses multiple hash tables to simulate the behaviour of an associative memory. This data structure also offers a better memory utilization than a hardware implementation of a content-addressable memory on FPGA. In addition to the proposed architecture, compilation from a P4 program is also demonstrated to show the possibility of using this implementation in a programmable data plane on FPGA. P4 is domain-specific language used to describe packet processing in a network device.

Finally, because the entire exact match implementation is done in hardware, it is possible to insert an element in the table directly from the hardware pipeline. This enables partial stateful processing by allowing the insertion of a new rule in a table from the data plane. A use case is for the insertion of a new entry when a lookup results in no key being found in the table. When the insertion is done from the hardware data plane, the time to process a packet is shorter by avoiding a round trip to the control plane.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ALU	Unité arithmétique et logique – *Arithmetic Logic Unit*
ASIC	Puce dédiée – *Application Specific Integrated Circuit*
AST	Arbre de syntaxe abstraite – *Abstract Syntax Tree*
BCAM	CAM binaire – *Binary CAM*
BRAM	Bloc de mémoire RAM – *Block RAM*
CAM	Mémoire adressable par le contenu – *Content-Addressable Memory*
CPU	Processeur à usage général – *Central Processing Unit*
DPI	Inspection approfondie des paquets – *Deep Packet Inspection*
DRAM	Mémoire vive dynamique – *Dynamic RAM*
DSL	Langage dédié – *Domain-Specific Language*
DSP	Processeur de signal numérique – *Digital Signal Processor*
FF	Bascule – *Flip Flop*
FIFO	Premier entré, premier sorti – *First In First Out*
FPGA	Réseau prédiffusé programmable par l'utilisateur – *Field-Programmable Gate Array*
HLS	Synthèse de haut niveau – *High-Level Synthesis*
HDL	Langage de description matérielle – *Hardware Description Language*
IP	Protocole internet – *Internet Protocol*
JSON	*JavaScript Object Notation*
LPM	Comparaison du préfixe le plus long – *Longest Prefix Match*
LRU	Moins récemment utilisé – *Least Recently Used*
LUT	Table de conversion – *Lookup Table*
MAC	Contrôle d'accès au médium – *Media Access Control*
MAT	Table de comparaison-action – *Match-Action Table*
NIC	Carte d'interface réseau – *Network Interface Card*
NPL	Langage de programmation réseau – *Network Programming Language*
NPU	Processeur réseau – *Network Processing Unit*
ONF	*Open Networking Foundation*
OSI	Interconnexion de systèmes ouverts – *Open Systems Interconnection*
P4	Programmation de processeurs de paquets indépendant des protocoles – *Programming Protocol Independent Packet Processors*
PISA	Architecture de commutateur réseau indépendante des protocoles – *Protocol Independent Switch Architecture*

PNA	Architecture de carte réseau portable – *Portable NIC Architecture*
PSA	Architecture de commutateur portable – *Portable Switch Architecture*
RAM	Mémoire vive – *Random Access Memory*
RMT	Table de comparaison configurable – *Reconfigurable Match Table*
SDN	Réseau défini par logiciel – *Software-Defined Networking*
SRAM	Mémoire vive statique – *Static RAM*
TCAM	CAM ternaire – *Ternary CAM*
TCP	Protocole de contrôle de transmission – *Transmission Control Protocol*
UDP	Protocole de datagramme utilisateur – *User Datagram Protocol*
VHSIC	Circuit intégré à très grande vitesse – *Very High Speed Integrated Circuit*
VHDL	Langage de description matérielle pour VHSIC – *VHSIC Hardware Description Language*
WNS	Délai négatif en pire cas – *Worst Negative Slack*

CHAPITRE 1 INTRODUCTION

Avec l'augmentation du nombre de clients connectés à Internet et le débit grandissant des liens de transmission, il est nécessaire que les équipements réseau soient performants pour répondre à ces besoins. Les commutateurs réseau utilisent des architectures matérielles spécialisées qui sont adaptées aux contraintes du traitement de paquets et qui offrent d'excellentes performances. Par contre, la facilité de programmation des processeurs à usage général est un aspect attrayant pour permettre la personnalisation et la mise à jour des algorithmes de traitement de paquets. Ainsi, des architectures spécialisées programmables ont été développées pour répondre aux critères de performance et de reconfigurabilité.

1.1 Définitions et concepts de base

Pour commencer, les concepts utiles à la compréhension des enjeux de ce travail sont présentés. Ces concepts incluent le plan de contrôle, le plan des données et l'arrivée de la programmabilité dans le réseau.

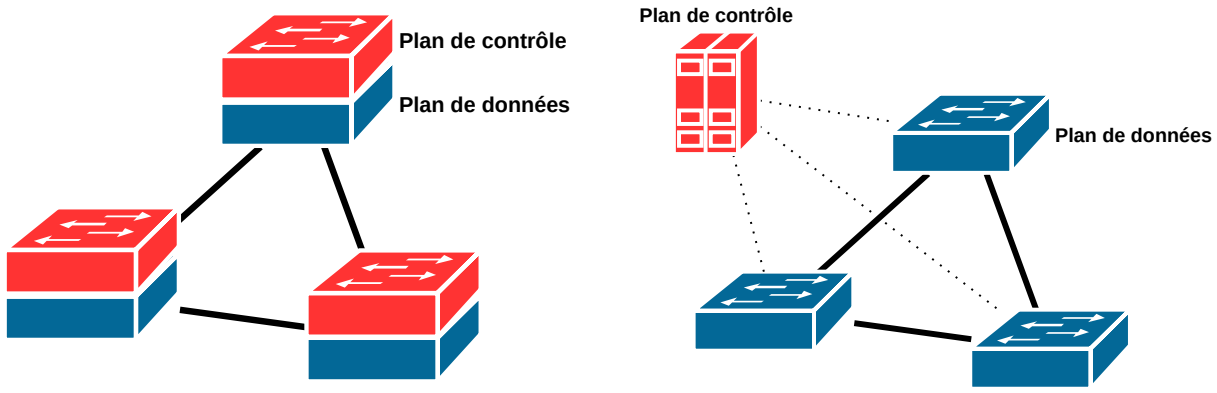
1.1.1 Plans dans l'architecture d'un réseau

Le fonctionnement d'un réseau, du point de vue du routage de paquets, est divisé en deux couches principales, soit le plan de contrôle et le plan des données. Ces plans font partie de l'architecture d'un routeur ou autres équipements réseau et ont chacun un rôle précis. Par exemple, tout routeur qui reçoit un paquet en entrée doit déterminer quel chemin le paquet doit emprunter et être en mesure d'acheminer rapidement le paquet vers le port de sortie approprié. Ces fonctions sont accomplies par le plan de contrôle (décision) et le plan des données (exécution). Dans un réseau traditionnel, comme montré à la Figure 1.1a, chaque commutateur est constitué d'un plan des données accompagné par un plan de contrôle local.

1.1.1.1 Le plan de contrôle

Le plan de contrôle est la partie responsable des décisions par rapport au traitement à effectuer sur les paquets entrants. Dans un routeur, c'est le plan de contrôle qui construit la table de routage et détermine le chemin que doit emprunter un paquet. Par exemple, un routeur peut décider que les paquets correspondants à une certaine plage d'adresses de destination IPv4 doivent être acheminés vers un port de sortie spécifique. C'est à travers des règles de

cette forme et qui sont contenues dans une table que le plan de contrôle détermine les actions à poser sur les paquets. C'est par ces tables que le plan de contrôle réalise l'interface avec le plan des données.



(a) Réseau traditionnel

(b) Réseau défini par logiciel (SDN)

FIGURE 1.1 Séparation du plan de contrôle et du plan des données. Le plan de contrôle est extrait des noeuds réseau et le rôle de contrôleur réseau est centralisé dans un ou plusieurs serveurs.

1.1.1.2 Le plan des données

Le plan des données est responsable d'appliquer le traitement sur les paquets entrants. Cela est fait par des recherches dans les tables qui contiennent les règles provenant du plan de contrôle. Pour continuer l'exemple du routeur, c'est le plan des données qui est responsable d'extraire le champ correspondant à l'adresse de destination IPv4 du paquet, d'effectuer la recherche dans la table de routage et d'acheminer le paquet vers le port de sortie obtenu de la table.

1.1.2 Le réseau défini par logiciel

Dans son ensemble, le réseau défini par logiciel - *Software-Defined Networking* (SDN) réfère à une architecture de réseau qui comporte des éléments programmables. C'est une transition progressive qui est déjà amorcée depuis plusieurs années, mais qui est encore en train de se développer pour couvrir un plus grand champ d'application.

Un point marquant de la transition vers le réseau défini par logiciel est la séparation du plan de contrôle et du plan des données [1]. Les deux plans maintiennent leurs fonctions et leurs rôles respectifs, mais le plan de contrôle est extrait des commutateurs réseau afin d'être placé

sur un noeud réseau à part. Ce noeud peut être un serveur, plusieurs serveurs ou même dans le nuage informatique. Un exemple de cette séparation des plans est présenté à la Figure 1.1b. Chaque commutateur est constitué du plan des données seulement et chacun est contrôlé par un plan de contrôle centralisé. Le plan de contrôle peut ainsi avoir une vue d'ensemble du réseau et cela simplifie la gestion et la création des règles.

La séparation du plan de contrôle et du plan des données est possible, car une définition ouverte de l'interface entre les deux plans existe. Une interface standardisée et ouverte permet l'interopérabilité entre les équipements réseau de différents vendeurs.

1.1.3 Le plan des données programmable

La séparation du plan de contrôle et du plan des données facilite la gestion du réseau par les opérateurs et améliore l'interopérabilité entre les équipements. Cependant, le plan des données a encore une fonction fixe avec peu de configuration possible. L'ajout d'un protocole réseau ou la personnalisation d'un algorithme de traitement de paquets n'est pas possible sans le développement et la fabrication d'une nouvelle puce matérielle. Pour améliorer cette situation, une architecture programmable pour exprimer le traitement de paquets a été proposée conjointement avec une puce matérielle qui implémente cette architecture [2]. Ces nouveaux commutateurs sont performants en plus d'être programmables, permettant ainsi de modifier et d'améliorer les fonctions effectuées par ces équipements lors de l'utilisation.

Pour permettre la programmation du plan des données, un langage de programmation spécialisé est utilisé. Le langage proposé avec l'architecture mentionnée ci-haut se nomme P4 [3]. À l'aide de ce langage, il est possible de décrire les en-têtes que le plan des données doit prendre en charge, permettant l'ajout de protocoles réseau. Le langage permet aussi de programmer le pipeline de traitement, qui peut ajouter, modifier ou enlever des en-têtes d'un paquet. Cette combinaison d'une puce programmable et d'un langage dédié est ce qui permet le plan des données programmable.

1.2 Éléments de la problématique

Il y a plusieurs considérations à prendre en compte dans la réalisation d'un plan des données programmable. Au niveau logiciel, des abstractions claires doivent être présentées au développeur et la programmation doit se faire par un langage haut niveau adapté pour ces abstractions. Au niveau matériel, des implémentations basées sur ces mêmes abstractions doivent être développées afin qu'elles soient programmables et performantes.

D'abord, l'adoption du plan des données programmable est possible, car il existe des solutions matérielles offrant le même niveau de performance et les mêmes fonctionnalités de base qu'un commutateur traditionnel. Ces solutions ont été réalisées sur des circuits intégrés utilisant des architectures spécialisées pour permettre la programmation d'un pipeline de traitement de paquets. Ces puces traitent des paquets avec un grand débit, garantissent une latence en pire cas et sont programmables. Par contre, ces commutateurs programmables comportent aussi des limitations. Deux limitations majeures sont la petite taille de la mémoire vive intégrée à la puce et la faible flexibilité pour exprimer certains algorithmes ou réaliser certaines fonctionnalités dans le pipeline. Ces limitations sont en partie voulues par conception, car pour obtenir une faible latence et garantir une borne supérieure, il n'est pas possible d'appliquer n'importe quel algorithme.

À l'inverse, les cibles logicielles contiennent une grande capacité de mémoire vive et peuvent exprimer des algorithmes de complexité variable. Le développement de programmes réseau se fait plus facilement en programmant sur une cible logicielle. C'est idéal pour prototyper rapidement, exécuter une suite de tests ou pour apprendre le langage de programmation. Par contre, les cibles logicielles ne peuvent pas atteindre la même performance en termes de latence et de débit que les puces programmables spécialisées. De plus, un programme réseau qui exécute sur une cible logiciel n'est pas assuré de fonctionner sur une cible matérielle étant donné que les limitations de l'implémentation matérielle ne sont pas représentées.

Ainsi, il existe un attrait pour une solution intermédiaire qui se situe entre le processeur à usage général et la puce spécialisée. Une technologie candidate pour cet intermédiaire est le FPGA. L'utilisation de FPGA pour réaliser un plan des données programmable permet de répondre à certaines des limitations mentionnées plus haut. Pour des applications spécifiques, tel que le traitement de paquets, un FPGA est plus programmable qu'une puce spécialisée et plus performant qu'un processeur à usage général.

D'abord, un FPGA est un circuit programmable, ce qui se prête naturellement à la programmation sur place, un requis pour un plan des données programmable. Les composants matériels d'un FPGA peuvent être programmés pour obtenir différentes implémentations et n'imposent pas une architecture particulière. Il est possible d'exprimer des algorithmes complexes pour traiter des paquets. De plus, certaines puces FPGA sont intégrées avec des mémoires externes, ce qui offre une plus grande capacité mémoire. Ensuite, du point de vue de la performance, il est possible d'atteindre un débit de traitement plus élevé sur FPGA que sur un processeur à usage général.

Par contre, une problématique avec l'implémentation d'un plan des données sur FPGA est l'expertise nécessaire pour pouvoir programmer un FPGA. La réalisation d'une solution performante sur FPGA nécessite une connaissance des composants du FPGA et de la conception de modules matériels adaptés aux besoins de l'application. La disponibilité d'une bibliothèque de composants en lien avec le traitement de paquets serait bénéfique pour le développement de plans des données programmables sur FPGA. Afin de permettre la programmation à partir d'un langage dédié, tel que P4, il faut aussi que ces composants soient conçus pour être instanciés à partir d'un compilateur. Ainsi, un programme P4 pourrait être utilisé pour implémenter un plan des données sur FPGA.

Parmi ces composants se trouve la table de recherche, permettant de chercher une donnée dans une mémoire associative. L'implémentation d'une telle table sur FPGA peut prendre plusieurs formes selon les besoins de l'application. Dans le contexte du plan des données programmable, il est requis d'effectuer plusieurs recherches dans des tables pour le traitement d'un paquet, tout en assurant une faible latence dans le pipeline. Il est intéressant d'étudier et de concevoir une architecture pour une table de recherche sur FPGA afin d'avoir un composant adapté aux besoins de plans des données programmables.

1.3 Objectifs de recherche

Ce travail a d'abord comme but de comprendre les besoins et contraintes du plan des données programmable. Plus spécifiquement, l'aspect qui est étudié est la fonction de recherche exacte utilisée pour la réalisation des tables de comparaison-action. Une architecture pour la recherche exacte est conçue sur FPGA en prenant en compte les différents besoins d'un plan des données programmable, tels que la latence de recherche, la latence d'insertion et la capacité de stockage. De plus, la compilation à partir du langage dédié P4 est utilisée afin d'instancier ce module de recherche exacte, permettant l'utilisation de cette implémentation dans une solution complète de plan des données programmable compilé de P4 vers FPGA.

Les contributions principales de ce travail sont :

- La conception d'une architecture matérielle pour la recherche exacte dans une table ;
- L'instanciation d'un module basé sur cette architecture à partir d'un langage de programmation de plans des données ; et,
- L'analyse de l'implémentation d'une table de hachage sur FPGA.

1.4 Plan du mémoire

La suite de ce mémoire est divisée en quatre chapitres. Dans le chapitre 2, une revue de la littérature est réalisée pour couvrir les concepts pertinents au plan des données programmable et pour présenter les travaux connexes. Par la suite, le coeur du travail sur l'architecture de comparaison exacte est présenté en détail dans le chapitre 3 en plus d'une analyse de performance de l'implémentation utilisant des tables de hachage. Le chapitre 4 présente le flot de compilation partielle du langage P4 vers l'implémentation matérielle proposée. Finalement, le chapitre 5 exprime certaines limitations et améliorations possibles de l'architecture proposée et conclut le travail de ce mémoire.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre approfondit les concepts vus dans l'introduction pour éclaircir le contexte de ce travail et bien en situer les contributions. D'abord, il est question des différents équipements qui forment le réseau, des architectures qui composent ces équipements et de leur implémentation. Ensuite, les tables de comparaison-action, un élément clé du plan des données programmable, seront présentées. Finalement, on abordera les langages et outils qui permettent de programmer ces abstractions.

2.1 Commutateur réseau et plan des données

Le plan des données est la partie dans tout appareil réseau qui est chargée d'effectuer les manipulations voulues sur les paquets entrants et de les diriger vers le bon port de sortie [4]. Le rôle exact que joue le plan des données varie d'un équipement à l'autre. Par exemple, la fonction traditionnelle d'un commutateur réseau est de relier les hôtes ou appareils d'un même réseau et d'acheminer les paquets aux bons destinataires. C'est un rôle qui s'opère normalement au niveau de la deuxième couche dans le modèle de référence OSI, soit la couche de liaison. Pour ce faire, un commutateur a connaissance des adresses MAC des appareils et d'un protocole de liaison tel qu'Ethernet. Dans le cas d'un routeur, son rôle est de connecter des réseaux entre eux et d'acheminer des paquets entre les réseaux. Il opère au niveau de la couche réseau, qui est la troisième couche du modèle OSI. Un routeur utilise le protocole IP et manipule des adresses IP pour effectuer le routage entre réseaux.

Avec l'arrivée du plan des données programmables, ces rôles traditionnellement associés à un noeud spécifique dans le réseau deviennent plus flous. Le terme commutateur programmable est utilisé pour décrire tout équipement réseau comportant un plan des données qui apporte des modifications à des paquets. Ainsi, un commutateur programmable peut être autant chargé de la commutation que du routage de paquets. Une architecture combinant commutateur et routeur opérant sur les couches de liaison et réseau existait déjà avant les commutateurs programmables [5]. Par contre, la programmation du plan des données pousse ce concept encore plus loin en permettant d'exprimer plus de fonctions et de pouvoir changer ces fonctions dans le temps. La flexibilité du plan des données programmable permet d'implémenter de nouvelles fonctionnalités directement sur un commutateur, comme l'équilibrage de charge au sein d'un centre de données [6]. Un commutateur programmable peut aussi être utilisé pour accélérer certaines applications, telles que l'entraînement d'un modèle d'apprentissage automatique [7].

Même si le changement vers le plan des données programmable permet de nouvelles applications et facilite la gestion du réseau par les opérateurs, ce ne doit pas être au détriment de la performance des commutateurs dans le réseau. L'évolution des technologies des liens de transmission et des connecteurs requiert que les commutateurs soient capables de traiter des paquets à un débit de plus en plus élevé. On observe que le traitement de paquets dans le plan des données est un problème comportant des contraintes différentes que le traitement sur un processeur à usage général. Le traitement de paquets est un domaine spécifique qui a avantage à utiliser des architectures personnalisées pour atteindre les besoins en termes de débit et de latence. Un requis important pour un commutateur est d'opérer sur des paquets au débit de la somme de ses liens de transmission. Pour assurer cela, un commutateur programmable ne peut pas contenir, par exemple, de boucle non bornée dans son traitement. Une boucle qui prend un nombre variable de cycles à l'exécution empêche de borner la latence de traitement d'un paquet.

Du point de vue du débit, les processeurs à usage général ne permettent pas d'atteindre la performance requise. Dans une étude sur la programmation de carte d'interface réseau, les systèmes sur puce composés de processeurs multicoeurs sont évalués pour le traitement de paquets [8]. Tandis que le système est réalisable et satisfaisant pour un débit de 10 Gbit/s, l'utilisation d'un lien de 40 Gbit/s demande 4 × plus de coeurs, ce qui augmente la taille de la puce, l'énergie consommée et le coût de production. Afin d'atteindre la performance requise d'un commutateur, une architecture personnalisée pour le plan des données programmable est nécessaire.

Les prochaines sous-sections présentent l'architecture de base utilisée pour programmer le plan des données, les architectures spécialisées associées aux différents types d'équipements réseau et la réalisation de ce modèle de traitement de paquets dans les commutateurs réseau.

2.1.1 L'architecture programmable de base

Bosshart et al. ont proposé en 2013 une architecture pour le traitement de paquets sur une puce matérielle programmable [2]. Cette architecture est devenue l'architecture de commutateur réseau indépendante des protocoles – *Protocol Independent Switch Architecture* (PISA). Comme son nom l'indique, PISA n'est pas associée à des protocoles réseaux communs comme IP ou UDP, mais permet d'exprimer explicitement ces protocoles standards tout comme des protocoles personnalisés. Même si l'architecture est définie pour un commutateur, le terme est utilisé pour signifier tout appareil réseau qui opère sur des paquets. En effet, cette architecture permet d'exprimer de nombreuses applications.

Les trois parties principales de PISA, montrées à la Figure 2.1, sont un parseur, un pipeline d'unités de comparaison-action et un déparseur.

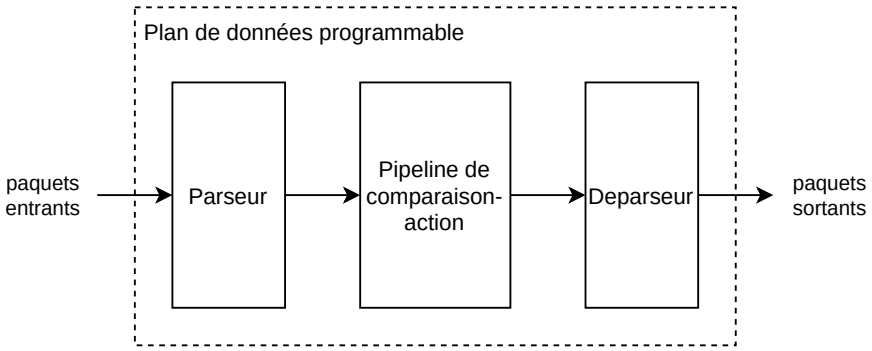


FIGURE 2.1 Modèle de base d'un plan des données programmable (PISA)

Le premier élément de l'architecture, le parseur de paquets programmable, a été étudié en profondeur par Gibb et al. [9]. Le rôle du parseur est d'extraire les en-têtes représentant ces protocoles dans les paquets entrants et de transmettre ces champs à la prochaine étape, le pipeline de comparaison-action. Un tel parseur occupe seulement 1.5× à 3× plus d'espace sur un ASIC de commutateur qu'un parseur fixe, ce qui représente environ 4% de l'espace total. L'avantage est que le parseur programmable peut être programmé pour fonctionner avec un ensemble de protocoles réseau au choix.

Le deuxième élément de PISA est le pipeline de tables de comparaison-action – *Match-Action Tables* (MAT) qui se base sur le concept de table de comparaison, un concept déjà présent dans le réseau. Par contre, certains ajouts permettent de rendre cette étape programmable. D'abord, il est possible de programmer les tables pour choisir la clé de comparaison. Une table peut utiliser un ou plusieurs champs d'en-êtes parmi les protocoles pris en charge par le parseur. Ensuite, la partie action des unités de comparaison-action est aussi programmable. Chaque bloc d'action peut être vu comme une unité arithmétique et logique – *Arithmetic Logic Unit* (ALU). Le bloc d'action reçoit à partir de la table l'opération à effectuer ainsi que les paramètres à opérer dessus. Cela permet d'effectuer des opérations arithmétiques de base sur les en-têtes d'un paquet. Finalement, une série de tables de comparaison-action peuvent être programmées afin d'exprimer des algorithmes plus complexes.

Enfin, le troisième et dernier élément de l'architecture est le déparseur. Le déparseur place les en-têtes qui ont été modifiés, ajoutés ou retirés, dans un ordre qui peut aussi être programmé. Il est responsable d'assembler ces en-têtes dans un flot de données pour être transmis et potentiellement combiner ce flot avec le corps du paquet pour acheminer le paquet complet vers le port de sortie du commutateur.

L'architecture PISA cherche à répondre aux besoins généraux du traitement de paquets dans le réseau, mais est toutefois spécifique à cette application et restreinte dans les algorithmes qu'elle peut implémenter. Par exemple, l'inspection approfondie des paquets – *Deep Packet Inspection* (DPI) ne se prête pas naturellement à l'architecture PISA. Il est nécessaire de connaître l'implémentation de l'architecture, telle que le nombre de tables de comparaison-action et la mémoire disponible, afin de concevoir un programme qui s'adapte à PISA. C'est ce qui a été réalisé pour la recherche de chaînes de caractères, un problème nécessitant de traiter en entier le corps d'un paquet [10]. Une architecture qui permet la recirculation de paquets, qui consiste à retourner un paquet au début du pipeline de traitement, est en mesure d'imiter la construction d'une boucle et d'inspecter le contenu du corps d'un paquet en plusieurs passes. Par contre, cela a pour conséquence de diminuer le débit de traitement du commutateur et d'augmenter la latence pour traiter un paquet.

2.1.2 Les architectures spécialisées

Des architectures spécialisées de plan des données programmables ont été établies à partir des blocs de base qui font partie de PISA. Ces architectures ont été créées pour standardiser les fonctions fixes et les métadonnées nécessaires à certains ensembles d'équipements réseau. L'organisation P4 a défini une spécification pour deux types d'équipements, soit un commutateur et une carte réseau.

2.1.2.1 L'architecture de commutateur portable

L'architecture de commutateur portable – *Portable Switch Architecture* (PSA) est basée sur le modèle PISA et étendue afin de mieux répondre aux besoins des commutateurs d'infrastructure à plusieurs ports (par ex. plus de 12 ports) [11]. Les mêmes blocs présents dans PISA font aussi partie de PSA, mais le pipeline de traitement est divisé et répété deux fois pour former un pipeline d'entrée et un pipeline de sortie. Deux modules à fonctions fixes sont ajoutés dans la description de l'architecture. Le gestionnaire de trafic se trouve entre les deux pipelines. Le gestionnaire de trafic peut avoir plusieurs rôles, tel que d'ordonnancer les paquets selon un ordre de priorité, de gérer le débit d'un flot et d'effectuer les opérations d'abandon, de mise en tampon ou de réplication des paquets. À la sortie du commutateur, des files d'attente stockent les paquets en mémoire tampon afin de les acheminer vers leur port de sortie. Dans la Figure 2.2, le gestionnaire de trafic et les files d'attente sont dans les blocs gris pâle, puisque ce ne sont pas des blocs programmables de l'architecture.

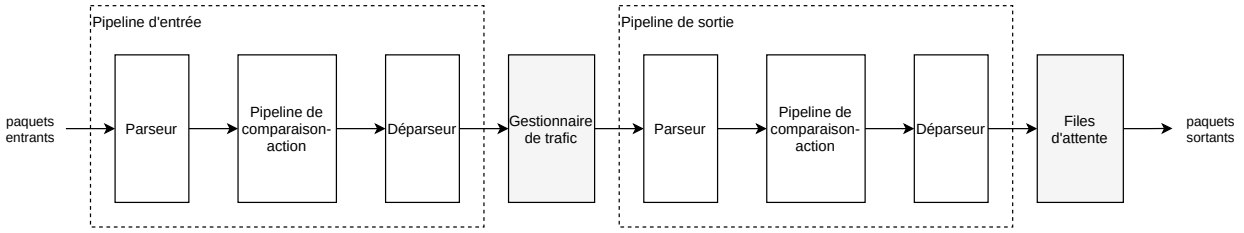


FIGURE 2.2 L'architecture de commutateur portable (PSA)

La présence d'un pipeline d'entrée et d'un pipeline de sortie offre plus de flexibilité pour le traitement. Lorsqu'un paquet passe par le gestionnaire de trafic, une décision peut être prise de soumettre à nouveau le paquet au pipeline d'entrée, d'effectuer une ou des copies du paquet pour la multidiffusion, d'abandonner le paquet ou, par défaut, de poursuivre le traitement par le pipeline de sortie.

2.1.2.2 L'architecture de carte réseau portable

Une carte d'interface réseau – *Network Interface Card* (NIC) permet de relier un hôte à un réseau. C'est la première étape par laquelle un paquet doit passer pour se rendre au processeur de l'hôte, et, à l'inverse, la dernière étape lorsqu'un paquet est envoyé de l'hôte vers le réseau. Tout comme un commutateur réseau, une carte réseau a avantage à pouvoir être programmée. Par contre, les besoins et contraintes d'une carte réseau sont différents, ce qui nécessite une architecture différente.

L'architecture de carte réseau portable – *Portable NIC Architecture* (PNA) est une telle architecture qui permet d'exprimer une NIC programmable [12]. PNA se base sur les blocs présents dans l'architecture PISA, comme le parseur, le pipeline d'unités de comparaison-action et le déparseur. Par rapport à la norme PSA qui est pour les commutateurs réseau, PNA comporte plusieurs différences. D'abord, PNA n'a qu'un seul pipeline principal qui permet d'opérer sur les en-têtes d'un paquet. Ce pipeline programmable est augmenté de fonctions externes qui implémentent des fonctionnalités plus avancées telles que le chiffrement du corps du paquet. Ensuite, la fonction de PNA étant située entre le réseau et un hôte, le standard inclut aussi une extension pour le traitement de message. Sur le réseau, l'unité de transmission est un paquet, par contre l'hôte peut opérer sur des messages de plus grande taille qui doivent être divisés en paquets. Cette extension a comme but de permettre la programmation de la conversion entre des messages et des paquets.

2.1.3 Les implémentations du modèle

Tandis que l'architecture décrit le modèle de traitement général, l'implémentation ou microarchitecture est la réalisation de ce modèle sur une cible d'exécution choisie. L'implémentation est adaptée aux particularités de la cible pour offrir fonctionnalités et performance.

Un commutateur réseau peut être implémenté sur différentes cibles d'exécution [13]. La cible la plus commune est la puce dédiée – *Application Specific Integrated Circuit* (ASIC). Ce sont des solutions prêtes à l'emploi qui jouent un rôle précis dans un réseau à cause de leurs fonctionnalités fixes. Un ASIC est conçu pour une fonction précise et généralement ne permet pas d'être programmé, c'est-à-dire qu'il n'est pas possible d'exprimer un nouvel algorithme particulier lors de l'utilisation. Par contre, il offre les meilleures performances en matière de traitement, c'est-à-dire la plus haute bande passante et la plus faible latence par rapport aux autres cibles. À l'inverse, les solutions basées sur un processeur à usage général – *Central Processing Unit* (CPU) sont complètement programmables (même si limitées par leur interface d'entrée/sortie), mais ne permettent pas d'atteindre les mêmes niveaux de performance de traitement qu'un ASIC.

Parmi les solutions intermédiaires, le processeur réseau – *Network Processing Unit* (NPU) se rapproche beaucoup plus du CPU en ce qui concerne la performance et la programmabilité [4]. L'architecture d'un NPU va intégrer un certain nombre de concepts en lien avec le traitement de paquets afin d'accélérer le traitement, mais fonctionne de façon similaire à un processeur général. Une autre solution est le FPGA, une technologie qui permet d'implémenter des circuits logiques. Un FPGA est programmé à partir d'une description d'un circuit matériel, permettant entre autres d'implémenter une architecture spécialisée pour le traitement de paquets. Par contre, la programmation d'un FPGA est plus complexe que la programmation sur un CPU.

2.1.3.1 RMT

L'avancée marquante pour le plan des données programmable est la réalisation d'un ASIC programmable pour le traitement de paquets. Proposé par Bosshart et al. en 2013, la première puce programmable de la sorte est le modèle des tables de comparaison configurable – *Reconfigurable Match Tables* (RMT) [2]. RMT est l'implémentation matérielle de l'architecture PISA. La proposition initiale décrit une puce qui atteint un débit total de 640 Gbit/s, soit pour un commutateur à 64 ports de 10 Gbit/s chaque. Aujourd'hui, les puces commerciales basées sur le modèle RMT atteignent jusqu'à 6.4 Tbit/s et 12.8 Tbit/s de débit de

traitement [14, 15]. Ces débits sont comparables à ceux des puces de commutateur fixe. Le grand avantage est le niveau de programmabilité plus élevé, mais tout en conservant le même niveau de performance attendu d'un commutateur.

Le besoin en mémoire associative pour implémenter les tables de comparaison est un coût à prendre en compte dans la conception matérielle. C'est un aspect qui limite la quantité de mémoire qui peut être rendue disponible pour implémenter les tables, ce qui contraint le nombre de règles qui peuvent être conservées dans le commutateur. Dans le modèle RMT proposé, la mémoire totale pour les tables de comparaison est de 370 Mb de SRAM et 40 Mb de TCAM. La mémoire SRAM est utilisée entre autres pour réaliser les tables de comparaison exacte, qui sont émulées à l'aide de tables de hachage. La mémoire TCAM est utilisée pour la comparaison du préfixe le plus long – *Longest Prefix Match* (LPM). L'utilisation de mémoire plus dense telle que la DRAM n'est pas envisagée comme solution pour l'implémentation des tables de comparaison vu la latence plus élevée et variable de la DRAM.

2.1.3.2 SDNet

La nature configurable d'un FPGA rend cette technologie apte à la création de plans des données programmables. Une des implémentations sur FPGA est SDNet, une solution fournie par Xilinx pour le traitement de paquets sur FPGA [16]. SDNet est composée de différents modules matériels qui peuvent être assemblés pour former un système qui opère sur des paquets. Ces modules ont des fonctions similaires aux blocs de base de l'architecture PISA. Par contre, SDNet est beaucoup plus flexible et permet d'assembler ces modules avec un plus grand degré de liberté. Puisqu'il est possible de reconfigurer le circuit matériel d'un FPGA, il n'est pas nécessaire d'adopter une architecture spécifique, telle que PSA ou PNA. Il est toutefois toujours possible d'implémenter ces architectures selon le rôle que joue le FPGA dans le réseau.

Les composantes de SDNet consistent en un module d'extraction des en-têtes de paquets, un module pour modifier le contenu de paquets, un module pour la recherche dans des tables et un module pour connecter des fonctions définies par l'utilisateur. La similarité avec les blocs de PISA (parseur, unité de comparaison-action et déparseur) est forte et ces modules permettent de construire un système qui a le même fonctionnement qu'un pipeline dans PISA. Un avantage du plan des données sur FPGA est la possibilité pour les développeurs de programmer des modules personnalisés qui s'intègrent dans le système SDNet. Par contre, une connaissance des langages de description matérielle est nécessaire, vu que ces modules utilisateurs doivent être codés en Verilog ou VHDL.

SDNet permet d'atteindre un débit de traitement sur FPGA dans la plage de 10 à 100 Gbit/s et pouvant aller jusqu'à 400 Gbit/s selon le système implémenté. C'est environ un ordre de grandeur de moins que le débit atteignable sur un ASIC programmable, mais ce sont des valeurs typiques pour un plan des données sur FPGA.

Pour réaliser un plan des données sur FPGA avec SDNet, il faut d'abord écrire une spécification du système en utilisant les modules SDNet dans un format textuel. Un compilateur spécialisé traduit la spécification SDNet en une description matérielle dans le langage Verilog. Ensuite, le code Verilog peut être fourni en entrée à l'outil de synthèse Vivado pour générer une liste d'interconnexions afin d'instancier la conception sur un FPGA.

2.1.3.3 Autres approches

Un plan des données programmable peut aussi être réalisé en logiciel sur un processeur à usage général. Un processeur à usage général comporte déjà la flexibilité requise pour exprimer l'ensemble des fonctions nécessaire pour le traitement de paquets. Le défi alors est d'optimiser ces implémentations logicielles pour aller chercher le plus de performance des processeurs. Même si les commutateurs logiciels ne permettent pas d'atteindre le même débit de traitement qu'un ASIC, ils sont utiles pour des cas qui ne peuvent pas être exprimés facilement sur un ASIC ou lorsqu'une grande capacité de mémoire est nécessaire. Un tel commutateur logiciel est le programme Open vSwitch (OVS) [17]. Ce programme permet d'instancier des commutateurs virtuels de façon automatique, ce qui est une utilisation commune dans le nuage informatique. OVS utilise certaines techniques afin que les opérations sur les paquets soient faites dans le noyau du système d'exploitation, ce qui est plus performant. La plupart des implémentations de plan des données en logiciel bénéficient de technologies comme DPDK ou eBPF qui permettent d'ignorer la suite TCP/IP du système d'exploitation et d'employer le noyau pour le traitement de paquets [18, 19].

Certaines implémentations logicielles servent de simulateurs pour vérifier le fonctionnement d'un plan des données. C'est les cas du modèle comportemental BMv2 qui est décrit à la section 2.4.2. Ces commutateurs ne sont pas créés pour être utilisés dans un environnement de production, mais pour aider le développement par l'emploi de simulations en logiciel.

2.2 La réalisation matérielle des tables de comparaison

L'implémentation des tables utilisées dans les unités de comparaison-action est essentielle à la performance du pipeline de traitement. La fonction principale est d'effectuer une recherche d'un ou de plusieurs champs d'en-êtes de protocoles réseau dans une table. La technologie

matérielle apte pour la recherche est la mémoire associative qui est la première solution présentée dans cette section. De plus, des méthodes basées sur des structures de données permettent d'émuler le fonctionnement d'une mémoire associative à l'aide de mémoire vive statique – *Static RAM* (SRAM). Parmi ces structures, les tables de hachage sont présentées, suivies d'une approche de hachage particulière, le hachage coucou.

2.2.1 Mémoire associative

Une mémoire associative, aussi connue sous le nom de mémoire adressable par le contenu – *Content Addressable Memory* (CAM), est un type de mémoire qui permet la recherche. À l'inverse de la mémoire vive – *Random Access Memory* (RAM), il n'est pas nécessaire avec une CAM de connaître l'adresse de la donnée. Pour utiliser une RAM, il faut fournir en entrée une adresse pour récupérer la donnée à cette position dans la mémoire. Une CAM permet de chercher une donnée, typiquement appelée la clé, dans une mémoire et de retourner la valeur qui est associée à cette clé. La combinaison d'une clé et d'une valeur est souvent appelée une paire clé-valeur. Une CAM est l'équivalent matériel du tableau associatif ou dictionnaire en logiciel. C'est une structure qui peut être présentée comme un tableau contenant des entrées sous la forme de paires clé-valeur.

Il existe deux types de CAM, soit la CAM binaire – *Binary CAM* (BCAM) et la CAM ternaire – *Ternary CAM* (TCAM) [4]. La clé dans une BCAM est un mot binaire. Lors d'une opération de recherche, chaque cellule de la mémoire ne peut correspondre qu'à une clé unique. Dans une TCAM, les clés peuvent inclure un troisième état qui signifie «peu importe». Ainsi, plusieurs clés recherchées peuvent toutes correspondre à une même entrée dans la mémoire TCAM. Dans ce document, on utilise le terme CAM pour référer au premier type, soit la BCAM.

L'inconvénient des CAM est leur coût élevé en ressources matérielles, qui découle du besoin d'une mémoire SRAM et d'un circuit complet de comparaison. En effet, une CAM peut être composée d'un comparateur pour chaque entrée dans la mémoire afin de pouvoir effectuer la recherche en parallèle et de retourner le résultat en un cycle. Par exemple, à la Figure 2.3, une clé composée de quatre bits identifiée par c_3, c_2, c_1 et c_0 est comparée avec les bits de chaque emplacement de 1 à 16. Un encodeur retourne l'adresse de l'emplacement qui correspond à la clé afin de récupérer la valeur dans une mémoire RAM à part.

Les mémoires CAM sont souvent utilisées dans les ASIC pour obtenir la fonctionnalité de recherche dans une table, une opération essentielle pour les commutateurs réseau. Par contre, les puces FPGA n'incluent pas ce type de mémoire. Ainsi, pour utiliser une mémoire CAM sur FPGA, il est nécessaire de reproduire le comportement d'une mémoire associative à l'aide

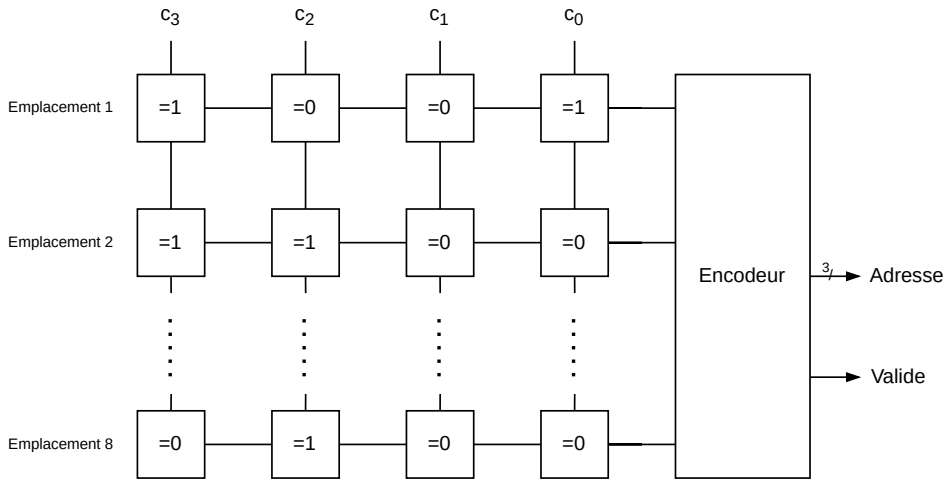


FIGURE 2.3 Exemple d'une mémoire CAM binaire

des blocs de base sur FPGA, tels que les tables de conversion – *Lookup Table* (LUT) et les éléments de mémoire comme les bascules ou les blocs RAM (BRAM). Les implémentations de CAM sur FPGA vont chercher à optimiser un ou plusieurs critères, tels que l'efficacité énergétique et la consommation de ressources [20, 21].

2.2.2 Tables de hachage

Il est intéressant d'émuler le fonctionnement d'une CAM en empruntant les principes des tableaux associatifs en logiciel, soit en utilisant une table de hachage ou un arbre de recherche.

Comme une CAM, une table de hachage opère sur des paires clé-valeur et forme un tableau associatif. Toutefois, alors qu'une CAM est une implémentation matérielle, la table de hachage est une structure de données qui peut se prêter à une implémentation logicielle ou matérielle. Une table de hachage fonctionne en calculant le code de hachage d'une clé pour être utilisé comme indice. La paire clé-valeur est stockée à cet indice dans la table. Une considération importante est la méthode de résolution des collisions, c'est-à-dire quoi faire lorsque deux clés ont le même code de hachage.

Deux types de méthodes existent pour gérer l'insertion de clés lors d'une collision, soit la méthode du chaînage et la méthode de l'adressage ouvert. La méthode du chaînage consiste à utiliser une liste chaînée pour chaque emplacement dans la table. Lors d'une collision entre deux clés, la seconde clé sera ajoutée à la suite de la première dans la liste chaînée de cet emplacement. Lors d'une recherche, la liste chaînée doit être parcourue pour retrouver la clé correspondante. Cette méthode se prête bien à une implémentation logicielle, mais elle est

peu propice pour une implémentation dans un plan des données matériel, car la recherche peut prendre plusieurs cycles selon la longueur de la chaîne. L'adressage ouvert consiste à trouver un emplacement alternatif dans la table lorsqu'il y a une collision entre deux clés. Lorsqu'une recherche est effectuée, il faut sonder potentiellement plusieurs emplacements pour trouver la clé correspondante. Différentes méthodes de sondage existent. Par exemple, le sondage linéaire place une clé qui est en collision dans l'emplacement libre suivant. La méthode de l'adressage ouvert est mieux adaptée à une implémentation matérielle, puisqu'il est possible d'utiliser plus efficacement la mémoire sans avoir à modifier dynamiquement la taille de la table. Cela permet d'atteindre un facteur de compression plus élevé pour une table de taille fixe.

2.2.3 Hachage coucou

Le hachage coucou est une approche à la résolution de collisions selon la méthode d'adressage ouvert. La particularité est l'utilisation de plusieurs tables et fonctions de hachage comme emplacements alternatifs pour les clés qui provoquent une collision [22]. L'exemple de base est une table de hachage coucou qui utilise deux tables et deux fonctions de hachage. Lors de l'ajout d'un élément, son code de hachage est calculé avec les deux fonctions de hachage simultanément et l'élément est ajouté dans la première ou la deuxième table si la case est vide. Dans le cas d'une collision, un élément dans une des deux tables est enlevé et placé dans l'autre table. L'algorithme continue récursivement jusqu'à ce que tous les éléments soient placés. Dans le cas où il est impossible de placer tous les éléments avec ces fonctions de hachage, il faut générer deux nouvelles fonctions de hachage et reconstruire la table de hachage au complet. L'intérêt principal du hachage coucou est qu'il permet la recherche d'un élément dans un temps constant dans le pire des cas.

Dans l'exemple à la Figure 2.4, la clé c_1 est utilisée pour effectuer une recherche dans un système à deux tables. Les fonctions de hachage h_1 et h_2 retournent les codes de hachage qui servent d'indices dans chaque table respective. Une comparaison est nécessaire par table, afin de vérifier que la clé se trouve dans la table.

Une généralisation du hachage coucou consiste à utiliser plus de deux tables et fonctions de hachage, ce qui permet plus de positions alternatives lors de l'ajout d'un élément [23]. Une autre amélioration est l'ajout d'une mémoire alternative de taille constante (en pratique trois ou quatre éléments) qui peut stocker les éléments avec une haute probabilité de provoquer une reconstruction de la table de hachage [24].

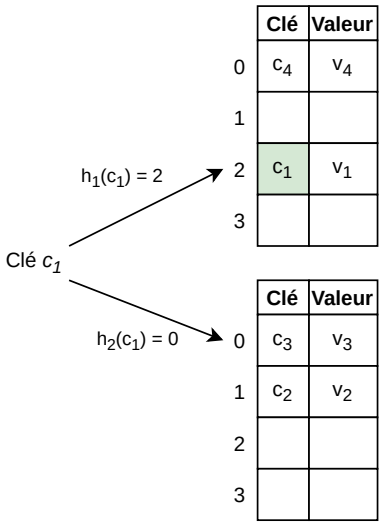


FIGURE 2.4 Recherche de la clé c_1 dans une table de hachage coucou composée de deux fonctions de hachage

Le hachage coucou est utilisé dans plusieurs implémentations de tables de hachage sur matériel. Par exemple, Bosshart et al. utilisent le hachage coucou pour implémenter la comparaison exacte dans les unités de comparaison-action du modèle RMT [2].

Liang et al. proposent une implémentation sur FPGA d'une base de données pour stocker des paires clé-valeur en utilisant le hachage coucou [25]. Leur implémentation utilise quatre tables pour le hachage coucou et quatre fonctions de hachage. Les clés sont enregistrées dans les tables de hachage avec comme valeur associée une adresse vers une mémoire RAM. Ainsi, les adresses permettent d'aller récupérer la réelle valeur voulue dans une mémoire RAM qui est partagée entre les tables. Le système peut recevoir une nouvelle requête de recherche à chaque cycle. Lorsqu'une insertion est en conflit avec les quatre tables, alors la nouvelle clé est insérée dans une table et la clé présente est retournée au début du pipeline afin de tenter de l'insérer à son tour. Ainsi, une requête d'insertion ne bloque pas le pipeline pour la recherche.

Pontarelli et al. proposent un système qui utilise le hachage coucou pour stocker des paires clé-valeur dans une mémoire externe DRAM [26]. Par contre, la recherche d'une clé nécessiterait plusieurs accès mémoires afin de chercher dans chaque table du hachage coucou. Afin d'éviter cela, ils proposent d'ajouter un filtre de Bloom sur la mémoire interne pour trouver la table qui doit être accédée en mémoire externe.

2.2.4 Autres approches

Yang et al. proposent FASTHash comme système de tables de hachage pouvant exécuter un nombre choisi de requêtes en parallèle, ce qui permet d'atteindre un plus grand débit [27]. Afin de résoudre les collisions, chaque table comporte plusieurs emplacements pour stocker les clés qui causent collision. En pratique, ils utilisent de deux à quatre emplacements alternatifs par entrée dans une table.

L'implémentation de la comparaison exacte dans un commutateur logiciel peut utiliser d'autres techniques de hachage qui sont mieux adaptées aux contraintes des systèmes avec un processeur à usage général. La capacité mémoire est moins limitée, par contre la cache du processeur doit être prise en compte dans la conception d'une table de hachage pour logiciel.

Pour les tables de comparaison ternaire et de plus long préfixe, d'autres approches sont mieux adaptées. Par exemple, l'utilisation de différentes structures de données pour concevoir un engin de recherche LPM, telles qu'une table de hachage et des arbres de recherche, permet d'obtenir une solution sur mesure pour IPv6 [28].

2.3 Langages pour la programmation du plan des données

L'arrivée des commutateurs programmables a poussé à la création de langages dédiés – *Domain-Specific Language* (DSL) pour programmer ces architectures. Parmi ces langages, le plus influant aujourd'hui est le langage P4, qui est abordé tout au long de ce travail. Ces langages sont bâtis sur les mêmes abstractions que les architectures programmables décrites plus haut, tels que le parsage d'un paquet pour y extraire ses en-têtes et le traitement sur les en-têtes par une série de tables de comparaison-action. Contrairement aux langages de programmation à usage général, les langages dédiés pour les applications réseau permettent d'exprimer plus facilement les manipulations sur un paquet.

Les prochaines sous-sections présentent les langages de programmation de plan des données P4, Domino et NPL.

2.3.1 Le langage P4

Le langage P4 a été proposé en 2014 pour décrire le traitement sur des paquets dans le plan des données d'un commutateur réseau [3]. C'est le langage qui reçoit aujourd'hui le plus d'intérêt autant en milieu académique qu'en industrie pour la programmation du plan des données. Suite à sa présentation initiale, le langage P4 a évolué pour abstraire le langage de l'architecture (la transition de $P4_{14}$ à $P4_{16}$), le rendant encore plus indépendant de la

cible d'exécution. Le langage est maintenant normalisé et sa spécification est publiée par l'organisation P4, qui fait partie de la *Open Networking Foundation* (ONF). Les trois objectifs visés par le langage P4 sont :

1. La possibilité de modifier le programme d'un équipement réseau même une fois déployé
2. L'indépendance du langage par rapport aux protocoles réseaux
3. L'indépendance du langage par rapport à la cible d'exécution

P4 est un langage dédié qui a été conçu spécialement pour l'application du traitement de paquets dans le plan des données. Contrairement à un langage de programmation à usage général qui est traduit en assembleur pour être exécuté sur un processeur à usage général, P4 cible une architecture conçue pour le traitement de paquets. C'est l'architecture de commutateur indépendant des protocoles réseau – *Protocol Independent Switch Architecture* (PISA) qui est exprimée par P4. PISA est composée de trois blocs programmables, soit un module pour l'extraction des en-têtes de paquet (*parser*), une série de tables de comparaison-action et un agrégateur d'en-têtes (*deparser*). Le langage P4 inclut des mots clés spécifiques pour référer à ces différents blocs.

Le langage P4 est construit à partir de ces abstractions de parseur et table de comparaison-action ; par contre, le langage se veut indépendant de la cible d'exécution. Différentes architectures de commutateurs réseau peuvent être définies avec P4. Le langage peut aussi cibler différentes plateformes, matérielles et logicielles.

Le langage P4 fixe les constructions de base qu'il est possible d'utiliser pour exprimer des algorithmes dans le plan des données. Par contre, il peut être avantageux d'implémenter certaines fonctions complexes à l'extérieur de P4, vu certaines limites imposées par le langage. De telles fonctions externes sont supportées par P4 par le mot clé *extern*. Certaines fonctions externes sont standardisées dans l'architecture PSA, telles que des fonctions de somme de contrôle ou de hachage, qui sont essentielles pour le traitement de paquets dans le plan des données et qui se prêtent moins bien à P4 vu le besoin de calculer la somme de contrôle sur le corps du paquet. Un fabricant peut aussi offrir des fonctions externes propres à sa cible matérielle.

Le langage P4 se veut indépendant des cibles d'exécution et par ce fait être compatible avec différentes architectures de commutateurs ou autres équipements réseau. Même si cet objectif pourrait être impossible à atteindre complètement, le langage a subi une évolution majeure qui le rapproche de ce but, soit la transition de $P4_{14}$ à $P4_{16}$. Lors de la transition de $P4_{14}$ à

$P4_{16}$, l'architecture PISA imbriquée dans $P4_{14}$ a été séparée de la définition du langage et est devenue l'architecture de commutateur portable – *Portable Switch Architecture* (PSA). PSA fait partie des spécifications publiées par l'organisation P4 [11].

2.3.2 Le langage Domino

Le langage Domino a été proposé en 2016 par Sivaraman et al. [29]. Contrairement à P4 ou NPL, le langage n'a pas été développé en parallèle avec un commutateur matériel programmable. Domino n'est à ce jour compatible avec aucune cible matérielle. Le but du langage n'est pas d'être une solution complète pour programmer un plan des données, mais cible un problème particulier, soit l'expression des algorithmes sur des paquets et le traitement à états.

Le langage Domino diffère de P4 et de NPL par sa concentration sur l'aspect du traitement de paquets, soit l'équivalent des tables de comparaison-action en P4. Le langage Domino n'intègre pas de concepts tels qu'un parseur de paquets, il est simplement assumé que les paquets arrivent déjà analysés.

La base du langage Domino est ce qui se nomme une transaction de paquet (*packet transaction*), soit un bloc de code séquentiel et atomique qui exprime un traitement à effectuer sur un paquet. Un programme complet pour un plan des données est écrit à l'aide de ces blocs de code. Le langage est similaire au langage de programmation C avec les mêmes restrictions que P4, soit l'impossibilité d'avoir des boucles non bornées, et pas de pointeurs.

De l'autre côté, une cible d'exécution est composée d'atomes, des unités de traitement qui peuvent représenter différentes opérations. Un modèle logiciel, nommé Banzai, permet de simuler un commutateur réseau programmable comportant ces atomes [29].

Un compilateur fait correspondre les blocs de code haut niveau vers ces atomes. Si le compilateur n'arrive pas à traduire un programme, alors cela veut dire qu'il n'est pas possible de l'implémenter à la vitesse du lien de la cible.

Le langage Domino et le modèle Banzai sont utilisés pour réaliser un ordonnanceur de paquets programmables [30]. C'est une partie du plan des données qui n'est pour l'instant pas programmable en P4.

2.3.3 Le langage NPL

Le langage NPL, un acronyme pour « langage de programmation réseau » – *Network Programming Language* (NPL), a été annoncé en 2019 par la compagnie Broadcom [31]. C'est un langage pour programmer le plan des données qui est similaire à P4. La spécification du langage est publique et un environnement pour la simulation de programmes NPL est aussi rendu disponible [32]. Le langage NPL permet de programmer les puces Trident 4 et Jericho 2 de Broadcom, mais peu d'information publique est disponible sur l'architecture interne de ces commutateurs programmables.

Les mêmes abstractions de base sont présentes dans le langage NPL que dans P4. Un parseur permet de déclarer les en-têtes valides et de les extraire d'un paquet. Le coeur du traitement est fait à l'aide de tables de comparaison-action. Un module pour ajouter, retirer ou modifier des en-têtes permet de réaliser la même fonction que le déparseur. Le langage NPL permet aussi de définir des fonctions spéciales qui permettent l'utilisation de blocs matériels personnalisés dans le pipeline. Une différence de NPL par rapport à P4 est que NPL offre un peu plus de flexibilité dans le création du flot de contrôle. Par exemple, une table de comparaison dans NPL peut être appelée plusieurs fois sur le même paquet, une fonctionnalité qui n'est pas possible dans P4 et qui doit être imitée en utilisant deux tables de comparaison.

2.4 Compilation, exécution et contrôle de programmes P4

Outre la définition du langage P4, il est important de connaître les autres éléments qui rendent possible la programmation du plan des données au sein d'un réseau défini par logiciel. Les sous-sections suivantes présentent le compilateur, le simulateur et l'interface avec le plan de contrôle.

2.4.1 Le compilateur de référence P4c

La structure du compilateur de référence du langage P4 est divisée en trois parties qui sont la partie frontale (*frontend*), la partie intermédiaire (*midend*) et la partie finale (*backend*).

La partie frontale est responsable de l'analyse lexicale, syntaxique et sémantique du code source P4. Le code source est traduit dans une représentation intermédiaire qui est utilisée à l'intérieur du compilateur pour les prochaines étapes de compilation. La partie intermédiaire contient une bibliothèque d'optimisations qui peuvent être exécutées ou pas sur la représentation intermédiaire, selon l'architecture ciblée. La partie finale dépend en entier de la cible, c'est-à-dire qu'elle est construite pour générer du code pour une cible particulière.

Le compilateur P4c de base contient plusieurs parties finales qui permettent de cibler différents langages ou architectures. La cible la plus commune pour le développement est le commutateur logiciel BMv2 qui agit comme référence pour l'exécution de code P4. Une description plus détaillée du BMv2 est donnée dans la prochaine section. Par ailleurs, le compilateur P4c permet de cibler d'autres architectures, telles que eBPF et DPDK. Ce sont des extensions au noyau d'un système d'exploitation qui permettent d'aller chercher plus de performance et de flexibilité pour faire du traitement de paquets réseau.

2.4.2 Le modèle comportemental BMv2

Le modèle comportemental BMv2 est le commutateur de référence pour le développement d'applications P4 [33]. C'est un commutateur logiciel qui ne vise pas à être performant, mais à permettre la vérification et le débogage de programmes P4. L'architecture prise en charge par BMv2 est le v1model, qui est l'équivalent du commutateur réseau intégré implicitement dans la version originale de P4, soit $P4_{14}$.

2.4.3 L'interface de contrôle P4Runtime

L'organisation P4 définit une interface de programmation d'applications – *Application Programming Interface* (API) pour la communication entre le plan de contrôle et le plan des données. Le commutateur réseau est équipé d'un serveur d'appel de procédure à distance – *Remote Procedure Call* (RPC) et utilise un format standard pour la spécification de l'interface. Ainsi, la façon de communiquer avec le plan des données ne change pas même si le programme P4 est modifié. Le compilateur P4c est toutefois nécessaire pour générer la description des éléments présents dans le programme P4, tel que les identifiants des tables, les paramètres des actions et autres attributs du programme.

L'API permet au contrôleur d'interagir avec les tables de comparaison-action du commutateur réseau avec des fonctions entre autres pour ajouter, modifier ou retirer une entrée. Un ensemble de fonctions permet de contrôler de cette façon différentes parties d'un plan des données défini en P4.

2.4.4 Compilation vers FPGA

2.4.4.1 P4FPGA

Wang et al. ont proposé P4FPGA en 2017 comme solution pour développer et évaluer des applications P4 sur FPGA [34]. C'est, à ce jour, la seule solution complète et à code source ouvert qui permette la compilation et l'exécution de programmes P4 sur FPGA. P4FPGA utilise le compilateur de référence P4c pour l'analyse du code source et combine une nouvelle partie finale du compilateur pour la génération de code de description matérielle. Le langage utilisé pour programmer les modules matériels et pour la génération de code est le langage Bluespec [35]. Un compilateur à code source ouvert fourni par Bluespec convertit le code matériel de haut niveau vers une représentation en Verilog [36]. Cette représentation est ensuite reconnue par les outils de synthèse FPGA de Xilinx et d'Intel.

2.4.4.2 P4-SDNet

En complément au système de traitement de paquets SDNet présenté à la section 2.1.3.2, Xilinx offre aussi un compilateur P4 vers SDNet, nommé P4-SDNet [37]. P4-SDNet permet de convertir un programme P4 écrit pour une architecture prise en charge par le compilateur en une spécification SDNet. Les outils de SDNet peuvent ensuite générer et instancier un plan des données sur FPGA.

P4-SDNet prend en charge les différents types de tables de comparaison présents dans P4 par défaut, soit la comparaison exacte, la comparaison ternaire et la comparaison LPM. Les mémoires CAM et TCAM sont implémentées en utilisant les blocs de propriété intellectuelle – *intellectual property* (IP) de Xilinx. Certaines restrictions existent autour des tables. Par exemple, pour la table de comparaison exacte, la clé doit avoir entre 12 et 384 bits, la valeur stockée doit avoir au maximum 256 bits et au maximum 512 k éléments peuvent être contenus dans la table.

L'architecture utilisée par P4-SDNet est très similaire à l'architecture de commutateur réseau qui était imbriquée dans la première version de P4, $P4_{14}$. C'est la même architecture que PISA avec un parseur, un pipeline d'unité de comparaison-action et un déparseur.

Le compilateur P4-SDNet est utilisé par la plateforme NetFPGA pour prototyper des programmes P4 sur FPGA [38].

2.4.4.3 Autres approches

Cao et al. ont proposé en 2020 un environnement pour la conversion de P4 vers VHDL et le développement de plans des données sur FPGA [39]. L'architecture est composée des blocs de PISA qui sont un parseur, un pipeline d'unités de comparaison-action et un déparseur. Les auteurs ont créé une bibliothèque de gabarits configurables en VHDL qui sert à l'instanciation des modules de l'architecture, tels que la mémoire CAM et SRAM pour former une table dans une unité de comparaison-action. Ils utilisent le compilateur de référence P4c pour l'analyse du code source P4 et la génération d'une représentation intermédiaire. Leur compilateur prend en charge d'ordonnancer les étapes dans le parseur et dans le pipeline de comparaison-action, d'apporter des optimisations, d'associer les blocs de haut niveau vers les gabarits des modules matériels et de générer le code VHDL.

Yazdinejad et al. ont proposé aussi en 2020 un compilateur et un modèle pour l'exécution de programmes P4 sur FPGA [40]. Leur architecture proposée est basée sur l'abstraction de comparaison-action afin d'implémenter le pipeline de traitement. Ils utilisent aussi le compilateur de référence P4c pour la partie frontale de la compilation, mais leur compilateur génère du code C++. Le code C++ est ensuite converti en description matérielle par l'outil de synthèse haut niveau – *High-Level Synthesis* (HLS) de Vivado. Leur implémentation de la mémoire CAM utilise une approche avec hachage basée sur le travail de Dhawan et DeHon [41].

La compagnie Netcope offrait une plateforme en ligne de type infonuagique pour modéliser des commutateurs réseau sur FPGA en utilisant son propre compilateur P4 vers VHDL [42]. La plateforme avait comme but d'être facile à utiliser par des développeurs logiciels qui n'ont pas nécessairement d'expertise en FPGA. Par contre, il était aussi possible d'utiliser directement le code généré par ce compilateur P4 vers VHDL pour y ajouter des modules personnalisés. Netcope ciblait des puces FPGA d'Intel et de Xilinx. Depuis que Intel a acquis la division P4 de Netcope, peu d'information publique est disponible sur le compilateur P4 vers VHDL.

2.5 Synthèse

Ce chapitre a présenté les avancées qui ont permis de rendre programmable le plan des données dans un commutateur. L'architecture et l'implémentation matérielle sont des éléments clés pour exposer une abstraction facilement programmable et performante. Pour atteindre cette performance, l'implémentation de chaque composante du plan des données programmable doit être conçue pour répondre aux besoins en latence et en débit du traitement de

paquets dans le réseau. Enfin, pour rendre cette technologie intéressante, il faut aussi un langage de programmation dédié et les outils pour faciliter la compilation et la vérification des programmes qui décrivent le plan des données.

Les chapitres suivants portent sur les travaux réalisés dans le cadre de ce mémoire afin de proposer une approche à la compilation de tables de comparaison-action du langage P4 vers une représentation matérielle pour FPGA.

CHAPITRE 3 ARCHITECTURE POUR LA RECHERCHE EXACTE SUR FPGA

La création d'un commutateur programmable performant sur FPGA requiert de concevoir avec attention les composantes individuelles qui composent le pipeline de traitement de paquets. Parmi ces composantes, on trouve la mémoire associative qui est utilisée pour implémenter les tables de recherche exacte. Ce chapitre traite de la conception d'une architecture pour la recherche dans une table, une opération essentielle à la classification de paquets et à l'implémentation des unités de comparaison-action.

3.1 Présentation du problème

Une partie centrale de l'abstraction de comparaison-action est la recherche de clés dans une table. Cette opération est très fréquente dans les réseaux afin d'appliquer des règles selon les paquets reçus. Dans ce contexte, une clé peut être composée d'un ou de plusieurs champs des en-têtes d'un paquet, tels qu'une adresse IP ou une adresse MAC. La valeur associée à la clé est l'identifiant d'une fonction à exécuter et ses paramètres, par exemple la fonction d'acheminer un paquet vers le bon port de sortie selon l'information de prochain saut. La valeur est ensuite utilisée par un module d'action qui peut modifier des champs d'en-têtes, ajouter ou enlever un en-tête et potentiellement effectuer plusieurs autres opérations. Dans ce chapitre, l'accent est mis sur la partie de recherche d'une clé dans une table.

Dans un commutateur réseau, la recherche exacte dans les unités de comparaison-action est souvent réalisée en matériel à l'aide d'une mémoire CAM. Cependant, l'inconvénient des CAM est leur coût élevé en ressources matérielles. Afin d'améliorer la capacité mémoire par rapport au coût en ressources matérielles, il est intéressant d'étudier les méthodes qui permettent d'imiter une mémoire associative à l'aide de structures de données. Cette approche alternative à l'utilisation d'une CAM dans un commutateur réseau peut être réalisée avec des structures telles que des tables de hachage [43], qui font partie de la solution proposée dans ce travail.

3.1.1 Critères de performance

Pour concevoir une solution adaptée au contexte des réseaux, il est important de connaître les besoins des pipelines de traitement de paquets. La mesure de performance des équipements réseau peut se faire sur trois aspects principaux : le débit, la latence et la capacité d'entre-

posage. D'autres critères importants sont souvent mentionnés dans le cas des commutateurs programmables, incluant le niveau de reconfigurabilité et la facilité de programmation. Nous décrivons ici ces critères de performance et les situons dans le contexte de ce travail.

3.1.1.1 Débit

Le débit d'un commutateur réseau est souvent mesuré en termes du nombre de bits pouvant être transférés par unité de temps. Les grandeurs du débit d'un commutateur sont de l'ordre des gigabits par seconde (Gbit/s) ou des térabits par seconde (Tbit/s). Le débit peut aussi être calculé par le nombre de paquets par seconde.

Un commutateur est composé d'un nombre de ports dont chacun est capable de transmettre à un certain débit. Par exemple, un commutateur réseau avec 64 ports de 100 Gbit/s peut admettre un débit total des liens de 6.4 Tbit/s. C'est le débit de traitement de la puce programmable Tofino d'Intel [14]. L'objectif visé par la puce est d'être capable de traiter au débit combiné des liens réseau du commutateur, donc dans cet exemple de 6.4 Tbit/s.

Une façon d'augmenter le débit de traitement d'une puce d'un commutateur est d'exploiter le traitement en parallèle. Une puce physique peut contenir plusieurs pipelines de traitement. Un pipeline peut être associé à une plage spécifique des ports du commutateur et ainsi être responsable de recevoir ou d'émettre des paquets sur ces ports. Cette méthode fonctionne bien pour le traitement qui ne comporte pas d'états entre les paquets. Pour le traitement qui entrepose des états, il n'est alors pas nécessairement possible de partager les états entre les pipelines. Selon l'application, une telle fonctionnalité nécessiterait de synchroniser les accès mémoire ou de partager la mémoire d'une autre façon, ce qui risquerait de causer une baisse du débit atteignable par la puce.

Avec les équipements d'aujourd'hui, on s'attend normalement à un débit de l'ordre du térabit par seconde pour les commutateurs centraux d'internet. Parmi les équipements programmables, les puces Tofino peuvent atteindre jusqu'à 12.8 Tbit/s [15]. Le débit est fortement lié à la technologie d'implémentation d'un circuit. Par exemple, il n'est certainement pas attendu d'obtenir le même débit de traitement d'un pipeline programmable sur un FPGA que sur un ASIC. Sur un FPGA, on peut s'attendre de 40 Gbit/s [44] à 200 Gbit/s [45] comme débit pour des pipelines de traitement de paquets programmables.

Dans le cas du module de recherche exacte, pour évaluer la performance en termes de débit il faut prendre en compte les opérations possibles sur la table, soit la recherche d'une paire clé-valeur et la mise à jour du contenu de la mémoire. Pour ce faire, nous utilisons les mêmes termes que Sarbishei [46].

Le critère de performance pour la recherche dans la table est le débit de recherche – *Lookup Throughput* (LT). Le même critère est appliqué pour la mise à jour de la table, soit le débit des mises à jour – *Update Throughput* (UT). De plus, l'évaluation est divisée pour les trois sous opérations de la mise à jour, soit l'ajout – *addition* (A), la modification – *modification* (M) et le retrait – *deletion* (D). L'ajout est l'insertion d'une nouvelle paire clé-valeur dans la table. La modification est seulement un changement de la valeur dans la table associée à une clé. Le retrait est l'invalidation d'une paire clé-valeur dans la table ce qui rend l'espace disponible pour un ajout. Les acronymes utilisés sont résumés dans le Tableau 3.1.

3.1.1.2 Latence

La latence dans un commutateur réseau est le temps passé entre l'arrivée d'un paquet et sa sortie. Plusieurs étapes avec un rôle particulier sont présentes dans le pipeline du commutateur. Un paquet doit passer par le parseur pour y extraire ses en-têtes avant de pouvoir être traité par le pipeline d'unités de comparaison-action. Chaque étape augmente la latence de traitement d'un paquet. Afin de pouvoir programmer des opérations plus complexes, plusieurs tables sont utilisées une à la suite de l'autre pour effectuer des recherches sur différents champs d'en-têtes d'un paquet ou sur des métadonnées internes. Une recherche dans une table étant effectuée dans chaque unité, il est avantageux de chercher à avoir une latence courte pour cette opération commune.

Outre le fait qu'une plus faible latence soit souhaitable en termes de performance, il est aussi important de connaître les fluctuations sur la latence. Un commutateur réseau doit pouvoir assurer un temps de traitement en pire cas, ce qui implique que la latence doive être bornée supérieurement en pire cas. Cela permet à un opérateur de garantir les performances de son réseau pour respecter des contraintes de qualité de service ou des spécifications de certains protocoles. C'est pour cette raison que la complexité du traitement dans le plan des données est limitée et ne permet pas de structures itératives.

Comme pour le débit, la latence des opérations sur la table de recherche exacte est évaluée de la même façon. La latence est évaluée selon la latence de recherche – *Lookup Latency* (LL) et la latence de mise à jour – *Update Latency* (UL), qui est aussi divisée en ses sous-opérations. Les acronymes pour la latence sont aussi résumés dans le tableau 3.1.

	Recherche	Mise à jour		
		Ajout	Modification	Retrait
Latence	LL	ULA	ULM	ULD
Débit	LT	UTA	UTM	UTD

TABLEAU 3.1 Termes utilisés pour qualifier la latence et le débit des opérations d'un module de recherche exacte

3.1.1.3 Capacité d'entreposage

Un commutateur réseau a différents besoins en mémoire selon la fonction de chaque composante. Par exemple, l'ordonnanceur de trafic a besoin d'une mémoire tampon pour stocker le corps des paquets le temps que leurs en-têtes soient traités par le pipeline de comparaison-action. De son côté, le pipeline de traitement a besoin de mémoires associatives pour implémenter les tables de recherche.

Dans ce travail, les mémoires qui nous intéressent sont celles disponibles à partir du pipeline de comparaison-action. Parmi celles-ci on trouve la mémoire SRAM et la mémoire associative de type CAM et TCAM. La mémoire TCAM sert aux tables utilisant la comparaison ternaire ou du plus long préfixe. La mémoire CAM est utilisée pour la comparaison exacte. Cependant, il est aussi possible d'utiliser la mémoire SRAM pour émuler le fonctionnement d'une CAM à l'aide de structures de données, telles qu'il est proposé dans ce mémoire. Il faut alors comparer la capacité disponible pour stocker des règles dans les tables de comparaison et le coût en utilisation de ressources pour implémenter ces tables. Dans un FPGA, ces ressources peuvent être les LUT, les bascules et les blocs de mémoire. De plus, les tables de hachages doivent prendre en compte la possibilité de collisions entre les clés, ce qui peut diminuer la capacité qui est possible d'atteindre dans la table. La capacité utilisable d'une table de hachage varie selon plusieurs facteurs comme la taille de la mémoire, la largeur des clés, le type de fonction de hachage et le taux d'occupation de la table.

3.1.1.4 Reconfigurabilité et programmabilité

Outre les métriques de performance, il faut aussi prendre en compte les fonctionnalités et la configurabilité offertes par le système. Avec le plan des données programmable, les tables faisant partie des unités de comparaison-action doivent pouvoir être de taille variable et fonctionner avec différentes largeurs de clé et de valeur. Un module de table de recherche exacte doit alors être générique afin de répondre à ces besoins et de fonctionner peu importe le choix des champs d'en-êtes à comparer.

3.1.2 Traitement avec états

Un autre aspect à prendre en compte dans les besoins des tables de comparaison-action est le traitement avec états. C'est une fonctionnalité essentielle pour l'identification de flot dans le plan des données. Un flot est un ensemble de paquets réseau qui découle d'une même application et partage certains champs dans leur en-tête. Une façon très commune d'identifier un flot est par la combinaison de cinq éléments présents dans les en-têtes des paquets, soit l'adresse IP source, l'adresse IP de destination, le port source, le port de destination et le protocole de la couche transport (TCP ou UDP). Une fois un flot identifié, le plan des données peut avoir besoin d'enregistrer des informations concernant l'état de ce flot. Pour ce faire, des éléments à mémoire sont nécessaires afin de faire persister l'information à propos d'un flot entre l'arrivée des paquets de ce flot. Le traitement avec états et l'identification de flots sont utiles à plusieurs applications réseau, tels qu'un pare-feu ou un équilibreur de charge.

Le traitement avec états dans un commutateur programmable est un sujet très étudié en ce moment [47–49]. Le besoin d'atteindre un débit élevé dans les commutateurs réseau impose une forte contrainte sur la complexité du traitement qu'il est possible d'appliquer sur les paquets. Le concept d'état tel que la conservation d'un état entre les paquets d'un flot est problématique pour le plan des données, car une dépendance de données peut exister entre deux paquets. Dans ce cas, lorsqu'un état est accédé par plusieurs paquets, il faut assurer de respecter l'ordre d'accès à cet état. Par exemple, si deux paquets consécutifs partagent le même état, alors tant que le premier paquet n'a pas complété son écriture dans l'état, le deuxième paquet ne peut pas le lire pour commencer son traitement. Cela veut dire que le pipeline doit bloquer pour le deuxième paquet, ce qui a comme conséquence d'augmenter la latence de ce paquet.

Ce problème est accentué par la hiérarchie des mémoires, c'est-à-dire que les mémoires avec la plus grande capacité, telles que les mémoires DRAM, sont celles avec le plus long temps d'accès. Chaque opération dans les unités de comparaison-action nécessitant de lire ou d'écrire un état a pour effet d'augmenter la latence. De plus, un temps d'accès variable causé par les mémoires externes est un inconvénient de plus pour assurer le temps de traitement en pire cas. Il existe alors un compromis entre la taille de la mémoire utilisable et la latence qui est admise ou la complexité des algorithmes qu'il est possible de programmer sur le plan des données. Sur un FPGA, les deux choix sont possibles, soit entre utiliser la mémoire interne ou la mémoire externe.

Dans le contexte de ce travail, on peut considérer qu'une table en P4 est en fait un élément avec états. Une table dans une unité de comparaison-action contient des règles qui persistent entre paquets, donc qui conservent un état. Par contre, les tables de comparaison ont la

propriété d'être accessibles en écriture seulement par le plan de contrôle tandis que le plan des données ne peut qu'en faire la lecture. Alors, pour qu'une règle soit ajoutée dans la table selon le contenu d'un paquet, il faut que le paquet soit envoyé au plan de contrôle. Cela a pour conséquence d'augmenter la latence vu la communication avec le plan de contrôle. L'architecture de recherche exacte proposée dans ce travail permet de faire l'insertion de règles en matériel, ce qui permet l'écriture dans la table à partir du plan des données. Par exemple, cela est utile pour l'application de l'apprentissage des adresses MAC. Lorsqu'un nouveau paquet arrive dans un commutateur et que son adresse MAC n'est pas présente dans la table, il serait possible de l'insérer directement à partir du plan des données. Cela évite une interaction avec le plan de contrôle.

3.2 Description de l'architecture proposée

Dans cette section, il est question du fonctionnement, de la conception et de l'implémentation du module de recherche exacte. D'abord, l'architecture générale est présentée, suivie par l'interface pour interagir avec le module et de différents cas d'utilisation. Ensuite, l'implémentation matérielle du module de recherche exacte est présentée.

3.2.1 Architecture

Le principe du hachage coucou, tel que vu dans la revue de littérature, est central au fonctionnement de l'architecture de recherche exacte qui est proposée dans ce travail. La fonctionnalité de recherche d'une clé dans une table est réalisée à l'aide d'une table de hachage basée sur le hachage coucou.

L'idée centrale est d'avoir un système composé de plusieurs sous-tables de hachage et d'utiliser un mécanisme pour distribuer les clés dans ces sous-tables lors de la résolution de collision. Le système est composé d'au moins deux sous-tables de hachage qui servent d'emplacement pour les clés. Si une clé entre en collision avec une clé déjà présente dans la première sous-table, alors elle peut être placée dans la deuxième sous-table et ainsi de suite.

Pour réaliser cela, chaque sous-table est indépendante des autres et est adressée par sa propre fonction de hachage. Les sous-tables fonctionnent effectivement comme une table de hachage et leur contenu est adressé par le code de hachage calculé à partir de la clé. Par contre, la différence est qu'une sous-table ne possède pas de mécanisme de résolution de collision. Si une clé à insérer entre en conflit avec une clé déjà présente dans la sous-table, alors une sous-table n'a pas de méthode pour accommoder les deux clés. C'est au niveau du système composé de plusieurs sous-tables de hachage que la résolution de conflit a lieu.

Ainsi, chaque sous-table de hachage est composée d'une fonction de hachage et d'une mémoire vive statique pour y stocker les paires clé-valeur. De plus, chaque sous-table utilise une fonction de hachage différente de celles des autres sous-tables. Pour une clé en entrée dans le module, différents indices sont utilisés pour l'emplacement potentiel de cette clé dans chaque sous-table.

Le système inclut aussi une mémoire CAM utilisée en parallèle aux sous-tables de hachage. La mémoire CAM est implémentée par-dessus les composantes programmables du FPGA. N'utilisant pas de hachage, il ne peut pas y avoir de collision entre deux clés dans la mémoire CAM. Par contre, sa capacité est plus petite vu le coût plus élevé en ressources matérielles. La CAM est utilisée comme emplacement secondaire lorsqu'il n'est pas possible d'insérer une clé dans aucune des sous-tables.

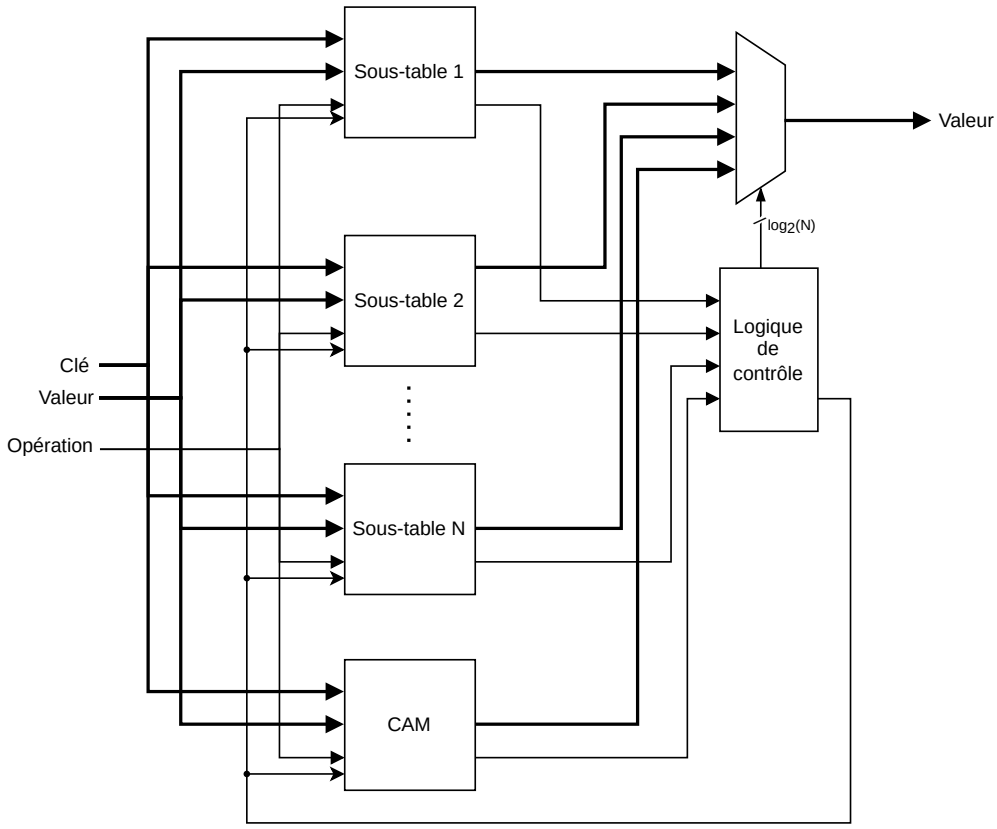


FIGURE 3.1 Vue d'ensemble simplifiée de l'architecture proposée

La Figure 3.1 montre une vue haut niveau du système en entier. Un nombre configurable de sous-tables, identifiées de 1 à N, compose l'espace mémoire principal pour stocker les paires clé-valeur. Une mémoire secondaire CAM sert d'emplacement alternatif pour les clés qui causent collisions. Les mémoires dans chaque sous-table et dans le bloc CAM peuvent

être accédées en lecture ou en écriture. En entrée, une clé doit être fournie pour la recherche ou l'insertion. Dans le cas de l'insertion, la valeur associée doit aussi être fournie en entrée. Dans le cas d'une recherche, si la valeur est présente dans une des sous-tables ou dans la CAM, elle est retournée en sortie avec un signal de validité. Un bloc de logique de contrôle implémente la résolution de collision. Ce bloc décide dans quelle sous-table insérer une paire clé-valeur ou s'il est nécessaire d'utiliser la mémoire CAM. Ce module est aussi responsable de retourner la valeur valide associée à une clé lors d'une opération de recherche.

3.2.2 Interface

Le module matériel permet la recherche et l'insertion de clés à l'aide de signaux personnalisés qui sont définis ci-dessous. Plusieurs paramètres permettent de déterminer la largeur des paires clé-valeur et de modifier l'architecture interne du module.

3.2.2.1 Entrées/sorties

L'interface du module de comparaison exacte est divisée en deux parties principales, soit une pour la recherche et une pour la mise à jour des règles. Le tout est contrôlé par une horloge commune et par un signal de réinitialisation. Les signaux d'entrées et sorties de ces deux parties sont présentés dans la Figure 3.2.

Pour la partie de recherche, les entrées sont constituées de la clé à chercher dans la table (`lookup_key`) et d'un signal de contrôle pour activer la recherche (`lookup_en`). Les sorties sont la valeur associée avec la clé recherchée (`lookup_value`) et un signal de contrôle pour indiquer si la clé a été trouvée dans la table (`lookup_hit`), indiquant ainsi que la valeur retournée est valide.

Pour la partie de mise à jour, les entrées sont la clé qui indique la paire clé-valeur sur laquelle opérer (`insert_key`), au besoin la valeur associée à écrire (`insert_value`) et deux signaux de contrôle pour activer l'insertion, la modification (`insert_en`) ou la suppression (`delete_en`). Pour les sorties, il y a un seul signal de contrôle qui indique si une erreur d'insertion s'est produite (`insert_err`), c'est-à-dire qu'il n'a pas été possible d'insérer une nouvelle paire clé-valeur.

La présence de ces deux groupes de signaux dans l'interface du module, un pour la recherche et un pour la mise à jour, permet d'exécuter simultanément ces opérations sur la table. Dans le cas où les deux types d'opérations partageraient le même port, il serait alors impossible

de garantir un rythme d'une recherche par cycle d'horloge, vu le besoin de mettre à jour les règles. Ainsi, avec un port d'accès à la table dédié pour la recherche, le module est mieux adapté pour soutenir un débit d'une recherche par cycle.

Des signaux supplémentaires peuvent être ajoutés pour donner plus d'information sur le résultat des opérations. Par exemple, dans le cas de la suppression, le module pourrait indiquer si la clé à supprimer était présente dans la table. C'est signaux ne sont pas inclus afin de simplifier l'interface et étant donné que l'état de la table est normalement conservé dans le plan de contrôle.

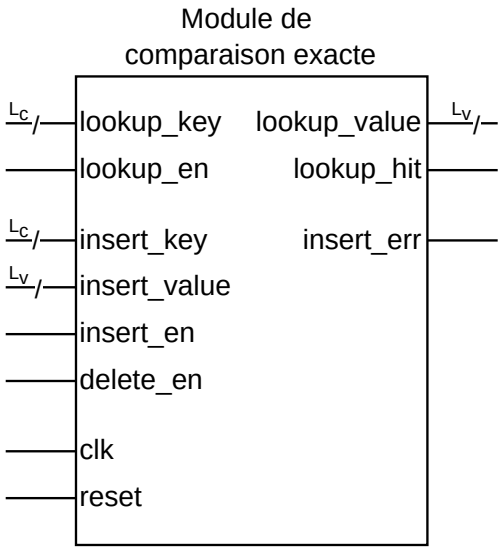


FIGURE 3.2 Interface du module matériel de comparaison exacte

3.2.2.2 Paramètres

Le module est aussi configurable à l'aide de cinq paramètres. Parmi ces paramètres, deux déterminent la largeur de la clé et la largeur de la valeur, qui ensemble constituent la donnée qui doit être stockée dans la mémoire de la table. Ces paramètres sont les suivants :

1. La largeur de la clé en bits (L_c)
2. La largeur de la valeur en bits (L_v)

De plus, trois autres paramètres influencent le nombre d'éléments que peut stocker le système, c'est-à-dire la capacité du module. Ces paramètres sont :

1. La capacité en nombre d'éléments d'une sous-table de hachage ($C_{sous\text{-}table}$)
2. Le nombre de sous-tables utilisées par le système (N)

3. La capacité en nombre d'éléments de la mémoire CAM intermédiaire (C_{CAM})

L'équation 3.1 exprime la relation entre les paramètres N, $C_{sous\text{-}table}$ et C_{CAM}.

$$\text{Capacité totale du module} = N \times C_{sous\text{-}table} + C_{CAM} \tag{3.1}$$

Certaines limites imposées par l'implémentation existent pour ces paramètres. Le nombre de sous-tables (N) doit être au minimum deux vu que la conception de l'architecture est basée sur l'utilisation de plusieurs sous-tables. La capacité d'une sous-table ($C_{sous\text{-}table}$) est exprimée par une puissance de deux, par exemple $2^{10} = 1024$ éléments. La capacité de la CAM (C_{CAM}) peut prendre une valeur quelconque et peut aussi être de zéro, ce qui implique que la CAM n'est pas instanciée.

3.2.3 Cas d'utilisation

Cette section présente les différents scénarios d'utilisation du module de comparaison exacte en rapport avec les entrées et sorties de l'interface.

3.2.3.1 Recherche d'une règle

Pour effectuer une recherche dans la table, une clé doit être fournie au module (`lookup_key`) et le signal de contrôle de recherche (`lookup_en`) doit être actif pour que l'opération soit effectuée. Les deux cas résultants suivants sont possibles :

1. Si la clé cherchée est présente dans le module, alors la valeur associée est retournée et le signal de contrôle de succès (`lookup_hit`) est actif pour indiquer que la règle a été trouvée, donc que la valeur est valide.
2. S'il n'existe pas de clé dans la table qui corresponde avec la clé en entrée, alors le signal de contrôle de succès est faux pour indiquer que la règle n'a pas été trouvée dans la table.

3.2.3.2 Mise à jour d'une règle

Ce cas d'utilisation correspond aux trois opérations de mise à jour, soit l'insertion, la modification et la suppression d'une règle. Pour insérer ou modifier une règle dans la table, une paire clé-valeur doit être fournie en entrée au module avec le signal de contrôle d'insertion actif. Les deux cas possibles sont :

1. Si une clé est déjà présente dans la table et correspond à la clé en entrée, alors la règle dans la table est mise à jour avec la nouvelle valeur qui est fournie en entrée.
2. Si la clé n'est pas trouvée dans la table, alors le module tente d'insérer une nouvelle règle constituée de la paire clé-valeur dans la table.

Pour retirer une règle, un autre signal de contrôle permet d'activer la suppression de la paire clé-valeur de la table. Il est alors seulement nécessaire de fournir la clé de la règle à retirer. Les résultats de l'opération sont :

1. Si la clé est présente dans la table, alors la règle est supprimée de la table.
2. Si la clé n'est pas présente dans la table, il n'y a aucun changement.

3.2.4 Implémentation

Comme vu précédemment, l'interface du module est divisée en deux parties principales, soit pour la recherche et l'insertion. L'implémentation de l'architecture reflète cette disposition. Une sous-table comporte aussi deux parties présentées dans cette section.

D'abord, pour la partie de recherche, la Figure 3.3 montre l'implémentation matérielle du module lorsqu'il est configuré avec deux sous-tables. La directive de recherche est envoyée à toutes les sous-tables et au bloc CAM. La recherche se fait en parallèle dans les sous-tables et dans le bloc CAM et chacun retourne un signal pour indiquer si la clé se trouve dans sa mémoire ainsi que la valeur associée si la clé s'y trouvait. La valeur trouvée est alors retournée à la sortie du module. Une paire clé-valeur ne peut pas se retrouver dans plus d'un emplacement à la fois, car le comportement du module lors d'un ajout d'une clé déjà présente dans la table est d'y modifier la valeur associée. Ainsi, seulement une sous-table ou la CAM va contenir la clé et retourner un succès. À l'intérieur d'une sous-table se trouve une instance de la fonction de hachage dédié pour la recherche.

Pour la partie de mise à jour, l'implémentation simplifiée est montrée à la Figure 3.4. Les opérations par cette interface incluent l'insertion, la modification et le retrait d'une paire clé-valeur. Les opérations se font en deux étapes. D'abord, la clé utilisée par l'opération est utilisée pour lire le contenu en mémoire de chaque sous-table. Les signaux `insert_hit` et `insert_occupied` prennent alors leur valeur. Le signal `insert_occupied` est toujours actif lorsqu'une paire clé-valeur est présente à l'emplacement recherché. Le signal `insert_hit` est actif si la clé correspondante est présente dans la sous-table. Si une opération de retrait n'est pas demandée, alors le fait que la clé soit présente indique une modification d'une règle. Le signal est transmis aux autres tables afin d'indiquer de ne pas insérer la clé dans un nouvel emplacement, vu que la paire clé-valeur est déjà présente. Si la clé n'est présente dans aucune

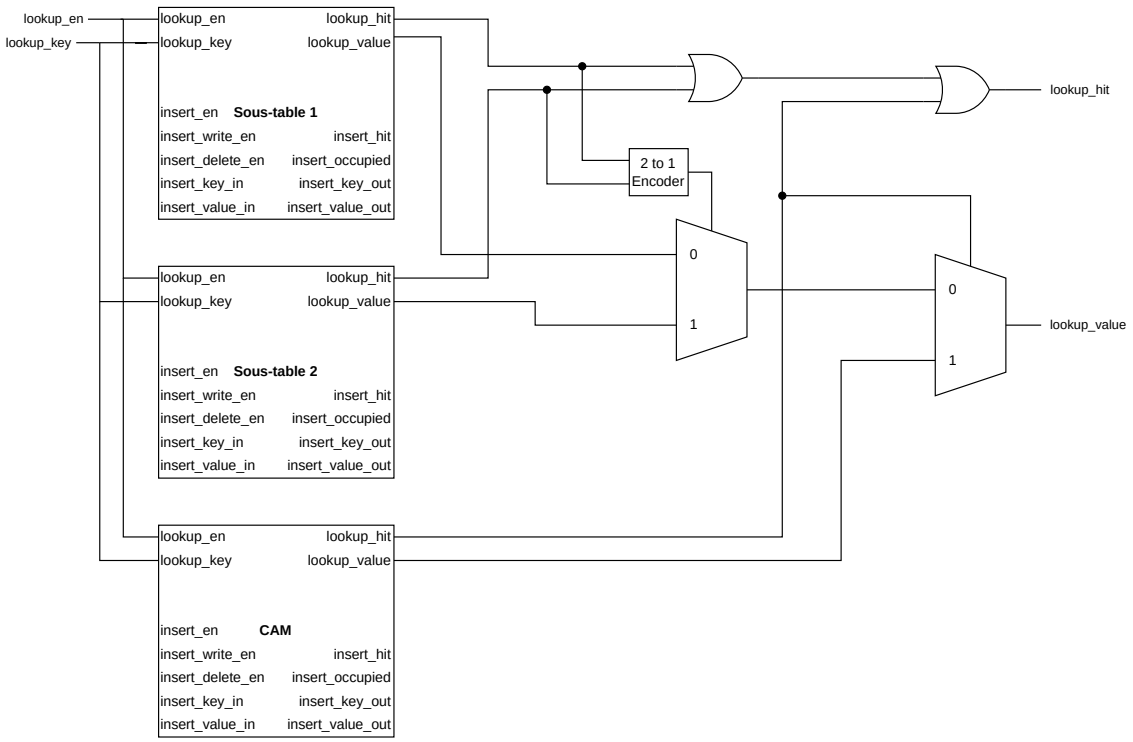


FIGURE 3.3 Implémentation de la partie recherche avec deux sous-tables

des sous-tables, alors le signal `insert_occupied` permet de sélectionner la table dans laquelle la nouvelle clé est insérée. L'ordre d'insertion se fait toujours de la première sous-table à la dernière, et par la suite dans la CAM. Ainsi, si l'espace pour une clé dans la première sous-table est déjà occupé par une autre clé et qu'on a une collision, alors `insert_occupied` sera vrai et l'insertion se fera dans une autre sous-table.

Le signal `insert_delete_en` est omis de la Figure 3.4. Ce signal est simplement transmis à partir de l'entrée du module à chaque sous-table et à la CAM et indique qu'une règle doit être supprimée au lieu d'être insérée ou modifiée.

L'insertion dans la CAM se fait principalement de la même façon. Si l'opération est pour une clé déjà présente dans la CAM, alors la paire clé-valeur dans la CAM est modifiée. Lorsqu'une clé à insérer n'est pas présente et entre en collision dans toutes les sous-tables, alors cette nouvelle clé prend la place dans la première sous-table et la paire qui est expulsée est insérée dans la CAM.

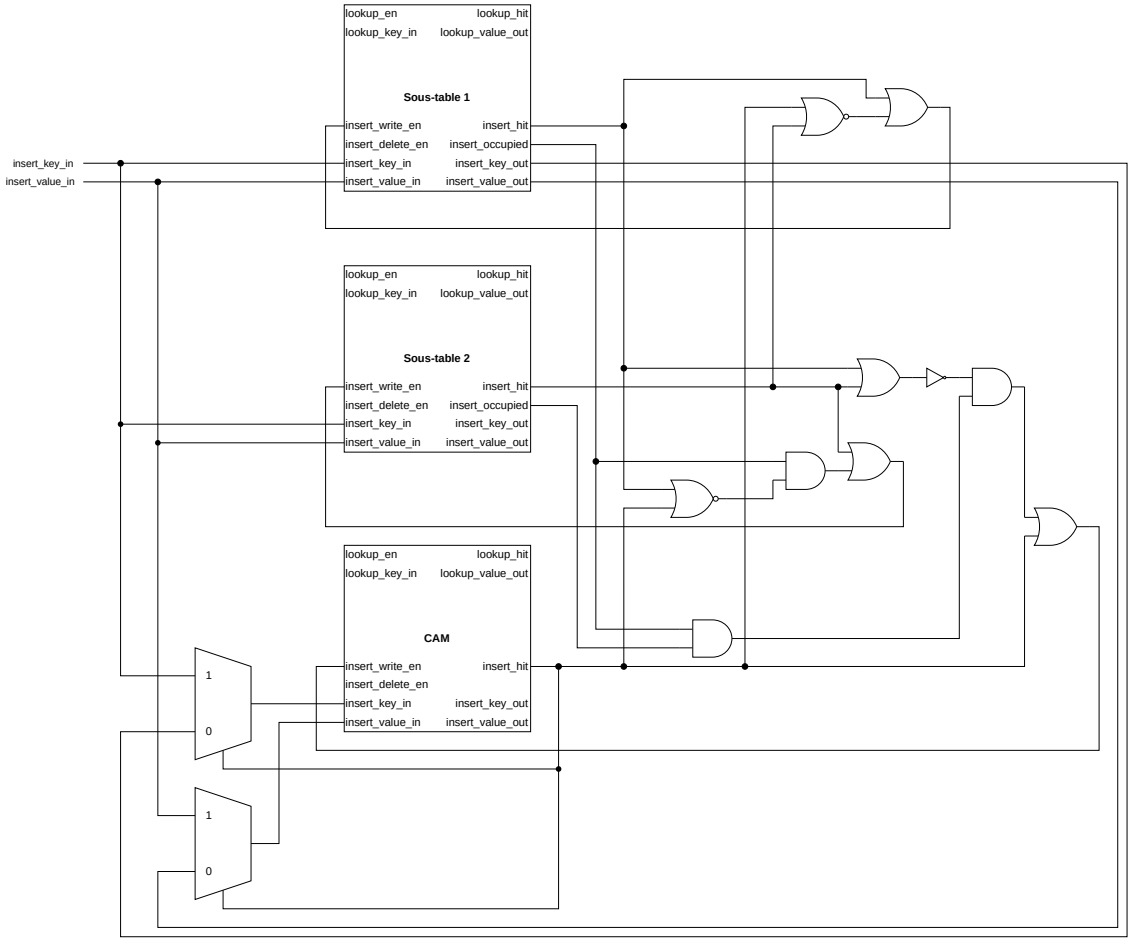


FIGURE 3.4 Implémentation de la partie insertion avec deux sous-tables

3.2.4.1 Détails d'implémentation sur FPGA

Pour réaliser la composante mémoire des sous-tables, les blocs RAM (BRAM) du FPGA sont utilisés. Les BRAM sont des mémoires SRAM intégrées aux circuits logiques du FPGA qui permet de stocker une plus grande capacité de données sans avoir recours aux composantes programmables telles que les bascules et LUT.

Il est commun pour les FPGA qui comportent des BRAM de posséder deux ports d'accès à chaque bloc. Ces ports d'accès indépendants permettent d'effectuer deux opérations par cycle, soit deux lectures, deux écritures ou une lecture et une écriture. Dans l'implémentation de l'architecture proposée, chaque sous-table est composée d'une mémoire RAM implémentée par des BRAM. Un port d'accès est dédié à la lecture afin de garantir une recherche dans la table par cycle. L'autre port d'accès est utilisé pour l'écriture dans la mémoire, c'est-à-dire l'insertion, la modification ou la suppression d'une entrée dans la table.

3.2.5 Fonction de hachage

Un élément important dans l'implémentation d'une table de hachage est le choix de la fonction de hachage. La fonction de hachage transforme la clé d'une paire clé-valeur en code de hachage. Ce code sert d'indice dans la table afin d'indiquer la position en mémoire pour lire ou écrire la paire clé-valeur.

L'objectif d'une fonction de hachage est de distribuer le plus uniformément possible la plage de clés sur l'espace mémoire disponible dans la table. Dans les tables de recherche d'un commutateur réseau, une clé peut être une adresse IP, une adresse MAC ou une combinaison de différents champs d'en-têtes d'un paquet. Par exemple, la mémoire interne d'un FPGA, qui est limitée à 52 mégabits dans la puce FPGA utilisée dans ce projet, permet de stocker autour de 413000 adresses IPv6 de 128 bits. Cela implique la transformation d'une clé de 128 bits vers un code de hachage de 18 bits afin de pouvoir adresser une mémoire qui s'implémente dans le FPGA. La transformation d'une plage de 128 bits à une plage de 18 bits implique qu'il y aura certainement des cas de collisions entre des clés. Dans une table de hachage avec des entrées qui ne sont pas connues d'avance, il est impossible de garantir qu'une fonction de hachage ne causera pas de collisions.

Toutefois, certains critères permettent de mesurer la qualité d'une fonction de hachage. Entre autres, une bonne fonction de hachage doit avoir un effet d'avalanche. C'est-à-dire que lorsqu'un bit est modifié dans la clé en entrée, les bits du code de hachage ont tous une probabilité de 50% d'être inversés. Donc même si on ne connaît pas le type de clé en entrée, on peut s'attendre à ce qu'un petit changement dans la clé génère un code de hachage complètement différent. Ne connaissant pas non plus la distribution des clés, une fonction de hachage avec cette propriété est ce qui convient le mieux pour réduire la probabilité et le nombre de collisions dans une table de hachage. Un certain type de hachage, le hachage universel, répond à ce critère de qualité. Le hachage universel est en soi une définition d'une famille de fonctions. Un aspect pratique d'utiliser une famille de fonctions est de pouvoir générer une nouvelle fonction de hachage différente pour chaque sous-table du module.

Du point de vue matériel, on cherche à traiter des paquets à haut débit avec une faible latence. Ainsi, un choix approprié de fonction de hachage doit avoir une implémentation matérielle qui peut exécuter le hachage en peu de cycles. Il n'est pas question d'utiliser de fonctions de hachage cryptographiques telles que MD5 ou SHA-1 qui peuvent nécessiter plusieurs itérations. De plus, comme dans toute conception matérielle, il faut aussi porter attention à la consommation de ressources engendrée par l'implémentation de la fonction de hachage. Dans le module proposé, la fonction est instanciée deux fois pour chaque sous-table du système. Par exemple, une fonction de hachage multiplicative demande plus de ressources

pour réaliser sur un FPGA, dont des multiplicateurs dédiés basés sur les processeurs de signal numérique – *Digital Signal Processor* (DSP). Le nombre de DSP sur un FPGA est limité et ces blocs peuvent être nécessaires pour la réalisation de la partie action dans le pipeline d'unités de comparaison-action.

Afin d'être générique, la fonction de hachage choisie doit aussi pouvoir transformer une clé d'une largeur quelconque en un code de hachage d'une autre largeur. Ainsi, contrairement aux fonctions de hachage utilisées en logiciel qui opèrent sur des types communs tels qu'un entier de 32 bits ou de 64 bits, la fonction de hachage pour réaliser une table de comparaison doit être configurable. À la compilation, connaissant la largeur des clés et des sous-tables à adresser, il faut générer différentes fonctions de hachage qui opèrent sur des clés de différentes tailles.

Ainsi, étant donné que le module proposé est composé de plusieurs sous-tables de hachage, il est nécessaire d'avoir aussi plusieurs fonctions de hachage. Chaque sous-table doit utiliser une fonction de hachage différente afin que les clés soient distribuées à des emplacements différents dans chaque sous-table. Avec les besoins présentés ci-haut, et dans l'idée de pouvoir instancier un nombre arbitraire de sous-tables dans le système, nous avons choisi d'utiliser la classe de fonctions H_3 pour générer les fonctions de hachage du système [50]. Cette classe de fonctions a la propriété d'être universelle et d'avoir une implémentation simple en matériel.

Pour une clé de n bits de long, l'équation 3.2 est le gabarit pour définir une fonction de la classe H_3. Les entrées de la fonction de hachage sont la clé à hacher et une constante q déterminée à la compilation. Dans cette formule, c_1 représente le bit le plus significatif de la clé, c_2 le deuxième bit de la clé et ainsi de suite. L'opération logique ET est appliquée entre chaque bit de la clé et une constante q. La constante q est un tableau de la même taille de la largeur de la clé. Chaque élément $q(1), q(2) \ldots q(n)$ est composé du nombre de bits qui permet d'adresser au complet la mémoire de la table.

$$h(c) = c_1 \cdot q(1) \oplus c_2 \cdot q(2) \oplus \ldots \oplus c_n \cdot q(n) \tag{3.2}$$

La constante q est ce qui permet d'obtenir différentes fonctions de hachage. Pour chaque fonction de hachage voulue, une constante q est générée en logiciel de façon aléatoire.

La classe de fonctions H_3 comporte plusieurs avantages pour ce projet. D'abord, elle est bien adaptée à une implémentation matérielle sur FPGA vu les opérations simples qui ne nécessitent que des portes logiques ET et OU exclusif, tel que montré à la Figure 3.5. De plus,

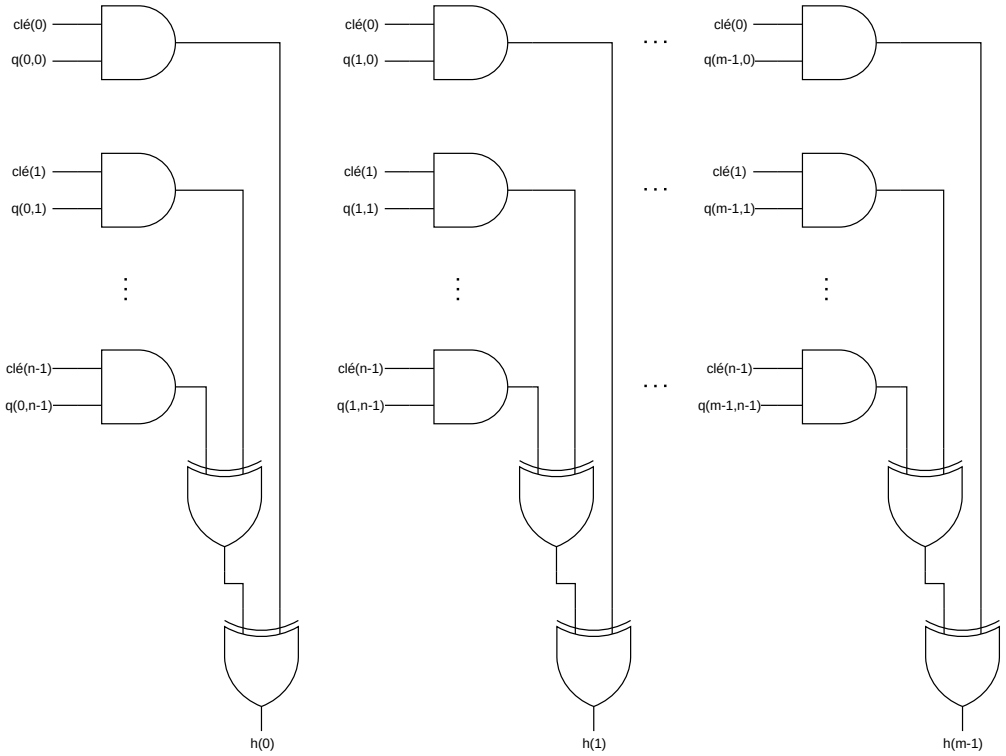


FIGURE 3.5 Implémentation de la fonction de hachage H_3

l'absence de multiplications évite d'utiliser les tranches DSP sur FPGA. Ensuite, la fonction permet de calculer le code de hachage d'une clé avec un circuit combinatoire, ce qui n'ajoute pas de cycles d'horloge à la latence des opérations sur la table.

3.3 Méthodologie de vérification

Le fonctionnement du module de comparaison exacte conçu dans ce travail est vérifié à l'aide d'une suite de tests qui accompagne le code VHDL. Deux types de tests font partie de la suite, des cas de tests simples et des scénarios complexes. Les cas de tests simples sont écrits pour chaque entité VHDL qui constitue le module de comparaison et permettent de vérifier le comportement attendu de chaque entité. Ce sont des tests unitaires qui valident une fonctionnalité à la fois. Ces tests s'exécutent rapidement et sont aussi utilisés lors des phases de développement et d'optimisation afin d'éviter des régressions. Quant aux scénarios complexes, ils sont exécutés seulement sur le système en entier et permettent de vérifier le fonctionnement et la robustesse du module. Un scénario comporte une série d'opérations qui seraient normalement effectuées sur la table, telles que des mises à jour et des recherches.

Pour les deux types de tests, les résultats sont toujours validés en comparant les signaux de sortie du module avec les signaux de référence attendus. De plus, pour certains tests, l'état interne ou le contenu en mémoire est aussi vérifié pour s'assurer que les opérations de mise à jour se propagent correctement dans le module.

Voici quelques exemples de cas de test, correspondant à des opérations sur la table, qui sont validés :

- Insertion d'une paire clé-valeur avec succès
- Modification d'une paire clé-valeur
- Retrait d'une paire clé-valeur
- Recherche d'une paire clé-valeur présente dans la table (succès)
- Recherche d'une paire clé-valeur absente de la table (échec)

Pour les tests complexes, une série d'opérations d'insertion, de modification, de suppression et de recherche est effectuée avec différents ordres. Cela permet de vérifier le comportement normal du système et le comportement lors de collision.

La nature configurable du module de comparaison exacte implique que les tests du banc d'essai sont exécutés pour une configuration spécifique. En pratique, quelques configurations sont testées afin de s'assurer que le comportement est valide même pour une instanciation avec des paramètres différents. Par exemple, faire varier la largeur de la clé ou de la valeur est peu probable de causer un problème, tandis que de tester le module avec un différent nombre de sous-tables est plus pertinent pour la vérification de l'implémentation du système. De plus, les bancs d'essai sont développés pour avoir accès aux paramètres, c'est-à-dire aux valeurs *generic* du code VHDL de l'instance sous test, ainsi il est facile de tester différentes configurations sans avoir à changer manuellement les tests.

La vérification est effectuée à l'aide du simulateur GHDL [51], du simulateur ModelSim et de l'outil cocotb [52]. GHDL est un simulateur VHDL à code source ouvert qui permet d'analyser un programme VHDL, de simuler l'exécution d'un banc d'essai et de générer en sortie un chronogramme des signaux. Cocotb est un environnement de vérification développé en Python pour la vérification de circuits numériques. L'outil interagit avec le simulateur afin de fournir une interface pour la programmation de bancs d'essai dans le langage Python. Plusieurs simulateurs propriétaires ou à code source ouvert sont compatibles avec cocotb, dont GHDL et ModelSim, les simulateurs qui sont utilisés pour la vérification dans ce projet.

L'utilisation de cocotb et de Python pour créer les bancs d'essai permet aussi de réutiliser le modèle de simulation logicielle pour la vérification. La fonction de hachage H_3 ainsi que le comportement du module de recherche exacte ont été implémentés en Python. Ce modèle est

utilisé pour la vérification automatique de chaque opération effectuée par le module matériel en comparant que le même résultat est produit par le modèle logiciel. L'utilisation du modèle logiciel est utile lors de la phase de développement afin de s'assurer que l'ajout d'optimisations n'affecte pas le fonctionnement du module matériel.

3.4 Résultats et analyse

Les différents résultats obtenus à partir de l'architecture de recherche exacte sont présentés ci-dessous.

3.4.1 Exploration des paramètres

Dans cette section, une exploration des différents paramètres de l'architecture est présentée. Les différentes expériences ont été effectuées à l'aide du modèle logiciel du système. Pour simplifier la simulation logicielle, la fonction de hachage est ignorée et seulement le comportement de la table de hachage est simulé.

On assume que la fonction de hachage distribue de façon uniforme les clés reçues en entrées. Cela revient alors à insérer directement dans la mémoire une clé générée aléatoirement, permettant ainsi de simplifier l'insertion. À l'aide de cette supposition, la probabilité que la clé soit insérée avec succès dans une sous-table dépend du nombre d'espaces disponibles qui y restent.

Les résultats suivants ont été obtenus à l'aide de cette simulation logicielle.

3.4.1.1 Nombre de sous-tables de hachage

Le paramètre principal de l'architecture est le nombre de sous-tables de hachage. Ce paramètre affecte directement la capacité pratique du système en augmentant le nombre d'emplacements pour stocker une clé et ainsi permettre une meilleure résolution de collisions.

La première expérience consiste à mesurer le taux d'échec d'insertion dans la table. Le taux d'échec est corrélé avec le taux de remplissage de la table, c'est-à-dire que plus il y a d'entrées dans la table, plus qu'il y a une grande probabilité d'échec d'insertion. Pour mesurer cela, on simule une insertion pour chaque niveau discret de remplissage de la table. Par exemple, pour un système comportant deux sous-tables de 16 éléments chacune, pour une capacité totale de 32 éléments, on simule une insertion lorsqu'il y a 0, 1, 2, ..., 31 et 32 éléments dans

la table. Pour chaque simulation d'insertion, on note si c'est un succès ou un échec. Cette expérience est répétée plusieurs fois avec de nouvelles insertions aléatoires et une moyenne est calculée à partir de ces essais.

Pour comparer l'effet avec différents nombres de sous-tables, il faut s'assurer de conserver la même capacité totale. Les configurations testées sont des puissances de deux pour le nombre de sous-tables. Dans la Figure 3.6, les quatre configurations testées ont chacune une capacité totale de 1024 éléments. L'axe horizontal est le taux de remplissage de la table au moment de la tentative de l'insertion. L'axe vertical est la probabilité d'avoir un échec d'insertion.

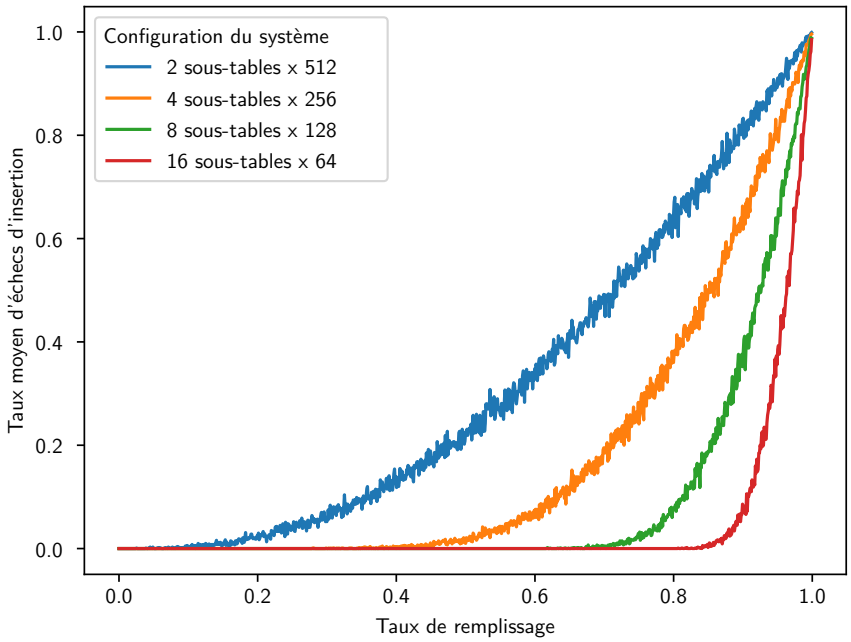


FIGURE 3.6 Probabilité d'échec d'insertion selon le nombre de sous-tables de hachage

On peut observer à la Figure 3.6 que l'ajout de sous-tables permet d'atteindre un plus grand taux de remplissage avant de commencer à avoir des échecs d'insertion, ce qui correspond à la partie de la courbe qui est constante à zéro. On remarque aussi lorsque plus de sous-tables sont ajoutées, leur effet sur taux de remplissage va en diminuant, c'est-à-dire qu'un plus petit gain est obtenu. En principe, pour atteindre un taux de remplissage de 100%, il faudrait 1024 sous-tables d'un élément chacune, ce qui revient à une mémoire complètement associative. Toutefois, augmenter le nombre de sous-tables est une façon d'améliorer l'utilisation mémoire de la table de hachage en offrant plus d'emplacements pour les clés qui génèrent une collision.

3.4.1.2 Capacité de la mémoire CAM secondaire

Le deuxième paramètre à déterminer est la capacité de la mémoire CAM secondaire qui accompagne les sous-tables de hachage. La mémoire CAM sert d'emplacement secondaire pour une clé lorsque celle-ci entre en collision dans toutes les sous-tables.

Pour estimer la taille de la CAM qui permet d'améliorer la capacité pratique de stockage du système, on utilise une simulation logicielle pour mesurer le nombre d'échecs d'insertion afin d'atteindre un taux de remplissage spécifique. L'expérience consiste à insérer des clés générées aléatoirement dans les sous-tables jusqu'à un certain taux de remplissage et de noter le nombre d'échecs d'insertion ayant eu lieu.

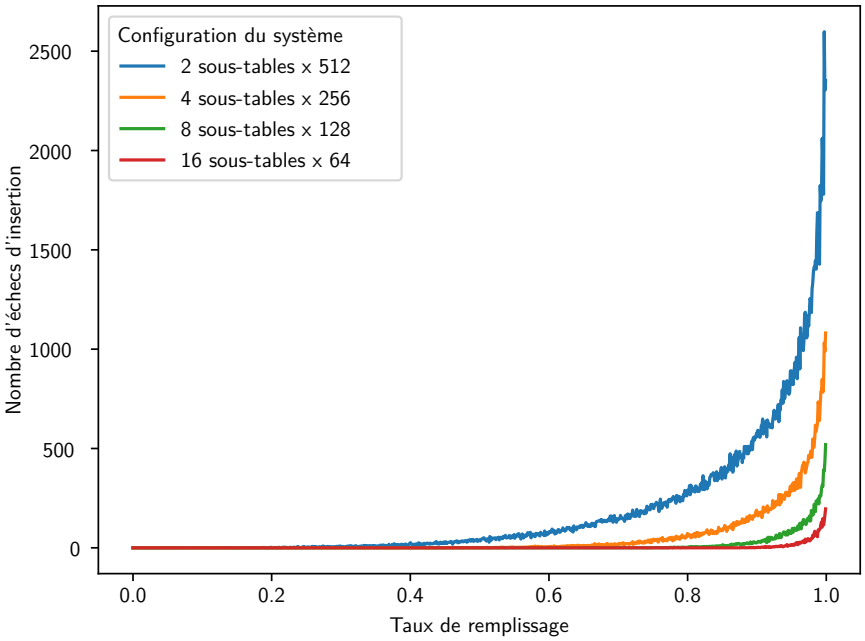


FIGURE 3.7 Estimation de la capacité de la mémoire CAM secondaire afin d'atteindre un taux de remplissage spécifique

La Figure 3.7 montre le nombre d'insertions dans les sous-tables qui ont abouti à un échec selon le taux de remplissage qu'on désire atteindre. On remarque que pour atteindre 100% d'occupation des sous-tables, le nombre d'échecs à accommoder augmente rapidement, surtout pour la configuration avec deux ou quatre sous-tables. Il est plus avantageux de viser un taux de remplissage entre 80% et 95%, ce qui demande une CAM d'une plus petite capacité et qui consomme moins de ressources matérielles.

Nombre de sous-tables	Capacité d'une sous-table (nombre d'éléments)	Capacité totale (nombre d'éléments)	Nombre d'échecs d'insertion
2	512	1024	562
4	256	1024	176
8	128	1024	26
16	64	1024	2

Tableau 3.2 Nombre d'échecs d'insertion pour atteindre un taux de remplissage de 95%

Le tableau 3.2 présente le nombre d'échecs d'insertion pour atteindre un taux de remplissage de 95%. Les mêmes configurations de sous-tables sont utilisées que dans la Figure 3.7. Les valeurs du nombre d'échecs sont utilisées pour définir la taille de la CAM secondaire pour l'implémentation avec CAM dans la prochaine section.

3.4.2 Résultats d'implémentation

Cette section présente les résultats obtenus par l'implémentation du module matériel de recherche exacte.

Le module a été synthétisé sur un FPGA de la famille Virtex-7 de Xilinx. La version exacte de la puce FPGA utilisée est `xc7vx690t-ffg1761-3`. L'outil de synthèse utilisé est la version 2020.2 de Vivado. Les résultats obtenus sont à la suite de la synthèse, du placement et du routage avec les paramètres par défaut de Vivado.

Comme dans la section précédente, une exploration des choix de paramètres est effectuée afin de montrer l'effet des différentes configurations du module sur la consommation de ressources. Pour chaque configuration, les résultats d'utilisation de ressources sont notés à partir des rapports produits par l'outil Vivado. Les éléments du FPGA notés sont les tables de conversion – *Lookup Table* (LUT), les bascules – *Flip Flop* (FF) et les blocs de mémoire – *Block RAM* (BRAM). La puce FPGA utilisée contient 433200 LUT, 866400 FF et 1470 BRAM. De plus, une estimation de la fréquence atteignable est calculée à partir du délai négatif en pire cas – *Worst Negative Slack* (WNS).

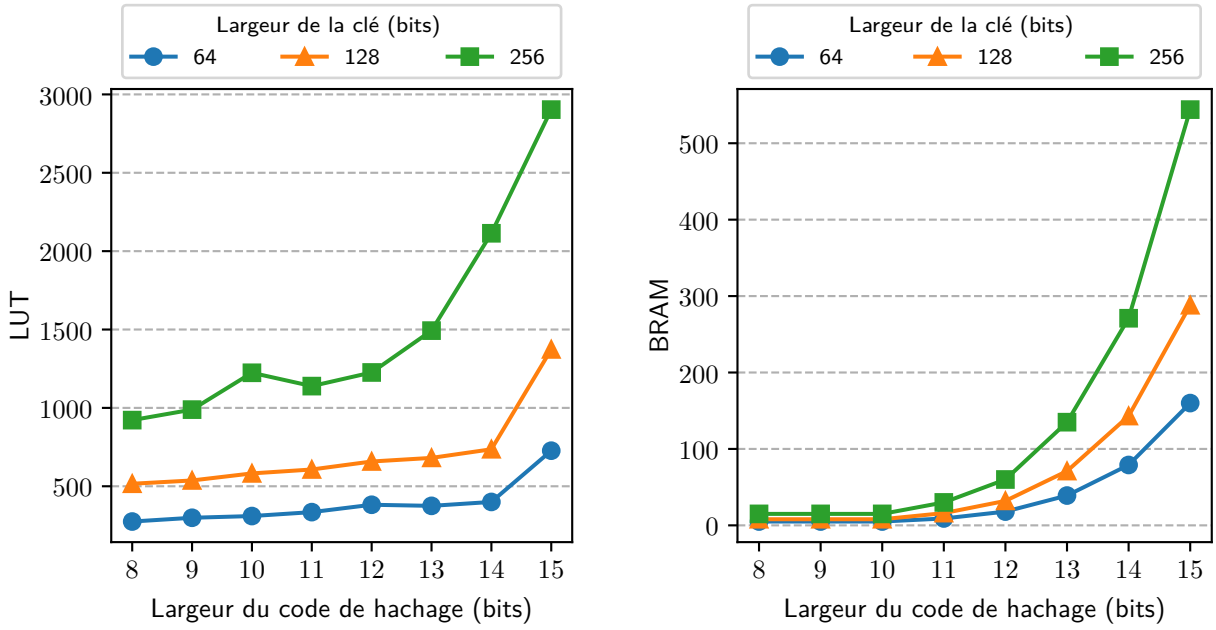
Afin de montrer l'utilisation de ressources dans des cas réels dans le contexte de recherche d'en-têtes dans un paquet, des clés de différentes tailles sont utilisées. Par exemple, une taille de 48 bits est la largeur d'une adresse MAC et est utile pour les commutateurs au niveau de la couche liaison. Un ensemble de champs de différents en-têtes peut aussi être utilisés, tels qu'une combinaison de cinq champs qui peuvent servir à constituer un flot, soit l'adresse IP source, l'adresse IP destination, l'adresse MAC source, l'adresse MAC destination et le

protocole. Cette clé de recherche constituée de plusieurs champs est d'une taille de 168 bits. Ainsi, les largeurs de clés utilisées pour l'implémentation sont de 16, 32, 64, 128 et 256 bits, ce qui couvre les valeurs qui pourraient être utilisées en pratique.

La largeur de la valeur n'est pas étudiée puisqu'une variation de ce paramètre n'affecte pas la complexité du circuit. Si on augmente la taille de la valeur, on peut s'attendre à une augmentation de la mémoire requise. Pour tous les tests effectués dans ce chapitre, la largeur de la valeur est le nombre de bits nécessaire pour adresser une mémoire externe de la même capacité que la table de hachage. Cela suppose le cas où une mémoire externe est utilisée pour stocker la valeur.

3.4.2.1 Largeur du code de hachage

Le premier paramètre qui affecte la capacité du système et ainsi la quantité de ressources nécessaires est la largeur du code de hachage. Le code de hachage est utilisé pour adresser les sous-tables et détermine la capacité d'une sous-table. Ainsi, la capacité d'une sous-table est toujours une puissance de deux. Par exemple, si on décide que le code généré par la fonction de hachage est de 10 bits, alors la capacité de la sous-table sera de 2^{10} éléments, donc 1024 éléments.



(a) LUT

(b) BRAM

FIGURE 3.8 Utilisation de ressources sur FPGA selon la taille des sous-tables (2 tables)

La Figure 3.8 montre l'utilisation de ressources pour une configuration avec deux sous-tables et avec une largeur de clé de 64, 128 et 256 bits. La largeur du code de hachage varie de 8 bits (256 éléments) à 15 bits (32768 éléments). À la Figure 3.8a, l'augmentation du nombre de LUT est principalement due aux fonctions de hachage. L'utilisation en ressources de chaque fonction dépend de la taille de la clé et de la taille du code de hachage. À la Figure 3.8b, le nombre de BRAM augmente de façon exponentielle, ce qui est attendu vu que la quantité d'éléments double à chaque augmentation de la largeur du code de hachage.

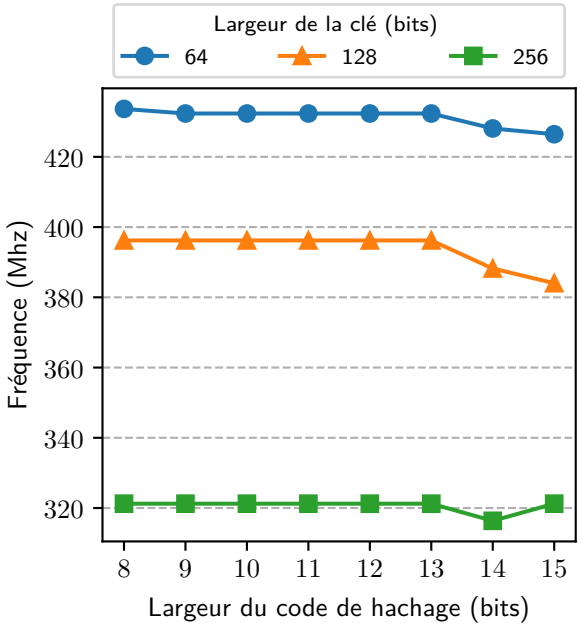


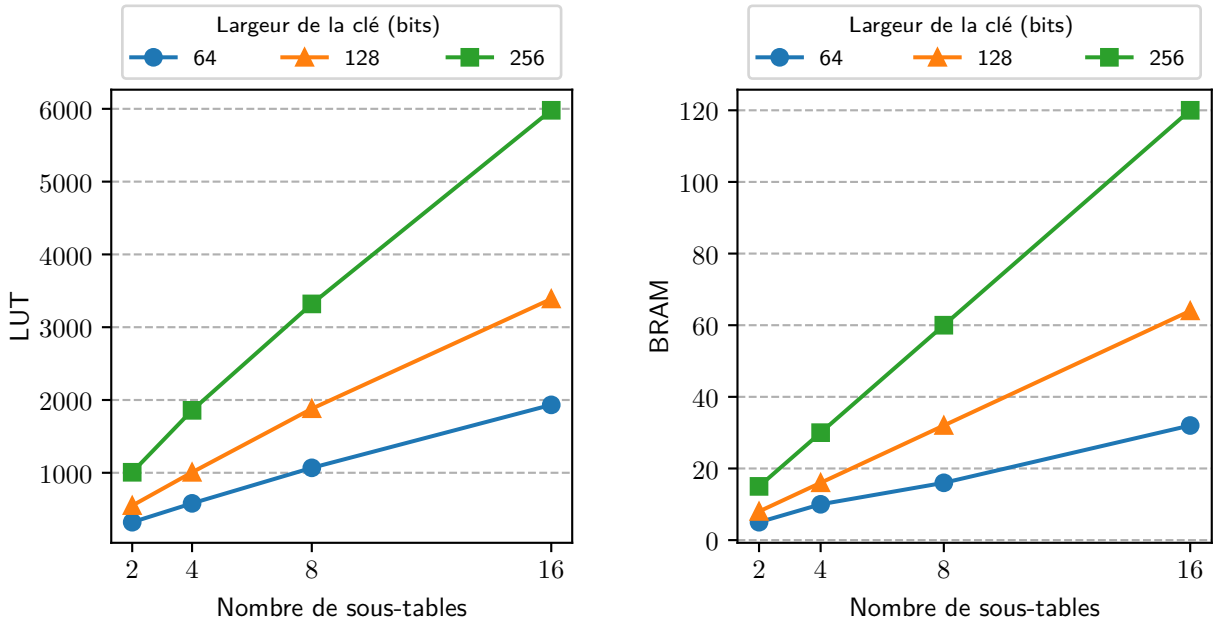
FIGURE 3.9 Fréquence d'horloge selon la taille des sous-tables (2 tables)

La Figure 3.9 montre une estimation de la fréquence atteignable sur FPGA suite au routage. La largeur du code ne hachage n'a peu d'effet sur la fréquence pour des sous-tables de 256 à 8192 éléments. Par contre, on remarque que la largeur de la clé, qui doit être stockée dans les sous-tables, a un effet considérable sur la fréquence. Ceci est dû à l'augmentation en portes logiques pour réaliser les fonctions de hachage et à l'augmentation du nombre de BRAM, vu le nombre de bits qui doit être stocké.

3.4.2.2 Nombre de sous-tables de hachage

Le deuxième paramètre qui influence la capacité de la table de hachage est le nombre de sous-tables dans le système.

Pour mesurer l'impact du nombre de sous-tables sur l'utilisation de ressources, la capacité totale du système est maintenue constante. Différentes configurations qui permettent d'avoir une capacité totale identique ont été testées. Les configurations consistent en 2 sous-tables de 512 éléments, 4 sous-tables de 256 éléments, 8 sous-tables de 128 éléments et 16 sous-tables de 64 éléments. La capacité totale est toujours égale à 1024 éléments.



(a) LUT

(b) BRAM

FIGURE 3.10 Utilisation de ressources sur FPGA selon le nombre de sous-tables de hachage (capacité totale constante de 1024 éléments)

L'effet du nombre de sous-tables est montré dans la Figure 3.10a pour l'utilisation en LUT et dans la Figure 3.10b pour l'utilisation en BRAM. Des tailles de clés de 16, 32, 64, 128 et 256 ont été utilisées. On remarque que le nombre de sous-tables augmente la complexité du circuit et l'utilisation de ressources.

L'augmentation en LUT est principalement due à la fonction de hachage. Chaque sous-table comporte deux instances de la fonction H_3, une pour l'opération d'insertion et une pour l'opération de recherche. Ainsi, à chaque fois que le nombre de sous-tables double, le nombre de LUT requis augmente environ de deux fois. Comme vu précédemment, le nombre de LUT augmente aussi selon la largeur de la clé étant donné que la fonction de hachage dépend de la clé.

L'augmentation en BRAM peut être expliquée par une implémentation non optimale pour FPGA. Tel que décrit dans l'architecture, le nombre de sous-tables est configurable. L'implémentation réalise cela par un paramètre `generic` en VHDL qui est utilisé dans des boucles `for... generate` afin d'instancier le bon nombre de sous-tables de façon automatique, c'est-à-dire sans besoin d'avoir à modifier manuellement le code VHDL. Dû à l'implémentation, cela implique que les mémoires des sous-tables sont indépendantes les unes des autres et sont implémentées sur des blocs BRAM séparés. Ainsi, l'augmentation du nombre de sous-tables nécessite plus de BRAM pour réaliser la mémoire de chaque sous-table. De plus, la taille fixe des blocs BRAM sur FPGA, soit de 18 kilobits ou 36 kilobits, ajoute une autre contrainte sur l'utilisation qu'il est possible d'en faire. Une mémoire qui n'est pas d'une configuration supportée par les BRAM du FPGA implique un certain surcout d'espace. Sans un meilleur partitionnement de la mémoire dans l'architecture, l'outil de synthèse utilise plus de BRAM pour implémenter les sous-tables.

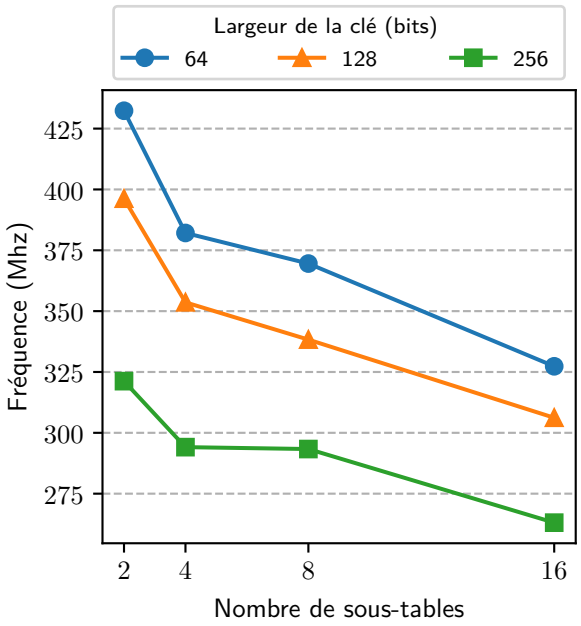
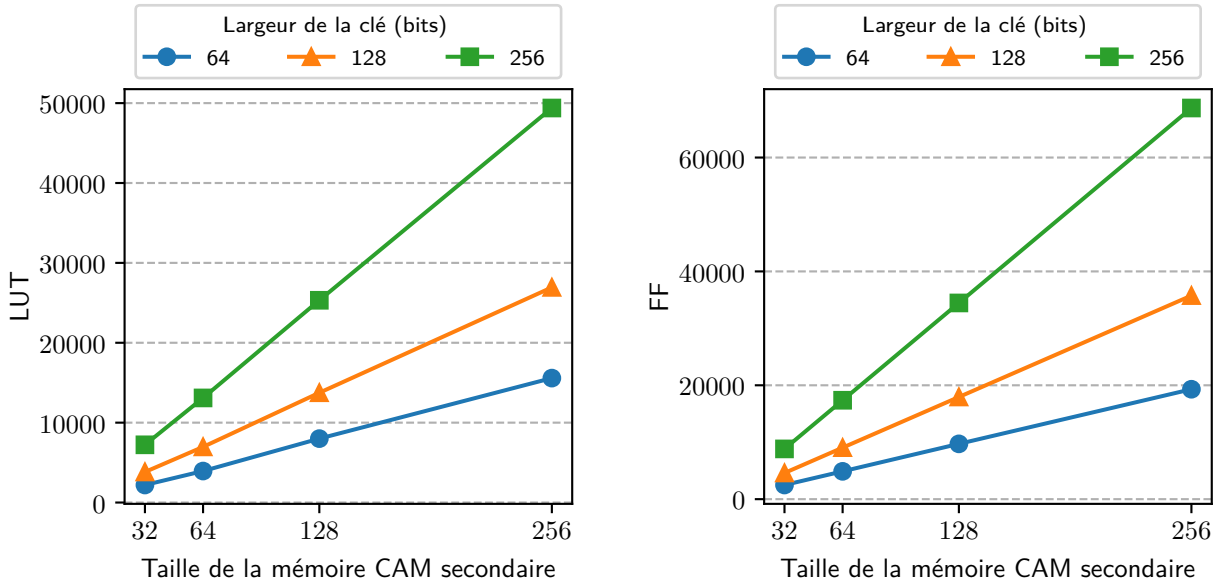


FIGURE 3.11 Fréquence d'horloge selon le nombre de sous-tables de hachage (capacité totale constante de 1024 éléments)

Pour la fréquence d'horloge, les résultats obtenus sont présentés à la Figure 3.11. Étant donné que l'ajout de sous-tables augmente la consommation de BRAM, cela à un impact direct sur la fréquence atteignable. Les BRAM sont des blocs fixes dans le FPGA. Lorsque plusieurs blocs sont connectés ensemble pour former une plus grande mémoire, le chemin critique devient plus long et la latence du circuit augmente.

3.4.2.3 Capacité de la mémoire CAM secondaire

Pour évaluer l'impact de la CAM secondaire sur la consommation de ressources dans le FPGA, des mémoires avec différentes capacités ont été implémentées. Les valeurs testées sont des mémoires comportant 32, 64, 128 et 256 éléments. Le nombre de sous-tables est constant à deux et la taille de chaque sous-table est constante à 1024 éléments.



(a) LUT

(b) FF

FIGURE 3.12 Utilisation de ressources sur FPGA selon la taille de la mémoire CAM secondaire

À la Figure 3.12, on peut voir la consommation de LUT et de FF pour ces différentes valeurs de taille de CAM et pour trois différentes largeurs de clé. Pour le nombre de LUT et le nombre de FF, l'augmentation est linéaire. On remarque que l'utilisation est élevée pour implémenter une mémoire CAM à l'aide de LUT et de bascules. Par exemple, pour une clé de 256 bits de large, la mémoire CAM de 256 éléments utilise 49389 LUT et 68701 FF, ce qui représente respectivement 11.4% et 7.9% des ressources du FPGA.

La fréquence atteignable sur FPGA pour les mêmes tailles de mémoire CAM est présentée à la Figure 3.13. Par rapport à une configuration n'utilisant que des sous-tables de hachage, l'ajout d'une mémoire CAM a pour conséquence de diminuer significativement la fréquence. Pour une CAM de 32 éléments, la fréquence peut diminuer de 62% à 100% selon la taille de la

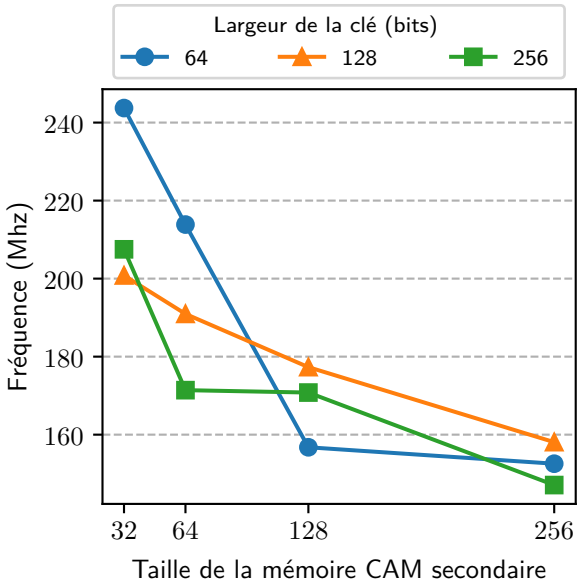


FIGURE 3.13 Fréquence d'horloge selon la taille de la mémoire CAM secondaire

clé par rapport à une implémentation sans CAM. Le besoin en LUT et FF pour implémenter le circuit de la mémoire CAM sur FPGA impose plus de contraintes à l'outil de placement et routage, ce qui cause cet effet.

3.4.3 Comparaison avec la littérature

Les résultats obtenus par l'implémentation de la solution proposée sont comparés avec les résultats de différents travaux de la littérature. Ces travaux sont tous dans le contexte du plan des données programmable et axé sur l'utilisation de tables de comparaison exacte dans un pipeline de traitement de paquets. Les travaux ont tous été synthétisés sur la même puce FPGA de la famille Virtex-7, soit la puce `xc7vx690t`.

La configuration utilisée est avec 8 sous-tables de hachage, ce qui offre environ 70% de taux de remplissage avant que le taux d'échecs d'insertion augmente. La mémoire CAM secondaire n'est pas utilisée étant donné que son implémentation augmente significativement le nombre de LUT et de bascules.

Parmi ces travaux, deux implémentations sont basées sur l'architecture de hachage dMHC [41]. C'est le cas de du projet P4FPGA proposé par Wang et al. [34] et du pipeline programmable proposé par Yazdinejad et al. [40]. Les résultats de synthèse pour ces deux travaux sont présentés dans le tableau 3.3. Par rapport à ces travaux, notre implémentation utilise de

Auteur	Largeur de clé	Capacité	LUT	FF	BRAM
P4FPGA [34]	72	1024	1185	1053	10.5
	144	1024	1395	1440	12.5
	288	1024	2030	2232	16.5
Ce travail	72	1024	1111	14	20
	144	1024	2105	14	36
	288	1024	3719	14	68
Yazdinejad et al. [40]	16	1000	578	352	4
	32	1000	756	562	5.5
	64	1000	905	850	7
	128	1000	1348	1264	9
	256	1000	1625	1924	11
Ce travail	16	1024	319	14	8
	32	1024	588	14	12
	64	1024	1068	14	16
	128	1024	1880	14	32
	256	1024	3320	14	60

Tableau 3.3 Comparaison avec les solutions basées sur l'utilisation d'une table de hachage sur FPGA

22% à 45% moins de LUT pour des clés de petite taille (16 ou 32 bits). Par contre, pour des clés de plus grande taille (128, 144, 256 et 288 bits), notre implémentation utilise de 51% à 104% plus de LUT. L'augmentation en LUT s'explique par la fonction de hachage qui augmente en complexité avec la largeur de la clé. Pour le nombre de bascules, notre implémentation utilise toujours une quantité fixe de FF qui est de 96% à 99% moins élevés que les autres implémentations. Cependant, notre implémentation utilise de 1.9 à 5.5 fois plus de BRAM. L'architecture dMHC est plus avancée et permet une meilleure utilisation des blocs de mémoires BRAM sur FPGA.

Auteur	Largeur de clé	Capacité	LUT	FF	BRAM
Cao et al. [39]	32	256	944	301	64
	48	256	1022	317	80
	48	128	581	188	40
Ce travail	32	256	454	14	12
	48	256	674	14	12
	48	128	562	14	12

Tableau 3.4 Comparaison avec la solution basée sur l'implémentation d'une CAM matérielle sur FPGA

Le troisième travail comparé, celui de Cao et al. [39], utilise une approche basée sur l'implémentation d'une CAM matérielle à l'aide des primitives du FPGA. Les résultats de synthèse de ce travail sont présentés dans le tableau 3.4. Notre implémentation utilise de 3% à 51% moins de LUT, encore une fois lié à l'implémentation des fonctions de hachage. En termes de bascules, il y a une réduction de 93% à 96% avec notre architecture qui nécessite qu'un nombre constant. Pour les blocs BRAM, notre implémentation utilise de 3.3 à 6.7 moins de BRAM qu'une implémentation de CAM matérielle sur FPGA.

3.5 Discussion

L'utilisation de tables de hachage pour imiter une mémoire CAM sur FPGA est une approche intéressante. Les résultats obtenus par notre implémentation montrent qu'il est possible d'utiliser moins de blocs BRAM, ce qui se traduit par la possibilité d'implémenter plus de tables. Même s'il n'est pas possible de remplir entièrement la table avec la méthode utilisant le hachage, on peut prédire un taux de remplissage minimum qui fait en sorte que la consommation de ressources demeure plus avantageuse qu'une CAM matérielle sur FPGA. Toutefois, l'architecture proposée peut être encore améliorée.

D'abord, l'implémentation pour la mémoire CAM secondaire est très coûteuse en LUT et bascules. Ainsi, il est peu avantageux d'utiliser cette mémoire secondaire comme emplacement pour les clés qui causent des collisions dans les sous-tables de hachage. Premièrement, il serait intéressant de comparer l'utilisation de ressources d'une CAM matérielle sur FPGA utilisant des BRAM. Il pourrait être avantageux d'utiliser moins de sous-tables de hachage et de remplacer cela par une mémoire CAM secondaire basée sur des BRAM. Deuxièmement, pour atteindre un taux d'occupation de 95% avec deux sous-tables de 512 éléments, il était nécessaire en moyenne d'ajouter une CAM secondaire de 560 éléments, ce qui est environ la même capacité qu'une sous-table. Une modification de l'architecture intéressante serait d'implémenter le déplacement de clés entre les sous-tables, comme il est fait dans le hachage coucou. Les clés présentes dans la mémoire CAM secondaire seraient insérées dans les sous-tables de hachage, expulsant potentiellement d'autres clés qui seraient insérées dans la CAM. Le processus peut continuer ainsi afin d'augmenter le taux de remplissage des sous-tables de hachage. Une telle méthode nécessiterait une CAM secondaire d'une moins grande capacité, car elle servirait d'emplacement temporaire pour les clés. Par contre, cela peut impliquer une attente de plus d'un cycle d'horloge entre deux insertions.

Ensuite, par rapport aux résultats de l'architecture dMHC, on remarque que notre implémentation peut être encore améliorée. D'abord, étant donné que notre architecture utilise plusieurs sous-tables dont chacune contient une fonction de hachage, le choix de la fonction

est important afin de réduire la consommation en LUT. Il serait intéressant d'évaluer les caractéristiques de différentes fonctions de hachage et leur utilisation en ressource sur FPGA. Ensuite, comme on peut le voir dans les résultats, l'architecture dMHC fait une répartition plus avancée des tables de hachage sur les blocs BRAM. Des concepts de l'architecture dMHC pourraient être intégrés à notre architecture afin d'améliorer l'utilisation de BRAM par les sous-tables de hachage. La combinaison avec le hachage coucou et une mémoire CAM secondaire pourrait être une solution intéressante.

3.6 Conclusion

Ce chapitre présente une architecture pour la recherche exacte sur FPGA. L'implémentation répond aux besoins des plans de données, permettant d'effectuer une recherche à chaque cycle et avec une latence d'un cycle d'horloge. De plus, la consommation de ressources est plus faible que l'implémentation d'une CAM sur FPGA.

Enfin, comme toute architecture matérielle, il existe un compromis entre la quantité de ressources utilisée et la performance du circuit. Pour un plan de données, il est essentiel d'être en mesure d'effectuer une recherche à chaque cycle avec une faible latence, tel que mentionné initialement dans les critères de performance. Cela implique un coût supplémentaire sur la consommation de ressources, dont ce travail a tenté de diminuer à l'aide de tables de hachage. L'architecture proposée permet aussi d'insérer une règle en même temps qu'une opération de recherche, ce qui demande un circuit plus complexe et une plus grande utilisation de ressources. Selon les besoins particuliers d'une application, il serait possible d'accepter une latence plus grande pour l'insertion et d'obtenir une consommation de ressources moins élevée.

CHAPITRE 4 COMPILATION DU FLUX DE CONTRÔLE P4 VERS UNE REPRÉSENTATION MATÉRIELLE

Le langage de programmation dédié P4 offre les abstractions nécessaires pour spécifier comment effectuer le traitement de paquets réseaux. À l'aide de différents compilateurs, il est possible de cibler des plateformes variées, autant logicielles que matérielles. Pour permettre l'utilisation du module de recherche exacte présenté dans le chapitre précédent, un flot de compilation de P4 vers une représentation matérielle est présenté dans ce chapitre.

4.1 Présentation du problème

L'objectif du langage dédié P4 est de permettre la programmation de plans des données par des développeurs ayant des connaissances en réseau informatique. Des concepts haut niveau sont présentés au développeur, tels que la possibilité de définir des tables de comparaison et des actions qui peuvent opérer sur les en-têtes d'un paquet. Dans un programme P4, la cible qui implémente le plan des données est abstraite le plus possible. Pour permettre cela, c'est à la partie dorsale du compilateur que revient la responsabilité de traduire correctement un programme P4 vers la cible d'exécution. Par exemple, pour une cible logicielle, le compilateur P4c permet de générer une représentation intermédiaire textuelle qui est fournie en entrée au modèle comportemental BMv2. Pour une cible matérielle, telle que la puce Tofino, un compilateur propriétaire génère un binaire qui est programmé dans la puce.

Le même principe s'applique lors de l'utilisation de FPGA comme cible d'exécution pour un plan des données programmable. Il ne devrait pas être nécessaire pour un développeur qui programme en P4 de connaître les implémentations spécifiques au FPGA. Ainsi, pour être compatible avec le langage P4, un compilateur doit permettre la compilation de P4 vers une représentation matérielle apte pour l'implémentation sur FPGA.

Dans ce chapitre, il est question de montrer que le module de recherche exacte proposé dans ce travail peut être utilisé dans une implémentation de plans des données définis à partir d'un programme P4. Pour ce faire, nous proposons la partie finale d'un compilateur afin de générer du code VHDL à partir des tables d'un programme P4. Cette solution fait usage du compilateur de référence P4c pour la partie frontale de la compilation [53].

4.2 Description du flot de compilation

La compilation de P4 vers une représentation matérielle se fait en trois étapes principales. D'abord, le code source P4 est analysé par un compilateur de référence. Ensuite, une étape intermédiaire calcule les paramètres de la table. Finalement, la description matérielle est générée.

Une vue d'ensemble du flot de compilation est montrée à la Figure 4.1. Un programme P4 (`code.p4`) est fourni en entrée au compilateur P4c qui produit une représentation intermédiaire, cela correspond à la première étape. Par la suite, cette représentation est utilisée par le compilateur de ce travail afin d'effectuer la passe intermédiaire et la génération de code matérielle, ce qui correspond à la deuxième et troisième étape.

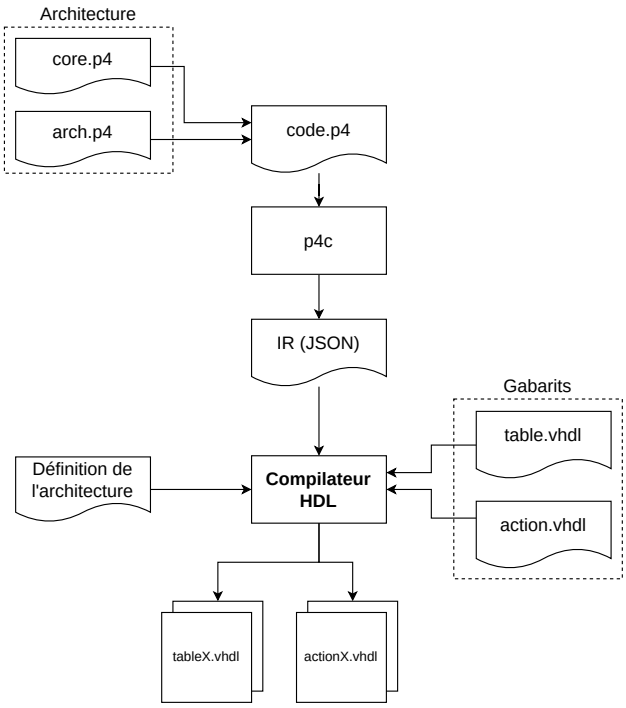


FIGURE 4.1 Flot de compilation de P4 vers une représentation matérielle des tables de comparaison

4.2.1 Parsage par le compilateur de référence

La première étape de compilation consiste en l'analyse du code source P4 afin d'en produire une représentation intermédiaire. Pour accomplir cette tâche, le compilateur de référence P4c est utilisé. P4c est responsable de l'analyse lexicale, de l'analyse syntaxique et de l'analyse sémantique du programme P4. L'utilisation du compilateur de référence permet d'avoir une

source de vérité pour la syntaxe du langage P4 et de faciliter l'ajout d'une cible pour la génération de code. De plus, P4c applique plusieurs passes intermédiaires pour apporter des simplifications au programme P4. La dernière tâche du compilateur P4c est la génération d'une représentation intermédiaire qui sera utilisée pour la suite du travail.

L'exemple de code à l'extrait de code 4.1 montre la façon dont une table de comparaison est déclarée en P4 à l'aide du mot clé `table`. Quatre paramètres sont importants pour l'implémentation de la table en matériel. Ces paramètres sont :

- la clé (*key*) : un ou plusieurs champs des en-têtes du paquet ou des métadonnées associées au paquet
- le type de comparaison : exacte (*exact*), ternaire (*ternary*) ou du plus long préfixe (*lpm*)
- une liste d'actions (*actions*) : les actions qui peuvent être contenues dans une règle et exécutées par la suite lorsqu'il y a correspondance
- la capacité de la table (*size*) : le nombre de règles que la table peut contenir

```
table dmac {
    key = {
        hdr.ethernet.dstAddr : exact;
    }
    actions = {
        forward;
        broadcast;
    }
    size = 1024;
}
```

Extrait de code 4.1 Exemple de déclaration d'une table de comparaison dans un programme P4

Afin d'obtenir une représentation intermédiaire qui soit facilement manipulable, la cible utilisée pour la génération de code est le modèle comportemental BMv2. C'est une plateforme logicielle qui est flexible et qui permet d'exprimer une grande variété de programmes P4. La représentation intermédiaire générée pour BMv2 représente le programme P4 de façon complète et générique, ce qui permet de construire le plan des données défini par le programme P4 dans un autre langage. La sortie du compilateur P4c lorsqu'il cible le modèle BMv2 est un format textuel de type JSON. Pour l'exemple de la définition d'une table en P4 donné à l'extrait de code 4.1, la représentation intermédiaire en JSON est montrée à l'extrait de code 4.2.

```
"tables" : [
    {
        "name" : "dmac",
        "id" : 1,
        "key" : [
            {
            "match_type" : "exact",
            "target" : ["ethernet", "dstAddr"]
            }
        ],
        "match_type" : "exact",
        "type" : "simple",
        "max_size" : 1024,
        "action_ids" : [2, 3, 0],
        "actions" : ["forward", "bcast", "NoAction"]
    }
]
```

Extrait de code 4.2 Exemple simplifié de la représentation intermédiaire d'une table de comparaison en P4

4.2.2 Passe intermédiaire

Une fois que le programme P4 est traduit en format JSON par le compilateur de référence, il faut lire cette représentation intermédiaire pour connaître les paramètres de la table de comparaison.

À partir de la représentation de la table en JSON, montrée à l'extrait de code 4.2, on peut extraire les paramètres de la table essentiels à l'implémentation. D'abord, deux informations peuvent être utilisées directement. Le premier paramètre est le type de comparaison de la table, qui est la recherche exacte, tel qu'il est indiqué par le champ `match_type`. Le deuxième paramètre est le nombre d'éléments que la table doit contenir et est donné par le champ `max_size`.

Ensuite, pour calculer la taille de la clé dans la table, il faut se référer aux en-têtes déclarés dans le programme P4. Dans cet exemple, la table réfère à l'adresse de destination de l'en-tête Ethernet, identifiée par `"ethernet", "dstAddr"]`. La représentation intermédiaire JSON inclut aussi tous les en-têtes du programme P4, comme montré à l'extrait de code 4.3. On voit que la variable dont le nom est `ethernet` correspond au type `ethernet_t`. Ce type est

une structure qui contient plusieurs champs, dont celui qu'on cherche, l'adresse de destination `dstAddr`. On obtient alors que la clé de recherche soit de 48 bits, soit la taille de l'adresse de destination de l'en-tête Ethernet.

```
"header_types" : [
    {
        "name" : "ethernet_t",
        "id" : 2,
        "fields" : [
            ["dstAddr", 48, false],
            ["srcAddr", 48, false],
            ["etherType", 16, false]
        ]
    }
],
"headers" : [
    {
        "name" : "ethernet",
        "id" : 2,
        "header_type" : "ethernet_t",
        "metadata" : false
    }
]
```

Extrait de code 4.3 Exemple simplifié de la représentation intermédiaire des en-têtes en P4

Enfin, le dernier paramètre à calculer est la taille de la valeur dans la table. Pour obtenir ce paramètre, il faut d'abord regarder le nombre d'actions potentielles qui peuvent être exécutées suite à une recherche dans la table. À l'extrait de code 4.2, on voit qu'il y a trois actions possibles, soit `forward`, `bcast` et `NoAction`. La valeur dans la table doit être capable d'adresser ces trois actions, ainsi il faut deux bits afin d'indiquer l'action à effectuer.

De plus, une action peut avoir des paramètres, ce qui requiert des bits de plus. L'action `NoAction` est insérée automatiquement par le compilateur de référence comme action par défaut, c'est-à-dire qu'elle est exécutée lorsqu'une recherche ne correspond avec aucune règle dans la table. Cette action ne fait rien, ainsi ne requiert pas de paramètre. L'action `bcast` signifie qu'il faut *broadcast* un paquet, donc aucun paramètre n'est nécessaire vu que le paquet va être répliqué sur tous les ports. Finalement, l'action `forward` indique sur quel port un paquet doit être transmis. Comme on peut voir dans la représentation intermédiaire de l'action à l'extrait de code 4.4, cette action prend un paramètre nommé `port` de neuf bits.

```
"actions" : [
    {
        "name": "forward",
        "id": 2,
        "runtime_data": [
            {
                "name": "port",
                "bitwidth": 9
            }
        ]
    }
]
```

Extrait de code 4.4 Exemple simplifié de la représentation intermédiaire d'une action en P4

Alors, pour connaître le nombre de bits de la valeur, il faut additionner le nombre de bits pour adresser les actions, soit deux, plus le nombre de bits du paramètre avec la plus grande taille, soit dans de cas de neuf bits. Cela donne une valeur qui occupe 11 bits.

4.2.3 Traduction dans une représentation matérielle

La dernière étape de compilation est la génération du code dans un langage de description matérielle. Le langage VHDL est utilisé pour cette partie étant donné que le module de recherche exacte présenté au chapitre 3 est écrit dans ce langage. Le compilateur prend en entrée un gabarit du module de recherche exacte et configure ses paramètres selon ce qui a été calculé à partir de la représentation intermédiaire du programme P4.

De plus, cette étape de compilation est aussi responsable de générer les constantes pour les fonctions de hachages. Ces constantes sont écrites dans un fichier VHDL qui est importé par l'instance du module de recherche.

4.3 Discussion

Il existe peu de compilateurs pour convertir un programme P4 en une représentation matérielle qui puisse être implémentée sur FPGA. Parmi ces compilateurs, deux sont propriétaires, soit le compilateur P4-SDNet de Xilinx et le compilateur P4-VHDL d'Intel. Les compilateurs propriétaires ont peu de possibilités de modification et d'amélioration. Un compilateur à

code source ouvert a été proposé en 2017, nommé P4FPGA, par contre il n'y a pas eu de continuation suite à ce projet. D'autres compilateurs P4 vers FPGA ont aussi été proposés dans la littérature, mais ils ne sont pas disponibles pour être évalués.

Il serait intéressant pour la communauté scientifique et de P4 de maintenir un compilateur à code source ouvert pour la compilation de P4 vers une représentation matérielle pour FPGA. Un FPGA est par conception programmable et est apte pour l'implémentation d'un plan des données. Les différents blocs de base peuvent être conçus et proposés individuellement. Enfin, pour assembler tout cela il faudrait l'ajout d'une partie finale au compilateur de référence P4c qui cible un FPGA.

CHAPITRE 5 CONCLUSION

Le travail de ce mémoire porte sur la conception d'un composant du plan des données programmable, soit la table de recherche exacte, et de la compilation de ce composant à partir d'un langage dédié. Dans ce denier chapitre, un résumé des éléments principaux du travail est présenté, suivi des limitations de la solution et des améliorations potentielles.

5.1 Synthèse des travaux

Le travail réalisé dans le cadre de ce projet porte sur deux aspects principaux. La première partie est la conception d'une architecture pour la recherche exacte dans le cadre d'un plan des données programmable. La deuxième partie est la compilation à partir d'un langage de programmation dédié pour le traitement de paquets, soit le langage P4, vers le module de table de comparaison conçu dans la première partie.

En premier lieu, la conception d'une table de recherche exacte pour FPGA est intéressante afin de répondre aux contraintes des plans des données. La solution proposée permet la recherche d'une règle en un seul cycle et aussi l'insertion dans le même cycle. Cela permet un taux de recherche et de mises à jour élevé sans causer d'interruptions dans le pipeline de traitement. De plus, dans le but de diminuer la consommation de ressources sur FPGA, une architecture à base de tables de hachage est utilisée afin d'imiter le comportement d'une mémoire associative.

En second lieu, une proposition de processus de compilation à partir du langage P4 vers le module de recherche exacte est présentée. À partir des paramètres d'une table de comparaison définis dans un programme P4, il est possible d'instancier automatiquement le module de recherche exacte.

5.2 Limitations de la solution proposée

L'implémentation de la mémoire associative dans ce travail est réalisée à l'aide de tables de hachage afin d'améliorer la consommation de ressources sur FPGA. Par contre, cette structure de données a pour effet que la table ne sera jamais remplie complètement lors d'une utilisation normale, vu la probabilité que deux clés entrent en collisions augmente avec le taux de remplissage. Ce travail montre qu'avec l'utilisation de plusieurs sous-tables et d'une mémoire CAM secondaire, il est possible d'arriver à une estimation de la capacité de stockage pratique de la table. Cependant, il n'est pas possible de prédire le nombre exact

d'insertions qu'il sera possible d'effectuer avant de recevoir un échec d'insertion d'une clé. L'échec d'insertion d'une clé peut arriver à différents moments selon les constantes qui ont été utilisées pour initialiser les fonctions de hachages et du patron du trafic réseau qui est traité par la table.

Le module proposé dans ce travail est conçu spécifiquement pour la recherche exacte et est basé sur l'emploi de la mémoire BRAM interne au FPGA. Par contre, un plan des données programmé en P4 fait aussi usage de recherche ternaire dans les tables de comparaison. La recherche ternaire n'est pas implémentée par le module proposé et la conception d'une solution performante sur FPGA pour ce type de recherche nécessiterait d'autres structures de données et algorithmes. Un constat similaire peut être fait pour la capacité de la table de recherche. La taille en bits d'une BRAM et le nombre de BRAM dans une puce FPGA sont limités. De plus, l'implémentation d'une table d'une grande capacité à partir des mémoires BRAM impose une contrainte forte pour le placement et routage dans la structure du FPGA. Cela a pour effet de nuire à la fréquence d'horloge que le module matériel peut atteindre. L'utilisation d'une mémoire externe, tel que la mémoire DRAM, fonctionne de façon différente que la mémoire BRAM. Il faut prendre en compte l'interface avec la mémoire, le débit et la latence variable des accès. Encore une fois, une solution qui fait usage de mémoires externes DRAM nécessiterait une approche différente pour être performante.

5.3 Améliorations futures

En utilisant des traces de trafic réseau, il serait possible de mieux caractériser le comportement des fonctions de hachage et des tables de hachage qui composent la solution proposée. Les champs, tels qu'une adresse IP ou une adresse MAC, présents dans les en-têtes des paquets de la trace réseau peuvent être fournis en entrée au module afin de simuler l'insertion de clés avec un patron de réel trafic réseau.

Une limitation de la solution est la possibilité d'un échec d'insertion d'une clé dans la table. Il serait alors intéressant d'utiliser un algorithme de remplacement selon le comportement voulu. Selon l'application, il est peut-être avantageux de laisser tomber l'insertion de la nouvelle clé et d'indiquer une erreur. Un autre scénario est d'expulser une clé dans la table afin de pouvoir insérer la nouvelle clé. Par contre, pour réaliser cela il faut connaître les clés en conflits dans les tables de hachage afin de pouvoir en expulser une qui permettrait l'insertion de la nouvelle clé. C'est un problème qui pourrait être étudié en combinaison avec les algorithmes de remplacement commun, tel que la donnée le moins récemment utilisée – *Least Recently Used* (LRU), selon le premier entré, premier sorti – *First In First Out* (FIFO) ou par une sélection aléatoire.

Un autre aspect à prendre en compte est le besoin pour une grande capacité de stockage de règles tout en gardant la latence d'accès minimale. C'est une application potentielle de l'architecture de recherche exacte proposée dans ce travail. Afin de pouvoir entreposer un plus grand nombre de règles, il faut nécessairement utiliser une mémoire externe. Vu la latence d'accès plus grande, il serait intéressant d'utiliser une structure dans le FPGA comme cache pour accélérer la lecture des règles les plus communes. Le module proposé dans ce travail pourrait être adapté afin de servir de cache interne dans le FPGA. Il faudrait alors étudier comment choisir les règles les plus communes à stocker sur la mémoire interne et de déterminer la taille de la cache qui donne un équilibre entre l'accélération de la recherche d'une règle et la consommation de ressources sur le FPGA.

RÉFÉRENCES

[1] N. Feamster, J. Rexford et E. Zegura, "The Road to SDN : An Intellectual History of Programmable Networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, n°. 2, p. 87–98, avr. 2014. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?doid=2602204.2602219

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica et M. Horowitz, "Forwarding metamorphosis : Fast programmable match-action processing in hardware for SDN," dans *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM - SIGCOMM '13*. Hong Kong, China : ACM Press, 2013, p. 99. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?doid=2486001.2486011

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et D. Walker, "P4 : Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, n°. 3, p. 87–95, juill. 2014. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?doid=2656877.2656890

[4] R. Giladi, *Network Processors : Architecture, Programming, and Implementation*, ser. The Morgan Kaufmann Series in Systems on Silicon. Amsterdam ; Boston : Morgan Kaufmann, 2008.

[5] J. Aweya, "Introduction To Switch/Router Architectures," dans *Switch/Router Architectures : Shared-Bus and Shared-Memory Based Systems*. IEEE, 2018, p. 1–16. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/8360683

[6] R. Miao, H. Zeng, C. Kim, J. Lee et M. Yu, "SilkRoad : Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs," dans *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA : Association for Computing Machinery, août 2017, p. 15–28. [En ligne]. Disponible : https://doi.org/10.1145/3098822.3098824

[7] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports et P. Richtarik, "Scaling Distributed Machine Learning with In-Network Aggregation," dans *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, 2021, p. 785–808. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi21/presentation/sapio

[8] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz et A. Greenberg, "Azure accelerated networking : SmartNICs in the public cloud," dans *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18. USA : USENIX Association, avr. 2018, p. 51–64.

[9] G. Gibb, G. Varghese, M. Horowitz et N. McKeown, "Design principles for packet parsers," dans *Architectures for Networking and Communications Systems*. San Jose, CA, USA : IEEE, oct. 2013, p. 13–24. [En ligne]. Disponible : http://ieeexplore.ieee.org/document/6665172/

[10] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref et R. Soulé, "Fast String Searching on PISA," dans *Proceedings of the 2019 ACM Symposium on SDN Research*. San Jose CA USA : ACM, avr. 2019, p. 21–28. [En ligne]. Disponible : https://dl.acm.org/doi/10.1145/3314148.3314356

[11] "P4-16 Portable Switch Architecture (PSA)," nov. 2018. [En ligne]. Disponible : https://p4.org/p4-spec/docs/PSA-v1.1.0.html

[12] "P4 Portable NIC Architecture (PNA)." [En ligne]. Disponible : https://p4.org/p4-spec/docs/PNA.html

[13] N. Zilberman, P. M. Watts, C. Rotsos et A. W. Moore, "Reconfigurable Network Systems and Software-Defined Networking," *Proceedings of the IEEE*, vol. 103, n°. 7, p. 1102–1124, juill. 2015.

[14] "Intel Tofino Programmable Ethernet Switch ASIC." [En ligne]. Disponible : https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[15] "Intel Tofino 2 Programmable Ethernet Switch ASIC." [En ligne]. Disponible : https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[16] "SDNet Packet Processor User Guide (UG1012)," déc. 2017. [En ligne]. Disponible : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/UG1012-sdnet-packet-processor.pdf

[17] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon et M. Casado, "The Design and Implementation of Open vSwitch," dans *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, p. 117–130. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[18] "Data Plane Development Kit (DPDK)." [En ligne]. Disponible : https://www.dpdk.org/

[19] "Extended Berkeley Packet Filter (eBPF)." [En ligne]. Disponible : https://ebpf.io/

[20] M. Irfan, Z. Ullah et R. C. C. Cheung, "Zi-CAM : A Power and Resource Efficient Binary Content-Addressable Memory on FPGAs," *Electronics*, vol. 8, n°. 5, p. 584, mai 2019. [En ligne]. Disponible : https://www.mdpi.com/2079-9292/8/5/584

[21] Z. Ullah, "LH-CAM : Logic-Based Higher Performance Binary CAM Architecture on FPGA," *IEEE Embedded Systems Letters*, vol. 9, n°. 2, p. 29–32, juin 2017.

[22] R. Pagh et F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, n°. 2, p. 122–144, mai 2004. [En ligne]. Disponible : https://linkinghub.elsevier.com/retrieve/pii/S0196677403001925

[23] D. Fotakis, R. Pagh, P. Sanders et P. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time," *Theory of Computing Systems*, vol. 38, n°. 2, p. 229–248, févr. 2005. [En ligne]. Disponible : http://link.springer.com/10.1007/s00224-004-1195-x

[24] A. Kirsch, M. Mitzenmacher et U. Wieder, "More Robust Hashing : Cuckoo Hashing with a Stash," *SIAM Journal on Computing*, vol. 39, n°. 4, p. 1543–1561, janv. 2010. [En ligne]. Disponible : http://epubs.siam.org/doi/10.1137/080728743

[25] W. Liang, W. Yin, P. Kang et L. Wang, "Memory efficient and high performance key-value store on FPGA using Cuckoo hashing," dans *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, août 2016, p. 1–4.

[26] S. Pontarelli, P. Reviriego et M. Mitzenmacher, "EMOMA : Exact Match in One Memory Access," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, n°. 11, p. 2120–2133, nov. 2018. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/8323198/

[27] Y. Yang, S. R. Kuppannagari, A. Srivastava, R. Kannan et V. K. Prasanna, "FAST-THash : FPGA-Based High Throughput Parallel Hash Table," dans *High Performance Computing*, ser. Lecture Notes in Computer Science, P. Sadayappan, B. L. Chamberlain, G. Juckeland et H. Ltaief, édit. Cham : Springer International Publishing, 2020, p. 3–22.

[28] T. Stimpfling, N. Bélanger, J. M. P. Langlois et Y. Savaria, "SHIP : A Scalable High-Performance IPv6 Lookup Algorithm That Exploits Prefix Characteristics," *IEEE/ACM Transactions on Networking*, vol. 27, n°. 4, p. 1529–1542, août 2019.

[29] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown et S. Licking, "Packet Transactions : High-Level Programming for Line-Rate Switches," dans *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*. Florianopolis, Brazil : ACM Press, 2016, p. 15–28. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?doid=2934872.2934900

[30] A. Sivaraman, N. McKeown, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall et S. Katti, "Programmable Packet Scheduling at Line Rate," dans *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*. Florianopolis, Brazil : ACM Press, 2016, p. 44–57. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?doid=2934872.2934899

[31] "Broadcom's new Trident 4 and Jericho 2 switch devices offer programmability at scale." [En ligne]. Disponible : https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale

[32] "NPL Specification," juin 2019. [En ligne]. Disponible : https://raw.githubusercontent.com/nplang/NPL-Spec/master/NPL%20spec%20version%201.3.pdf

[33] "Behavioral Model v2 (bmv2) : Reference P4 software switch," The P4 Language Consortium. [En ligne]. Disponible : https://github.com/p4lang/behavioral-model

[34] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster et H. Weatherspoon, "P4FPGA : A Rapid Prototyping Framework for P4," dans *Proceedings of the Symposium on SDN Research - SOSR '17*. Santa Clara, CA, USA : ACM Press, 2017, p. 122–135. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?doid=3050220.3050234

[35] P. Coussy et A. Morawiec, édit., *High-Level Synthesis : From Algorithm to Digital Circuit*. New York : Springer, 2008.

[36] "Bluespec Compiler," B-Lang, sept. 2021. [En ligne]. Disponible : https://github.com/B-Lang-org/bsc

[37] "P4-SDNet User Guide (UG1252)," janv. 2018. [En ligne]. Disponible : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1252-p4-sdnet.pdf

[38] S. Ibanez, G. Brebner, N. McKeown et N. Zilberman, "The P4->NetFPGA Workflow for Line-Rate Packet Processing," dans *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside CA USA : ACM, févr. 2019, p. 1–9. [En ligne]. Disponible : https://dl.acm.org/doi/10.1145/3289602.3293924

[39] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen et C. Zhang, "P4 to FPGA-A Fast Approach for Generating Efficient Network Processors," *IEEE Access*, vol. 8, p. 23 440–23 456, 2020.

[40] A. Yazdinejad, R. M. Parizi, A. Bohlooli, A. Dehghantanha et K.-K. R. Choo, “A high-performance framework for a network programmable packet processor using P4 and FPGA,” *Journal of Network and Computer Applications*, vol. 156, p. 102564, avr. 2020. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S1084804520300382

[41] U. Dhawan et A. DeHon, “Area-efficient near-associative memories on FPGAs,” dans *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’13. New York, NY, USA : Association for Computing Machinery, févr. 2013, p. 191–200. [En ligne]. Disponible : https://doi.org/10.1145/2435264.2435298

[42] “Netcope - P4 to VHDL.” [En ligne]. Disponible : https://www.netcope.com/en/products/p4-to-vhdl

[43] Y. Yu, D. Belazzougui, C. Qian et Q. Zhang, “Memory-Efficient and Ultra-Fast Network Lookup and Forwarding Using Othello Hashing,” *IEEE/ACM Transactions on Networking*, vol. 26, nº. 3, p. 1151–1164, juin 2018. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/8335783/

[44] N. Zilberman, Y. Audzevich, G. A. Covington et A. W. Moore, “NetFPGA SUME : Toward 100 Gbps as Research Commodity,” *IEEE Micro*, vol. 34, nº. 5, p. 32–41, sept. 2014.

[45] “Alveo U200 Data Center Accelerator Card.” [En ligne]. Disponible : https://www.xilinx.com/products/boards-and-kits/alveo/u200.html

[46] I. Sarbishei, “A Scalable High-Performance Memory-Less IP Address Lookup Engine Suitable for FPGA Implementation,” Mémoire de maîtrise, Polytechnique Montréal, 2016. [En ligne]. Disponible : https://publications.polymtl.ca/2355/

[47] N. Gebara, A. Lerner, M. Yang, M. Yu, P. Costa et M. Ghobadi, “Challenging the Stateless Quo of Programmable Switches,” dans *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. Virtual Event USA : ACM, nov. 2020, p. 153–159. [En ligne]. Disponible : https://dl.acm.org/doi/10.1145/3422604.3425928

[48] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar et S. Seshan, “TEA : Enabling State-Intensive Network Functions on Programmable Switches,” dans *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’20. New York, NY, USA : Association for Computing Machinery, juill. 2020, p. 90–106. [En ligne]. Disponible : https://doi.org/10.1145/3387514.3405855

[49] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici et G. Siracusano, "FlowBlaze : Stateful Packet Processing in Hardware," dans *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, p. 531–548. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[50] J. L. Carter et M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, nº. 2, p. 143–154, avr. 1979. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/0022000079900448

[51] T. Gingold, "GHDL : VHDL 2008/93/87 simulator." [En ligne]. Disponible : https://github.com/ghdl/ghdl

[52] C. Higgs et S. Hodgson, "Cocotb : Coroutine based cosimulation library for writing VHDL and Verilog testbenches in Python." [En ligne]. Disponible : https://github.com/cocotb/cocotb

[53] "P4c Reference Compiler," P4.org, avr. 2022. [En ligne]. Disponible : https://github.com/p4lang/p4c