Titre: Title:	Optimisation de la compilation de déparseurs pour processeurs réseau implémentés sur FPGA
Auteur: Author:	Thomas Luinaud
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Luinaud, T. (2022). Optimisation de la compilation de déparseurs pour processeurs réseau implémentés sur FPGA [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10357/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Optimisation de la compilation de déparseurs pour processeurs réseau implémentés sur FPGA

THOMAS LUINAUD
Département de génie informatique et génie logiciel

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

Mai 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Optimisation de la compilation de déparseurs pour processeurs réseau implémentés sur FPGA

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

Samuel PIERRE, président
Pierre LANGLOIS, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
Guy BOIS, membre
Mohamed-Faten ZHANI, membre externe

DÉDICACE

À mes grands-parents, à mes parents,
à mon frère, à mes amis
et à mon amoureuse.

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur de recherche, Pierre Langlois, et mon codirecteur, Yvon Savaria, qui m'ont accompagné tout au long de la réalisation de cette thèse. Je voudrais notamment les remercier pour l'encadrement, les conseils reçus au cours de ce doctorat ainsi que le soutien sans lequel ce manuscrit n'aurait pas vu le jour. Aussi un merci particulier à Pierre pour s'être toujours rendu disponible.

Merci à Kaloom, et particulièrement à Laurent Marchand, Ludovic Béliveau et Bochra Boughzala pour la confiance qu'ils nous ont accordés ainsi que l'accès a un environnement stimulant pour nous aider à mener nos recherches. Je voudrais également remercier Normand Bélanger pour ses conseils et les échanges aussi bien personnels que techniques que nous avons eux, mais aussi pour son soutien. Un merci particulier à Martine Bellaïche pour m'avoir écouté et encouragé pendant ces années de doctorat.

Je veux exprimer toute mon amitié à mes collègues de laboratoires qui m'ont soutenu durant la réalisation de ce doctorat. Merci Bachir, Charles, Imad, Erika, Mickaël, Vincent et Patrick. Un énorme merci à Thibaut et Jeferson pour m'avoir encouragé et soutenu au cours de cette expérience. Je suis fier de ce que nous avons accompli ensemble. J'espère que nous continuerons à partager de bons moments ensemble.

Un énorme merci à ma famille qui m'a soutenu tout au long de ce parcours malgré la distance. Un merci particulier à mon frère qui est toujours là quand il y a besoin. Un grand merci à mes parents pour m'avoir toujours soutenu et encouragé dans mes projets. Un grand merci à Sylvie, Stephen et Danielle pour votre soutien.

Un grand merci à tous mes ami.e.s. Merci à Paul de m'avoir aidé à garder le moral. Merci à Sylvain, Élodie, Yann, Adrien, Mélissa et Johanna pour toutes ces activités et les moments d'évasion passés ensemble et pour votre soutien.

Finalement un énorme merci à Laurie. Je te remercie particulièrement pour tout ton soutien au cours de ces années de doctorat . Tu m'as encouragé, motivé tout au long de ce projet, et tu as mis de côté certains de tes projets. Pour tout ça merci !

RÉSUMÉ

Les réseaux de télécommunication ont vu leur taille, leur complexité et leur débit augmenter de façon très importante ces dernières années. Pour suivre cette évolution, les commutateurs, responsables d'aiguiller les paquets, ont perdu en flexibilité ce qui a rendu l'intégration de nouveaux protocoles plus compliquée. Afin de redonner de la flexibilité aux réseaux, le paradigme du réseau défini par logiciel — *Software Defined Networking* (SDN) a été introduit. Avec le SDN, le plan des données et le plan de contrôle sont séparés et peuvent donc être modifiés de manière indépendante. Récemment, afin de rendre le plan des données programmables, l'architecture de commutateur indépendante des protocoles — *Protocol Independent Switch Architecture* (PISA) a été introduite avec le langage P4 permettant de la programmer. Dans cette thèse, nous nous intéressons à l'optimisation de la compilation d'applications décrites dans le langage P4 en vue de leur implémentation sur des réseaux prédiffusés programmables — *Field Programmable Gate Array* (FPGA).

Tout d'abord, nous cherchons à définir les limites des FPGA pour l'implémentation d'applications décrites avec le langage P4. Pour cela, nous proposons une évaluation des différents éléments composant l'architecture PISA, et des pistes de travail afin d'améliorer l'implémentation de plans des données sur FPGA. Les résultats de notre évaluation montrent que la microarchitecture actuelle des FPGA limite l'implémentation de certains types de comparaisons ainsi que le débit maximum qui peut être atteint. Également, nos résultats montrent que l'implémentation du déparseur, responsable de générer le paquet à émettre, peut être améliorée.

À la suite de notre analyse de l'implémentation de programmes P4 sur FPGA, nous avons identifié deux éléments limitant les implémentations actuelles du déparseur. D'une part, le graphe de déparsage généré depuis P4 est trop séquentiel. D'autre part, les architectures de déparseurs ne tirent pas suffisamment avantage de la structure interne des FPGA.

Pour augmenter le parallélisme exprimé par le graphe de déparsage, nous proposons d'effectuer une fermeture transitive du graphe. Cette transformation permet d'exposer le parallélisme pour l'évaluation des conditions de transition entre les sommets du graphe.

Avec la transformation du graphe de déparsage, nous proposons une nouvelle architecture de déparseur générée depuis le graphe. L'architecture proposée maximise l'utilisation de connexions fixes afin de tirer avantage de la capacité de reconfiguration du FPGA. Notre approche permet ainsi de générer des déparseurs depuis des programmes P4 utilisant de 4× à 10× moins de ressources que les travaux précédents.

Afin de réduire le coût d'implémentation de notre nouvelle architecture, nous proposons d'élaguer le graphe de déparsage en effectuant la spécialisation de programmes P4. La spécialisation est effectuée à l'aide de l'évaluation symbolique de certaines variables du programme P4. L'élagage du graphe de déparsage permet de réduire de 50 % l'utilisation de ressources par le déparseur comparé à la première approche proposée.

ABSTRACT

Computer networks have, in recent years, significantly increased in size, complexity and throughput. To keep up with this evolution, switches, which route network packets, have lost in flexibility, hence making the integration of new protocols more complex. To restore flexibility into the network, the Software Defined Networking (SDN) paradigm has been introduced. With SDN, the data and control planes are separated and can therefore evolve independently. Recently, to bring programmability to the data plane, the Protocol Independent Switch Architecture (PISA) has been proposed alongside the P4 language to program it. In this thesis, we focus on the optimization of the compilation of applications described using the P4 language with regards to their implementation on Field Programmable Gate Arrays (FPGAs).

First of all, we seek to determine the limits of FPGAs concerning the implementation of applications described with the P4 language. For this purpose, we propose an evaluation of the different elements composing the PISA architecture, and avenues to improve implementation of network data planes on FPGA. Our evaluation results demonstrate that the current microarchitecture of FPGAs limits the implementation of certain types of comparisons as well as the maximum achievable throughput. Also, our results indicate that the implementation of the deparser, which generates the packet to be emitted, can be improved.

Resulting from our analysis of the implementation of P4 programs on FPGAs, we have identified two elements limiting current deparser implementations. Firstly, the deparser graph generated from the P4 code is too sequential. Secondly, deparser architectures do not take sufficient advantage of the internal structure of FPGAs.

To increase the parallelism expressed in the deparser graph, we propose to perform a transitive closure on the graph. With this transformation it is possible to expose the parallelism of the transition conditions evaluation between the vertices of the graph.

Along with the transformation of the deparser graph, we propose a new deparser architecture generated from the graph. The proposed architecture maximizes the use of fixed connections in order to take advantage of the reconfiguration capability of the FPGA. Our approach makes it possible to generate deparsers from P4 programs using 4× to 10× fewer resources compared to previous work.

In order to reduce the cost of our new architecture, we propose to prune the deparsing graph by performing P4 program specialization. The specialization is carried out by applying

symbolic evaluation on certain variables of the P4 program. Pruning the deparsing graph reduces the deparser resource usage by 50 % compared to the first approach we proposed.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ASIC	Circuit intégré spécifique à l'application — *Application Specific Integrated Circuit*
BMv2	Modèle comportemental version 2 — *Behavioral Model version 2*
BRAM	Bloc RAM
CAM	Mémoire adressable par contenu — *Content Adressable Memory*
CLB	Bloc de logique configurable — *Configurable Logic Bloc*
eBPF	BPF étendue — *extended BPF*
FPGA	Réseau prédiffusé programmable — *Field Programmable Gate Array*
GDA	Graphe Dirigé Acyclique
LPM	Comparaison du plus long prefixe — *Longest Prefix Match*
LUT	Table de conversion — *LookUp Table*
NIC	Carte d'interface réseau — *Network Interface Card*
NoC	Réseau sur puce — *Network on Chip*
ODG	Graphe de dépendance des opérations — *Operation Dependency Graph*
OLNE	Optimisation Linéaire en Nombres Entiers
PHV	Vecteur d'en-têtes de paquet — *Packet Header Vector*
PISA	Architecture de commutateur indépendante des protocoles — *Protocol Independent Switch Architecture*
PSA	Architecture portable de commutateurs — *Portable Switch Architecture*
RISC	Processeurs à jeu d'instructions réduit — *Reduced Instruction Set Computer*
SDN	Réseau défini par logiciel — *Software Defined Networking*
SRAM	Mémoire à accès aléatoire statiques — *Static Random Access Memory*
TCAM	Mémoire adressable par contenu ternaire — *Ternary Content Adressable Memory*
TDG	Graphe de dépendance de tables — *Table Dependency Graph*
TNA	Architecture native du Tofino — *Tofino Native Architecture*
VVE	Vecteur de Validité d'En-tête

CHAPITRE 1 INTRODUCTION

1.1 Contexte

Les réseaux informatiques fournissent un moyen d'échanger des paquets de données entre des utilisateurs et ces paquets sont aiguillés à travers les réseaux de données à l'aide de commutateurs. La taille et la complexité des réseaux a significativement augmenté au cours des dernières années, en passant d'environ 1 million d'équipements connectés dans le monde en 1992 [1] à plus de 15 milliards en 2018 [2]. Cette augmentation s'accompagne également d'une augmentation très importante des débits : en 1995 un port réseau avait un débit de 100 Mb/s [3], aujourd'hui la norme est de 100 Gb/s. En plus de devoir gérer des réseaux plus grands et plus rapides, les commutateurs doivent, aiguiller les paquets, fournir des fonctionnalités pour la sécurité, assurer la qualité de service et surveiller le trafic. Afin de pouvoir effectuer toutes ces opérations et atteindre les hauts débits requis, les commutateurs sont devenus de moins en moins programmables et les réseaux de communication se sont ossifiées [4]. Cette ossification a amené à une incapacité à déployer de nouveaux algorithmes et de nouveau protocoles pour gérer les réseaux.

Afin de redonner de la flexibilité aux réseaux, le paradigme du réseau défini par logiciel — *Software Defined Networking* (SDN) a été introduit afin de proposer de nouvelles abstractions [5]. Dans le paradigme SDN, le plan des données, responsable du traitement des paquets, et le plan de contrôle, logiquement centralisé et commandant le plan des données, sont séparés afin de pouvoir les modifier de manière indépendante. Plus précisément, le paradigme SDN permet d'abstraire la gestion d'un réseau en fournissant trois séparations claires pour le développement des applications réseau :

- Un contrôleur centralisé (le plan de contrôle) exécute des applications réseau en configurant le plan des données par l'insertion de règles et en récupérant le statut du réseau.
- Une application réseau qui est exécutée sur le plan de contrôle, et est développée en définissant un ensemble de règles de gestion en fonction du statut du réseau.
- Un plan des données transfère les paquets en fonction des règles définies par le plan de contrôle.

Afin d'avoir une interface standard entre le plan de contrôle et le plan des données, McKeown et al. ont proposé OpenFlow en 2008 [6]. En plus de définir une interface standard, les différentes versions d'OpenFlow définissent des ensembles d'actions et de protocoles qu'un commutateur doit prendre en charge. Par conséquent, pour ajouter de nouveaux protocoles

ou de nouvelles actions sur un commutateur, il faut proposer une nouvelle version de la norme. Cependant, lors de la création d'une nouvelle version de la norme, l'intégration de ces changements peut s'avérer coûteuse et longue. Finalement, si un utilisateur n'a besoin que de certaines fonctionnalités d'une version d'OpenFlow, il est obligé d'utiliser un commutateur qui implémente toutes les fonctionnalités de cette version. Ceci peut obliger les utilisateurs à utiliser des commutateurs qui contiennent plus de ressources que nécessaire.

Pour s'émanciper des limites d'OpenFlow, l'architecture de commutateur indépendante des protocoles — *Protocol Independent Switch Architecture* (PISA), une abstraction pour le traitement des paquets, a été proposée par Bosshart et al. en 2013 [7]. PISA est une architecture de type flux de données — *dataflow* qui définit trois éléments configurables, élémentaires et nécessaires au traitement des paquets sur un commutateur réseau :

1. Un parseur responsable d'extraire les en-têtes du paquet à traiter.
2. Des tables de comparaisons-actions qui comparent les en-têtes avec un ensemble de règles, insérées par le plan de contrôle, et qui exécutent une action en fonction du résultat de la comparaison.
3. Le déparseur qui génère un paquet à transmettre en assemblant les en-têtes modifiés et la charge utile du paquet d'entrée.

PISA se présente comme un pipeline au travers duquel passe un paquet. Le pipeline se compose d'un parseur, puis d'une succession d'étages de comparaison-action, et il se termine par le déparseur. Les étages de comparaison-action sont configurables aussi bien sur la taille des tables que des actions qui peuvent être effectuées, un étage pouvant effectuer plusieurs comparaisons et plusieurs actions en parallèle sur différentes portions des en-têtes. Avec le modèle PISA, il est possible de concevoir des commutateurs programmables qui contiennent uniquement les opérations nécessaires au traitement du paquet. Cela facilite la reconfiguration en fournissant une abstraction claire.

Afin de pouvoir programmer le modèle PISA, Bosshart et al. ont proposé, en 2014, le langage dédié P4 [8]. Ce langage permet de définir l'ensemble des en-têtes que le parseur doit extraire, la taille des tables de comparaison-action, les parties des en-têtes à comparer, les actions qui peuvent être exécutées et comment le paquet en sortie doit être constitué. Ainsi, avec le langage P4, tous les éléments de PISA peuvent être configurés. Le langage P4 permet aussi de générer des interfaces pour le plan de contrôle afin de pouvoir configurer le plan des données.

En 2016, le premier circuit intégré spécifique à l'application — *Application Specific Integrated Circuit* (ASIC) basé sur PISA et programmable en P4, le Tofino™, a été commercialisé. Cette première version du Tofino™ atteint des performances équivalentes à celle d'un com-

mutateur non programmable et peut traiter jusqu'à 6,5 Tb/s de trafic [9]. Toutefois, pour atteindre ces hauts débits, certaines opérations demandant beaucoup d'éléments de calculs, tels que le chiffrement, peuvent nécessiter de faire passer un paquet plusieurs fois dans le commutateur [10], ce qui réduit le débit réel de traitement.

En outre, les centres de données ont besoin d'équipements permettant de traiter des données à haut débit tout en étant programmables afin de s'adapter aux nouvelles applications. À cet effet, les centres de données ont récemment intégré des réseaux prédiffusés programmables — *Field Programmable Gate Array* (FPGA) afin d'alléger la charge de calcul sur les serveurs [11–13]. L'utilisation des FPGA requiert toutefois de bonnes connaissances en développement matériel malgré l'évolution récente des outils de synthèse de haut niveau [14].

Les langages dédiés ont montré par le passé qu'ils permettaient d'élever le niveau d'abstraction pour l'implémentation d'applications sur FPGA [15]. Des travaux récents ont montré qu'il était possible d'implémenter des programmes P4 sur FPGA [16, 17], ce qui permet de s'affranchir des limites liées aux ASIC. Cependant, la performance atteinte par les FPGA reste bien inférieure à celle de puces plus spécialisées comme le Tofino. Ceci est dû d'une part à la microarchitecture des FPGA, mais également à un manque d'optimisations, dans le compilateur P4, adaptées aux spécification de l'architecture des FPGA.

Si le SDN a permis d'ouvrir la voie aux réseaux programmables, P4 et PISA sont les clés de voûte qui permettent de rendre le réseau entièrement programmable. Ainsi PISA est aux commutateurs réseau ce que l'architecture de processeurs à jeu d'instructions réduit — *Reduced Instruction Set Computer* (RISC) est aux processeurs génériques, en fournissant une abstraction claire sur l'interface qui doit être fournie au développeur pour programmer des plans des données. Également, le langage P4 peut être vu comme le C des architectures RISC, un langage qui permet de s'émanciper de l'architecture matérielle. P4 et PISA sont donc des avancés majeures pour rendre les réseaux programmables. Dans ce travail, nous proposons d'évaluer la place que peuvent prendre les FPGA dans ce contexte, mais aussi de proposer des modifications aux compilateurs P4 afin d'améliorer l'implémentation de programmes P4 sur FPGA.

1.2 Éléments de la problématique

Les gestionnaires de centres de données ont besoin de matériels configurables afin de pouvoir s'adapter aux différentes applications qu'ils doivent exécuter. En effet, les centres de données sont des environnements dynamiques qui exécutent une variété d'applications [18]. Par conséquent, les gestionnaires ont besoin de pouvoir accélérer divers types d'applications ainsi

que de pouvoir prédire quels types d'accélération doit être disponible à un moment donné. Par exemple, le service Amazon AWS facture ses services à l'heure. Ainsi, des FPGA ont été déployés dans les centres de données afin d'avoir des accélérateurs de calculs flexibles [11, 19]. Cependant, comme les gestionnaires de réseaux veulent pouvoir programmer facilement les FPGA, il leur faut donc des langages et des compilateurs leur permettant de programmer ces plateformes.

En plus des considérations citées, il faut que les réseaux soient flexibles, il faut donc des cartes d'interface réseau — *Network Interface Cards* (NIC) programmables. Cependant les NIC standard ne permettent généralement pas d'atteindre des débits suffisamment élevés lorqu'on les utilise suivant un modèle programmable [20]. Dans le cas des NIC basés sur des ASIC, l'ajout de nouvelles fonctionnalités prend trop de temps au vu du changement des besoins internes des centres de données. Dans le cas des NIC basées sur des architectures de processeurs multicœur, la consommation d'énergie est déjà élevée pour des débits de 40 Gb/s, alors que les nouvelles connexions demandent des débits excédant 100 Gb/s [21]. Les FPGA semble être une plateforme de choix, puisqu'ils peuvent implémenter diverses applications réseau tout en soutenant des débits supérieurs à 400 Gb/s [22].

Également, grâce au langage P4 et à PISA, qui ont rendu le plan des données des réseaux programmables, l'idée du calcul dans le réseau a émergé. Il a même été démontré que cette approche pouvait alléger la charge de traitement sur les serveurs et même réduire la consommation d'énergie [23]. Toutefois, les commutateurs programmables actuels tels que le Tofino™ ne permettent pas de tirer pleinement avantage de cette possibilité. Les FPGA, qui permettent d'implémenter un large panel d'applications, semblent être une solution pour le calcul dans le réseau [24].

Bien que des compilateurs P4 vers FPGA existent [16, 17, 25], il est nécessaire de déterminer si les solutions de compilation actuelles implémentent au mieux les programmes P4 sur FPGA. En effet, les FPGA ont une quantité limitée de ressources qui peuvent être utiles à beaucoup d'applications. Il faut donc déterminer pour chacun des modules de PISA lesquel sont les plus adaptés à une implémentation sur FPGA, il sera ainsi possible de guider les développeurs pour une utilisation plus efficace du FPGA. Également, il est nécessaire d'évaluer sur les compilateurs actuels si des améliorations sont possibles, que ce soit par des transformations du programme lors de la compilation ou bien par l'ajout de nouvelles solutions pour implémenter des parties d'un programme P4 sur le FPGA .

Les FPGA présentent donc un intérêt pour implémenter des applications réseau et ainsi remplacer les NIC. Le fait qu'ils soient programmables en P4 permet aux gestionnaires de réseau d'implémenter plus facilement de nouveaux plans des données. Toutefois, il est néces-

saire de déterminer les modules des applications P4 qui ne s'implémentent pas efficacement sur FPGA, et de savoir si l'inneficacité d'implémentation de ces modules est due à l'architecture des FPGA ou à l'abscence de solutions adaptées pour les FPGA. Plus précisément, dans un pipeline PISA, il est essentiel de pouvoir extraire les en-têtes du paquet, avec le parseur, et d'assembler les en-têtes avec la charge utile du paquet d'entrée pour générer un nouveau paquet en sortie, avec le déparseur. Bien que de nombreux travaux se soient intéressés à la compilation de parseurs de paquets et à leur implémentation sur FPGA [26–30], peu de travaux ont exploré la compilation et l'implémentation du déparseur. Dans ce travail, nous proposons de combler ce manque en évaluant l'implémentation de programme P4 sur FPGA et d'améliorer l'implémentation du déparseur sur FPGA en proposant une nouvelle architecture de déparseur ainsi que des modifications au compilateur P4.

1.3 Objectifs de recherche

Dans cette thèse, nous cherchons à déterminer les améliorations pouvant être apportées aux FPGA afin d'accroître leur capacité pour implémenter des plans des données réseau. Pour cela, nous explorons d'une part les contraintes architecturales liées aux FPGA actuels ainsi que les limites des compilateurs générant des plans des données réseau sur FPGA. Par conséquent, les trois objectifs suivants sont considérés :

1. Déterminer les forces et faiblesse des FPGA actuels pour l'implémentation d'application réseau. Proposer des modifications à la microarchitecture des FPGA et des modifications aux architectures implémentées sur FPGA.
2. Proposer un compilateur P4 permettant de générer une architecture de déparseur adaptée à la structure des FPGA.
3. Proposer une passe de compilation permettant de réduire le coût d'implémentation des applications réseau sur FPGA.

1.4 Contributions de cette thèse

Les contributions des travaux présentés dans cette thèse peuvent se résumer comme suit :

1. Nous proposons d'évaluer l'implémentation des différents blocs de PISA sur FPGA dans le but de proposer des axes d'amélioration pour mieux prendre en charge des programmes P4 sur FPGA. Suite à cette évaluation, nous proposons certaines transformations qui pourraient aider à mieux adapter les FPGA au traitement d'applications réseau. Également, nous constatons que les solutions actuelles pour l'implémentation

de déparseurs sur FPGA consomment une quantité importante de ressources. Ces travaux sont détaillés dans le troisième chapitre de cette thèse.

2. Nous proposons une transformation au graphe de déparsage d'un programme P4 permettant d'exposer le parallélisme du processus de déparsage et de prendre avantage de l'architecture très parallèle des FPGA. Cette transformation est présentée dans le quatrième chapitre de cette thèse.
3. Nous proposons une nouvelle architecture de déparseur pour FPGA et un compilateur permettant de générer cette architecture depuis le graphe de déparsage proposé à la contribution 2. Cette architecture permet de réduire de 4× à 10× l'utilisation de ressources par rapport aux solutions précédentes tout en soutenant des débits de plus de 400 Gb/s. Ces travaux sont présentés dans le quatrième chapitre de cette thèse.
4. Nous proposons de spécialiser un programme P4 afin de diminuer le coût d'implémentation du déparseur présenté à la contribution 3. La spécialisation s'effectue par l'analyse symbolique du programme P4 qui permet de réduire le nombre de chemins dans le graphe de déparsage proposé à la contribution 2. Ces travaux sont présentés dans le cinquième chapitre de cette thèse.

Les différents travaux de cette thèse ont mené à la publication de trois articles :

1. Thomas Luinaud, Thibaut Stimpfling, Jeferson Santiago da Silva, Yvon Savaria et J.M. Pierre Langlois, "Bridging the Gap : FPGAs as Programmable Switches", 2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR).
2. Thomas Luinaud, Jeferson Santiago da Silva, J.M. Pierre Langlois et Yvon Savaria, "Design Principles for Packet Deparsers on FPGAs", 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '21). Association for Computing Machinery, New York, NY, USA, 280–286.
3. Thomas Luinaud, Jeferson Santiago da Silva, J.M. Pierre Langlois et Yvon Savaria, "Network data plane program specialization using symbolic analysis", révisé avec demande de modifications majeures au journal ACM Transactions on Architecture and Code Optimization (TACO).

1.5 Plan de la thèse

Cette thèse est organisée en six chapitres. Tout d'abord, un survol des réseaux programmables et une revue de la littérature sont présentés dans le chapitre 2. Les chapitres suivants de cette thèse présentent les contributions. Tout d'abord, dans le chapitre 3, l'analyse de l'implémentation de programmes P4 sur FPGA est présentée. Cette analyse montre que

l'implémentation du déparseur sur FPGA peut être améliorée. Dans le chapitre 4, une architecture de déparseur pour FPGA est présentée ainsi qu'une transformation du graphe de déparsage depuis le programme P4 permettant de générer l'architecture. Dans le chapitre 5, la spécialisation du déparseur exprimé dans des programmes P4 en effectuant leur analyse symbolique est présentée. Cette spécialisation permet de réduire le coût d'implémentation du déparseur sur FPGA en réduisant le nombre de chemins dans le graphe de déparsage. Finalement, le chapitre 6 présente la conclusion de cette thèse et propose des axes de travaux futurs.

CHAPITRE 2 PRÉLIMINAIRES ET REVUE DE LA LITTÉRATURE

Dans ce chapitre, nous donnons les préliminaires et l'état de l'art permettant de comprendre les réseaux programmables et l'état de la recherche dans ce domaine. Tout d'abord, nous présentons les plans des données programmables. Par la suite, nous présentons différents langages réseau pour les plans des données et notamment le langage P4. Cela nous amène à discuter de différentes architectures P4. Ensuite, nous présentons la compilation du langage P4 à l'aide des différents compilateurs existants. Finalement, nous concluons cette section en posant certaines questions concernant l'implémentation, sur FPGA, de processeurs réseau modélisés avec le langage P4.

2.1 Le réseau programmable

Dans cette section nous présentons différentes recherches qui ont permis d'arriver au plan des données programmables. Dans un premier temps, nous présentons les évolutions qui ont permis de rendre le plan de contrôle programmable. Puis nous discutons des processeurs réseau. Finalement, nous présentons l'architecture de commutateur indépendante des protocoles — *Protocol Independent Switch Architecture* (PISA).

2.1.1 La programmation du plan de contrôle

La programmation des réseaux est étudiée depuis de nombreuses années par les chercheurs [31]. En effet, l'incapacité à programmer les réseaux a pendant longtemps limité le test et le déploiement de nouveaux algorithmes [6], ce qui a limité l'innovation dans les réseaux.

Un tournant majeur a été atteint pour la programmation des réseaux lors de l'introduction du SDN [32]. Le SDN découle de deux éléments :

- L'introduction de l'abstraction du plan de contrôle et du plan des données
- La création d'une architecture réseau ou le plan des données est séparé du plan de contrôle.

Le plan des données effectue le traitement sur les paquets. Le plan de contrôle est une entité logiquement centralisée qui configure le plan des données, à travers des règles, afin de déterminer les opérations à effectuer sur les paquets.

Pour configurer le plan des données depuis le plan de contrôle, McKeown et al. propose une interface standard, OpenFlow [6]. Comme cette interface est standard, il est possible de concevoir des systèmes d'exploitation pour le réseau ainsi que des langages de haut niveau

pour décrire des applications réseau [33–35]. On obtient ainsi une structure, telle que présentée dans la figure 2.1, où des applications réseau utilisent une interface fournie par le plan de contrôle qui configure le plan des données à l'aide d'une autre interface tel qu'OpenFlow.

Bien qu'OpenFlow permette de configurer le plan des données, les commutateurs compatibles avec OpenFlow ne sont pas totalement programmables. En effet, la spécification d'OpenFlow [36] définit un ensemble de protocoles qu'un commutateur doit pouvoir traiter afin d'y être compatible. Il est donc nécessaire de rendre également le plan des données programmable afin d'avoir un réseau totalement programmable.

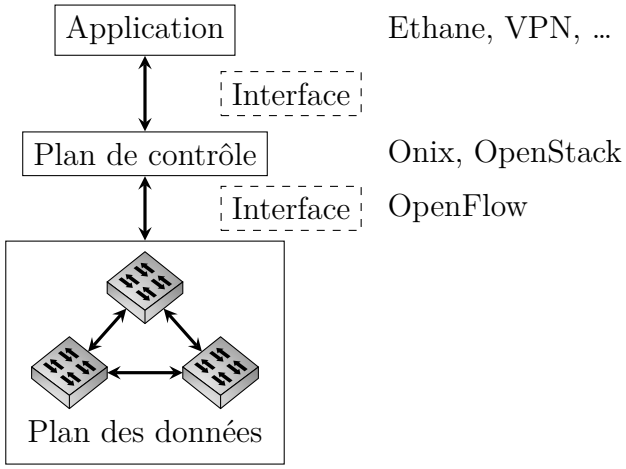


Figure 2.1 Les différentes abstractions d'un réseau

2.1.2 Les processeurs réseau

Les processeurs réseau sont apparus vers la fin des années 1990 [3]. Il s'agit d'équipements spécialisés dans le traitement des paquets réseau, mais pouvant être programmés afin d'exécuter différentes applications. Les processeurs réseau ont des architectures parallèles contenant plusieurs cœurs de calculs avec des coprocesseurs afin d'accélérer le traitement de certaines fonctionnalités telles que les opérations de recherche. Un ordonnanceur assigne à chaque cœur de calcul un paquet à traiter [37].

Les processeurs réseau sont principalement implémentés en utilisant l'architecture de Von Neumann [38]. Cette approche permet aux processeurs réseau d'exécuter une variété importante d'algorithmes allant du routage de paquets jusqu'à l'inspection en profondeur de paquets. En général, les paquets sont assignés à un cœur de calcul qui effectue tous les traitements. De plus, la latence de traitement ne peut être garantie, notamment parce que les différents cœurs de calcul doivent accéder à de la mémoire partagée. Comme la latence de

traitement varie et qu'elle ne peut être garantie, il est courant de devoir réordonnancer les paquets dans un processeur réseau pour garantir leur ordre de sortie.

Un des éléments limitants des processeurs réseau est la complexité à les programmer. Kulkarni et al. ont identifié trois éléments limitant dans la programmation des processeurs réseau [39]. Le premier élément est la répartition des fonctionnalités à exécuter sur les différents cœurs de calcul. Le second élément est l'organisation et l'assignation des différentes ressources disponibles. Le troisième élément est le transfert de données de la mémoire vers les cœurs de calcul ainsi que la répartition des données dans la mémoire. Firestone et al. ont également montré dans le cas des NIC que, pour supporter des débits de plus de 100 Gb/s, l'utilisation d'architectures multicœurs n'était pas possible notamment à cause de la synchronisation entre les différents cœurs [14].

2.1.3 Une architecture de commutateur indépendante des protocoles

Afin de résoudre les problèmes liés aux architectures des processeurs réseaux, Bosshart et al. ont proposé en 2013 une architecture de commutateur indépendante des protocoles — *Protocol Independent Switch Architecture* (PISA) [7]. PISA est une architecture qui propage les données vers l'avant et elle se structure autour de trois abstractions principales : le parseur de paquets, les tables de comparaison-action et le déparseur de paquets. Tous les modules de PISA sont configurables et dépendent du programme que le plan des données doit exécuter. La figure 2.2 présente une architecture PISA composée d'un parseur de paquets, de deux étages de comparaison-action et d'un déparseur. Le parseur reçoit un paquet et en extrait les en-têtes pour les insérer dans le vecteur d'en-têtes de paquet — *Packet Header Vector* (PHV). Les étages de comparaison-action reçoivent le PHV et le modifient. Finalement, le déparseur reçoit le PHV et génère un nouveau paquet en tenant compte des délais dans le traitement des en-têtes. La charge utile est transmise directement depuis le parseur vers le déparseur. Dans le reste de cette section, les différents modules composant PISA sont présentés.

Le parseur de paquets

Le parseur de paquets a pour rôle d'extraire les en-têtes du paquet et de les insérer dans le PHV. Gibb et al. présentent les principes pour la conception d'un parseur de paquets [40]. On considère trois éléments que tous les parseurs de paquets contiennent : un élément pour identifier un en-tête, un bloc pour extraire les champs de l'en-tête et les insérer dans le PHV et finalement un tampon pour stocker le PHV. L'ordre d'extraction des en-têtes ainsi que les éléments permettant d'identifier l'en-tête sont définis à l'aide du Graphe Dirigé Acyclique

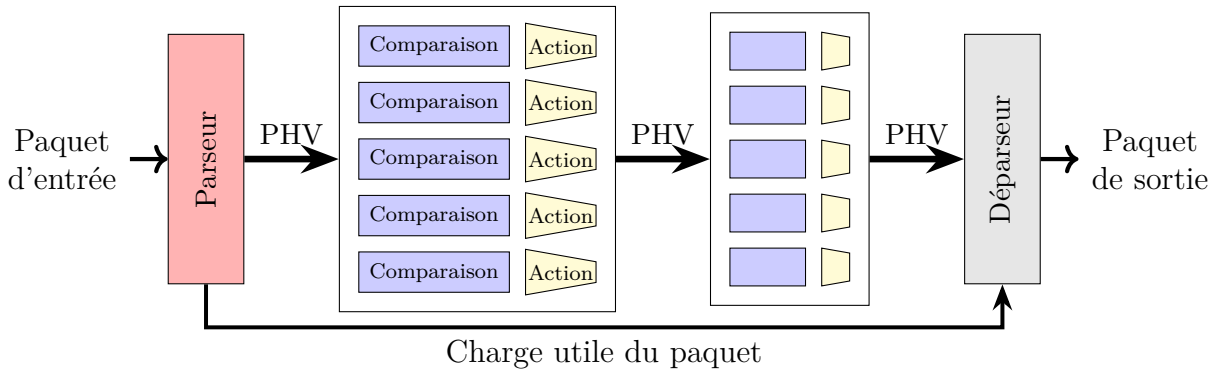


Figure 2.2 Architecture PISA [7] avec deux étages de comparaison-action

(GDA) de parsage. Une fois tous les en-têtes extraits du paquet, le reste du paquet, appelé charge utile, est transmis au déparseur.

L'étage de comparaison-action

L'étage de comparaison-action se compose de deux éléments : le bloc de comparaison et le bloc d'action. Le principe de la comparaison-action consiste à effectuer une action en fonction du résultat d'une comparaison. La table de comparaison associe ainsi une clé à des valeurs et les valeurs contiennent des actions et des paramètres associés aux actions. Une action consiste en une suite d'opérations qui peuvent entre autres modifier les en-têtes du PHV.

Trois types de comparaisons sont possibles : la comparaison exacte, la comparaison ternaire et la comparaison du plus long prefixe — *Longest Prefix Match* (LPM). La comparaison exacte consiste à regarder si la clé recherchée est présente dans la liste de clés de la table. La comparaison ternaire permet de déterminer pour les clés présentes dans la table quelles parties de la clé sont à comparer. Finalement, la comparaison de type LPM consiste à rechercher une portion contiguë de la clé dans la table.

Dans la figure 2.2, deux étages de comparaison-action sont montrés et chacun contient plusieurs tables de comparaison et plusieurs unités pour mettre en œuvre l'action spécifiée. Dans un étage de comparaison-action, plusieurs comparaisons et plusieurs actions peuvent être effectuées en parallèle. Comme PISA est une architecture avec la propagation de données vers l'avant, si des dépendances de données sont présentes, alors la dépendance est résolue en exécutant l'opération sur un étage subséquent. Ainsi, si une comparaison-action B requiert le résultat d'une opération A, alors un étage de comparaison-action va effectuer l'opération A et l'étage de comparaison-action suivant effectuera l'opération B.

Le déparseur

Le déparseur est le dernier module dans le pipeline PISA, il est responsable de la construction du nouveau paquet à émettre. Pour cela, le déparseur reçoit le PHV du dernier étage de comparaison-action ainsi que la charge utile du parseur. La construction du paquet de sortie se fait en combinant les en-têtes modifiés avec la charge utile dans un nouveau paquet. L'ordre d'insertion des en-têtes est défini par le graphe de déparsage.

2.2 Les langages pour les plans des données

Dans cette section, nous présentons des langages permettant de programmer les plans des données. Dans un premier temps, nous résumons l'évolution des langages de programmation pour les plans des données. Par la suite, nous introduisons le langage P4 et ses évolutions.

2.2.1 L'évolution des langages pour le plan des données

Afin de palier à la complexité de programmation des processeurs réseau (Section 2.1.2), des langages dédiés pour les plans des données ont été proposés. Dans cette section nous présentons les langages dédiés tel que défini par Mernik et al., à savoir un langage propre à un domaine ou bien des bibliothèques augmentant l'expressivité d'un langage existant [41].

Click [42] et sa variante, NP-click [43] ciblant plus spécifiquement les processeurs réseaux, sont des exemples de langages dédiés pour le traitement de paquets pour le plan des données. Le langage Click est une biliothèque de classes C++ permettant de définir des modules élémentaires pour le traitement de paquets. Les modules sont connectés entre eux par des ports d'entrées sorties. Les opérations effectuées par les modules dérivent d'opérations de haut niveau des réseaux telles que la vérification de la somme de contrôle d'un en-tête. Pour ajouter de nouveaux protocoles, il faut modifier des classes de la bibliothèque de Click.

En 2014, Bonelli et al. ont proposé PFQ-Lang, un langage fonctionnel pour le traitement des paquets [44]. Le langage permet de définir les traitements à effectuer sur les paquets réseau, cependant, il est dépendant des protocoles intégrés. Ainsi, pour pouvoir utiliser un nouveau protocole réseau, il est nécessaire de modifier le langage afin d'y insérer le nouveau protocole.

Les langages Click, NP-click, PFQ-Lang fournissent une abstraction de haut niveau pour programmer des applications réseau. Toutefois, ces langages ne permettent pas de programmer un plan des données personnalisé. En effet, pour intégrer de nouveaux protocoles, il faut modifier ces langages.

En 2009, Duncan et Jungck ont proposé packetC, un langage dérivé de la norme C99 adapté au traitement de paquets [45]. Le langage consiste à définir un programme pour le traitement de paquets, et ce programme est ensuite exécuté sur plusieurs fils d'exécution. Bien que le langage autorise certaines abstractions propres au réseau, telles que les ensembles de recherche, le langage reste de très bas niveau et requiert une bonne connaissance, par le programmeur, du matériel sur lequel le programme est implémenté pour être performant. Le langage requiert également une architecture de processeur réseau massivement multicœur dans lesquel chaque cœur traite un paquet.

En 2013, Song a proposé POF, un langage dérivé de la spécification d'OpenFlow [46]. Le langage peut être considéré comme un jeu d'instructions adapté aux plans des données réseau et indépendant de la plateforme cible du programme. Les instructions disponibles dans POF peuvent être séparées en deux classes : les instructions de recherche et les instructions de calculs. Un élément majeur de POF est d'être indépendant des protocoles, ce qui permet de le distinguer des langages précédents. Le langage contient toutefois des instructions spécialisées comme le calcul de la somme de contrôle IPv4. La présence d'instructions spécialisés implique que lors de la création d'un processeur réseau, ces instructions doivent être insérées.

Brebner et Jiang ont développé le langage PX en 2014. Ce langage cible les FPGA Xilinx et permet de définir un pipeline de traitement pour des paquets. Le langage est indépendant des protocoles et il propose une structure pour exprimer le parseur des en-têtes du paquet et une structure différente pour le traitement à effectuer sur les en-têtes avec notamment des table de comparaison.

Les langages packetC, POF et PX permettent de programmer un plan des données et d'y insérer de nouveaux protocoles. PacketC et POF contraignent la conception des cibles sur lesquelles les programmes sont exécutés. Dans le cas de packetC, il faut que la cible soit massivement multicœur. Pour pouvoir être compatible avec POF, une cible doit pouvoir exécuter certaines opérations spécialisées. Le langage de Xilinx, PX, est propriétaire et n'est donc utilisable qu'avec des FPGA Xilinx.

2.2.2 Le langage P4

La première version du langage P4, $P4_{14}$, a été proposée en 2014 par Bosshart et al., il s'agit d'un langage qui permet d'exprimer le traitement à effectuer sur un paquet dans un plan des données réseau [8]. Le consortium P4 est responsable du maintien de la spécification du langage ainsi que du compilateur de référence.

P4 est un langage impératif basé sur le modèle de comparaison-action. Le modèle d'exécution s'organise autour de l'architecture PISA. On retrouve donc dans le langage P4 cinq composantes principales :

— la définition des en-têtes ;
— le parseur de paquets ;
— la définition des actions ;
— les tables de comparaison ; et
— le bloc de contrôle.

En P4, le vecteur d'en-têtes est une structure composée de champs de type `header`. Le type `header` est lui-même une structure pouvant être définie par le programmeur et contenant toujours un champ booléen dit de validité de l'en-tête. Ce champ de validité est caché, et il existe trois méthodes pour y accéder : `isValid`, `setValid` et `setInvalid`. Le champ de validité est initialisé à faux au démarrage du programme.

Le parseur de paquets est défini comme une machine à états. Le rôle du parseur est d'extraire du paquet les en-têtes qui sont traités dans le programme. Lors de l'extraction d'un en-tête, les données de ce dernier sont insérées dans un des champs du vecteur d'en-tête, et le champ de validité de cet en-tête est mis à vrai. Le parseur permet ainsi de définir les en-têtes qui seront traités par le commutateur.

Les actions sont des éléments qui peuvent être comparés à des procédures. Les actions peuvent avoir des paramètres et contiennent une séquence d'opérations élémentaires à effectuer, ces opérations peuvent notamment modifier le champ d'un en-tête.

Une table est un conteneur défini dans un bloc de contrôle. Le programme P4 permet de définir la structure de la table mais ne peut pas modifier son contenu. La modification du contenu d'une table s'éffectue uniquement par le plan de contrôle pendant l'exécution du programme. La définition d'une table se compose de quatre éléments :

— la clé ;
— les actions ;
— le nombre d'entrées ; et
— une action par défaut.

La clé se compose d'un ou plusieurs éléments sur lesquels on doit effectuer la comparaison, pour chacun de ces éléments, un type de comparaison y est associé. Les actions indiquent quelles actions définies dans le code peuvent être exécutées par la table. La taille indique le nombre d'entrées que la table contient. L'action par défaut permet de définir l'action à exécuter si aucune correspondance n'est trouvée lors de la recherche dans la table.

Le bloc de contrôle permet de programmer le flot de traitement dans le programme. Les tables et les actions sont définies dans un bloc de contrôle et sont également appliquées dans ce dernier. Dans le bloc de contrôle, on peut organiser l'ordre dans lequel les comparaisons-actions sont effectuées ainsi que les conditions qu'il faut rencontrer pour effectuer ces opérations. Également, d'autres opérations peuvent être faites comme la modification du champ d'un en-tête, notamment pour le champ de validité des en-têtes permettant ainsi d'insérer ou de retirer un en-tête du paquet.

En 2016, le langage P4 a été révisé en profondeur [48], et la nouvelle version du langage est appelée $P4_{16}$. $P4_{16}$ a pour objectif de stabiliser le langage, tout en permettant d'ajouter de nouvelles fonctionnalités aux commutateurs. Pour cela, le langage n'est plus spécifique à une architecture, il est possible d'intégrer des opérations non disponibles dans le langage à l'aide de fonctions dite externes. Également, une modification importante entre $P4_{14}$ et $P4_{16}$ est le fait que le déparseur de paquet n'est plus dépendant du parseur. En effet, en $P4_{14}$, le déparseur est le symétrique du parseur, alors qu'avec $P4_{16}$ le déparseur est défini dans un bloc de contrôle dans lequel on indique l'ordre d'émission des en-têtes. Ceci implique qu'un programme $P4_{16}$ peut émettre un paquet que le commutateur ne peut pas parser.

2.3 Les architectures P4

Avec $P4_{16}$, il est possible de définir différentes architectures de commutateurs. Dans ce contexte, une «architecture» est un modèle de programmation indiquant au concepteur les capacités ainsi que l'organisation logique du pipeline de traitement de la cible à programmer. Un programme P4 est donc développé pour une architecture spécifique. Toutefois, le programme doit être compilé pour les différentes cibles sur lesquels il est exécuté. Par exemple, si un programmae P4 est exécuté sur une cible logiciel et un FPGA, il sera nécessaire de compiler le programme deux fois même s'il utilise la même architecture.

Une architecture peut contenir certaines parties non programmables en P4. Également, la définition d'une architecture peut définir un ensemble de fonctionnalités externes que le programme P4 peut implémenter à l'aide de fonctions externes définies avec le mot-clé `extern`. On dénombre quatre architectures courantes dans le cas de P4 :

- `v1model` ;
- architecture portable de commutateurs — *Portable Switch Architecture* (PSA) ;
- architecture native du Tofino — *Tofino Native Architecture* (TNA) ; et
- `SimpleSumeSwitch`.

L'architecture `v1model` est l'architecture $P4_{16}$ correspondante à l'architecutre utilisée en $P4_{14}$. Tel que montré dans la figure 2.3, elle se compose d'un parseur suivi d'un pipeline

de comparaison-action puis d'un gestionnaire de trafic, d'un pipeline de comparaison action et d'un déparseur. La présence de cette architecture permet au compilateur P4c de convertir des programmes $P4_{14}$ vers $P4_{16}$. D'autre part, il s'agit également de la première architecture supportée par le modèle comportemental de référence modèle comportemental version 2 — *Behavioral Model version 2* (BMv2).

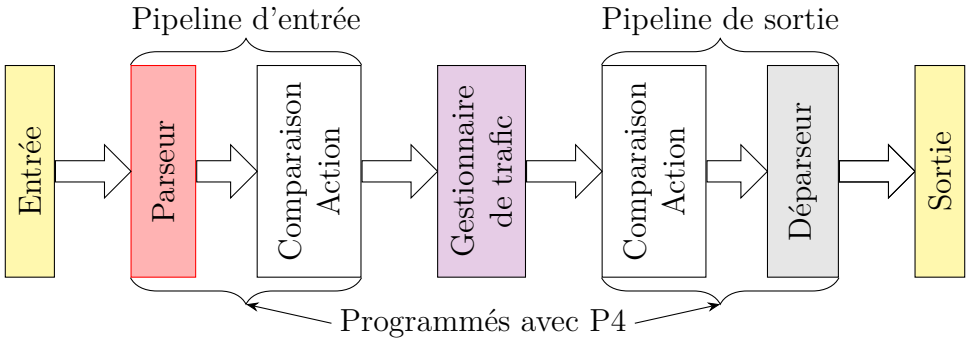


Figure 2.3 Vue d'ensemble de l'architecture P4 `v1model`

La figure 2.4 montre l'architecture PSA, qui se compose de deux pipelines, un d'entrée et un de sortie, séparés par un gestionnaire de trafic [49]. Les pipelines d'entrée et de sortie sont identiques, ils sont composés d'un parseur, d'un pipeline de comparaison-action et d'un déparseur. Seulement ces pipelines sont pour le moment programmables en P4. Le gestionnaire de trafic n'est actuellement pas programmable en P4, cependant des recherches sont en cours pour essayer d'intégrer au langage P4 la programmation de ce module [50].

L'architecture TNA est l'architecture P4 propritétaire du Tofino™, qui est une version étendue de PSA. Comme les puces Tofino™ peuvent avoir deux ou quatre unités de traitement, l'architecture TNA contient deux à quatre pipelines PSA [51]. Également l'architecture TNA définit certaines fonctions externes qui ne sont pas définies dans l'architecture PSA [52].

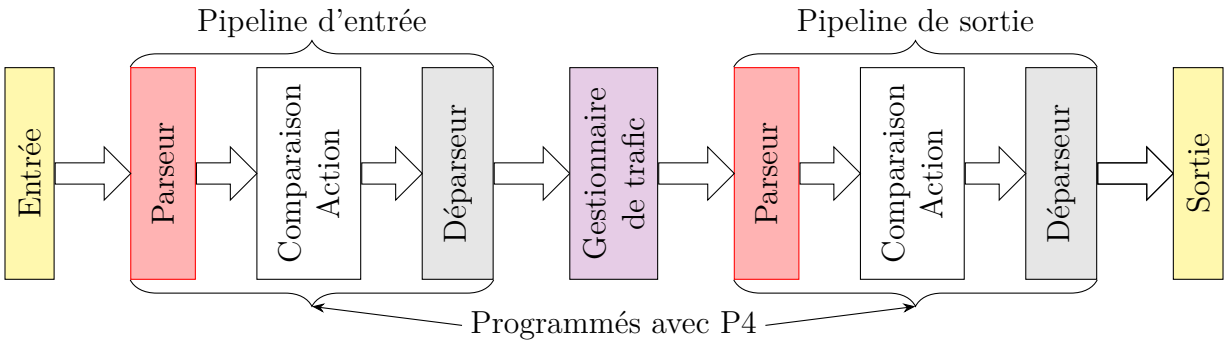


Figure 2.4 Vue d'ensemble de l'architecture P4 PSA [49]

L'architecture `SimpleSumeSwitch`, présentée dans la figure 2.5, contient un pipeline composé d'un parseur, d'un étage de comparaison-action, d'un déparseur et d'un gestionnaire de trafic [50]. Comme pour PSA, le gestionnaire de trafic ne peut pas être programmé en P4. Seul le parseur, l'étage de comparaison-action et le déparseur sont programmés en P4. L'architecture `SimpleSumeSwitch` est généralement l'architecture utilisée pour implémenter des programmes P4 sur FPGA [16, 17, 25, 53].

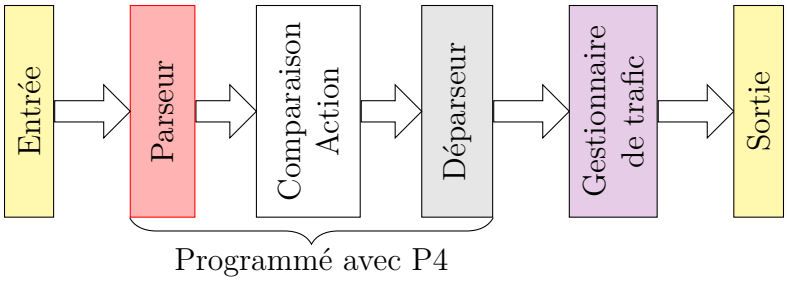


Figure 2.5 Présentation de l'architecture P4 `SimpleSumeSwitch` [25]

2.4 Compilation de P4

Les compilateurs P4 traduisent un programme P4 dans un langage qui pourra être exécuté sur une cible spécifique. Tout d'abord, nous présentons la structure couramment utilisée par les compilateurs P4. Par la suite, nous présentons différentes cibles ayant des compilateurs P4. Ensuite, nous discutons de différents travaux traitant de l'analyse de programmes P4. Finalement, nous explorons des techniques permettant d'améliorer la compilation de langages dédiés.

2.4.1 La structure d'un compilateur P4

Les compilateurs P4 se basent généralement sur un modèle à deux couches. La première couche est une partie frontale commune à tous les compilateurs et la partie dorsale est spécifique à une cible [52].

La partie frontale permet de convertir le code P4 dans la représentation intermédiaire et effectue les analyses sémantiques indépendantes des architectures et des cibles. La représentation intermédiaire est par la suite utilisée par la partie dorsale pour effectuer les transformations spécifiques à l'architecture et à la cible. Comme plusieurs cibles peuvent utiliser la même architecture, la partie dorsale est composée de deux parties, une partie pour effectuer des modifications basées sur des politiques propres à l'architecture et une partie qui effectue les

transformations propres à la cible. Afin de pouvoir réutiliser certaines transformations de la représentation intermédiaire, des bibliothèques de passes de transformations sont disponibles [54].

La représentation intermédiaire utilisée est celle fournie par le compilateur de référence P4c [55]. La représentation intermédiaire est basée sur des objets C++ et elle peut être exportée en P4 ou dans un fichier au format JSON [54]. La communauté P4 propose un ensemble de compilateurs P4 de référence pour différentes cibles telles que le modèle comportemental BMv2, eBPF ou bien uBPF.

2.4.2 Les cibles pour la compilation de P4

On distingue trois catégories de cibles vers lesquelles le code P4 est compilé : les processeurs à usage général, les cibles matérielles et les FPGA. Dans cette section, nous commençons par présenter les différents compilateur P4 à destination des processeurs à usage général. Par la suite, nous explorons différentes solutions matérielles étant compatibles avec P4. Finalement, nous présentons différents compilateurs P4 pour FPGA.

Compilation et exécution de P4 sur des processeurs à usage général

Il y a deux approches de compilation pour l'exécution du code P4 sur des processeurs à usage général. Une première méthode est la compilation du code P4 dans un langage qui est interprété. La seconde méthode est la compilation du code P4 vers un code exécuté par un processeur à usage général.

Plusieurs solutions exécutants une représentation intermédiaire d'un code P4 existent. Le BMv2 [56], est le logiciel proposé par la communauté P4 afin de tester des programmes P4. Ce commutateur logiciel charge un fichier JSON qui est compilé avec le compilateur P4c de référence. Trois autres solutions, `p4c-ebpf` [57], `p4c-ubpf` [58] et `p4c-xdp` [59], compilent des programmes P4 vers une représentation eBPF [60] afin de les exécuter directement avec des outils intégrés au noyau Linux.

Il existe également des solutions pour la compilation de programmes P4 vers des fichiers binaires exécutés sur des processeurs à usage général. Shahbaz et al. proposent PISCES, un commutateur logiciel qui est généré à partir d'un code P4 compilé en C [61]. Zanna et al. proposent de compiler des codes P4 pour être exécutés sur la carte d'évaluation et de test ZodiacFX [62, 63]. Finalement, Vörös et al. proposent T_4P_4S pour compiler des codes P4 vers du code C indépendant de la cible [64].

Compilation pour des puces spécialisées programmables

Il existe des compilateurs P4 pour des puces spécialisées programmables. Intel® Tofino™ est la première puce spécialisée pouvant être programmée avec P4 à avoir été proposée. La première version pouvait traiter un débit agrégé de 6,5 Tbit/s [9], la dernière version de la puce, Tofino™ 3 [65], peut traiter jusqu'à 25,6 Tbit/s de trafic réseau agrégé. Intel® fournit un compilateur P4, Intel® P4 Studio. Seibulescu et Baldi proposent un compilateur P4 pour le processeur de Pensando [66, 67]. Les auteurs utilisent des fonctions externes qu'ils ajoutent à l'architecture P4 utilisée afin de pouvoir utiliser les instructions spécialisées disponibles sur le processeur. Finalement, Netronome [68], propose aussi un compilateur P4 pour leur NIC.

L'implémentation de P4 sur FPGA

Plusieurs solutions ont été proposées pour compiler des programmes P4 afin de les implémenter sur des FPGA. Wang et al. proposent P4FPGA, un compilateur qui traduit un programme P4 en BlueSpec SystemVerilog qui peut ensuite être traduit en Verilog synthétisable par le compilateur propriétaire BlueSpec [16]. Benáček et al. proposent un compilateur qui convertit directement en VHDL le code P4 [17]. Finalement, Xilinx® propose le compilateur Xilinx SDNet [53] qui permet de convertir un code P4 en VHDL pour des FPGA Xilinx. Ibanez et al. présentent P4 → NetFPGA, une méthode permettant de compiler des codes P4 vers la plateforme NetFPGA [25]. Cette méthode utilise le compilateur Xilinx SDNet pour compiler le code P4 vers du VHDL.

D'autres travaux pour la compilation de P4 sur FPGA se sont plus axés sur les aspects de microarchitecture pour l'implémentation de structure PISA. L'implémentation du parseur de paquets sur FPGA a été explorée [27–30]. Les résultats montrent qu'il est possible de faire des parseurs de paquets pouvant soutenir des débits de plus de 100 Gb/s. Kekely et Korenek étudient l'implémentation de tables de comparaison-action définies en P4 sur un FPGA [69]. Stimpfling explore l'implémentation de mémoires associatives de type LPM définies en P4 sur FPGA [70]. Pontarelli et al. proposent une approche semblable aux processeurs RISC pour implémenter des actions définies en P4 sur FPGA [71]. Cabal et al. explorent l'implémentation du déparseur P4 sur FPGA [72], l'architecture est extraite du travail de Benáček et al. [17]. Le travail proposé se base cependant sur $P4_{14}$, une version de P4 dans laquelle le graphe de déparsage est dérivé du graphe de parsage.

2.4.3 L'analyse de programmes P4

Dans la section précédente, on présente un ensemble de compilateurs P4 permettant de cibler une variété de plateformes pour l'exécution du code. Dans cette section, nous présentons les travaux orientés sur l'analyse et l'optimisation des applications P4.

Jose et al. s'intéressent à l'allocation des ressources pour PISA dans le cas de commutateurs spécialisés comme le Tofino [73]. Les auteurs introduisent le concept de graphe de dépendance de tables — *Table Dependency Graph* (TDG). Il s'agit d'un Graphe Dirigé Acyclique (GDA) dans lequel les sommets représentent les tables et les arêtes représentent la dépendance entre les tables dans le flot de contrôle du programme. Jose et al. montrent que l'allocation de ressources peut être faite en utilisant un algorithme d'Optimisation Linéaire en Nombres Entiers (OLNE).

Chole et al. proposent une variante au TDG en introduisant le concept de graphe de dépendance des opérations — *Operation Dependency Graph* (ODG) [74]. La différence entre les deux approches est principalement liée à l'architecture cible. En effet, dans [74], l'architecture est orientée autour de processeur qui accèdent à des mémoires de comparaison alors que l'approche de [73] cible des commutateurs avec des mémoires de comparaison organisée dans un pipeline. Les deux solutions peuvent toutefois utiliser des OLNE afin de résoudre le problème d'allocation des ressources.

Vass et al. analysent les aspects algorithmiques de la compilation de programmes P4 [75]. Les auteurs caractérisent les aspects algorithmiques notamment en ce qui concerne la complexité, afin d'évaluer les techniques actuellement utilisées pour la compilation de programme P4. Le résultat de leur analyse est que la compilation d'un programme P4 peut se faire dans un temps presque linéaire avec la taille du programme.

Gibb et al. présentent une solution pour faire des parseurs de paquets configurables, l'extraction des en-têtes se fait par le parcours d'un GDA dans lequel les sommets représentent un en-tête et les arêtes indique l'en-tête suivant à extraire en fonction d'une condition [40]. Santiago da Silva et al. proposent de transformer le graphe de parsage afin de simplifier le pipeline implémenté sur le FPGA [28].

Dangeti et al. utilisent la représentation intermédiaire de LLVM pour compiler un programme P4 [76]. Le compilateur proposé convertit la représentation intermédiaire de p4c vers la représentation intermédiaire de LLVM, puis génère un fichier JSON après les phases d'optimisations propres à LLVM. Avec ce travail, les auteurs montrent que LLVM s'adapte bien à la compilation de code P4 et que les optimisations existantes dans LLVM sont utiles à l'optimisation de code P4.

Ma et al. proposent d'insérer une table de comparaison-action comme cache à l'entrée du traitement à l'aide d'un préprocesseur sur le code P4 [77]. Cette cache permet d'accélérer le traitement de certains flots à l'exécution. Cette approche requiert cependant d'avoir des mécanismes afin de contourner les étages de comparaison-action n'étant plus utile à évaluer. Abhashkumar et al. proposent d'intégrer des contraintes liées aux applications lors de la compilation du code P4 [78]. Les contraintes sont indiquées par le programmeur et permettent notamment d'enlever des dépendances dans le TDG. Wintermeyer et al. proposent comme Abhashkumar et al. une approche pour prendre en compte les contraintes des applications en analysant des traces de trafic réseaux lors de la compilation. Cette approche permet ainsi de ne pas avoir à insérer les contraintes comme Abhashkumar et al., mais requiert d'avoir des traces réseaux valides.

2.5 Conclusion

L'utilisation du langage P4 et l'architecture PISA pour implémenter des applications réseau sur FPGA pourrait aider à l'utilisation des FPGA comme NIC dans les centres de données. En effet, Firestone et al. ont montré que les FPGA sont des plateformes d'intérêt pour réaliser des NIC programmables, mais qui requierent de bonnes connaissances en développement matériel pour être utilisé, et cela malgré les outils de synthèse de haut niveau [14]. La question qui se pose alors est de savoir s'il est possible de faire des compilateurs qui implémentent efficacement des applications P4 sur les FPGA. Nous étudions cet aspect dans le chapitre 3 de cette thèse.

De plus, on constate à travers les nombreux travaux effectués que la compilation de programmes P4 sur une cible est efficace si l'on connaît la microarchitecture de celle-ci. Santiago da Silva et al. ont montré que dans le cas des FPGA, une transformation du graphe de parsage et une architecture sur FPGA adaptée permettent d'améliorer le résultat d'implémentation d'un parseur [28]. L'implémentation efficace d'un code P4 sur FPGA nécessite donc un travail sur le graphe du programme associé à une architecture adaptée au FPGA. On peut donc se demander si, dans le cas du déparseur sur FPGA, il peut exister une architecture adaptée aux FPGA. C'est une question à laquelle nous apportons une réponse dans le chapitre 4 de cette thèse.

Finalement, le langage P4 et l'architecture PISA ont permis de rendre programmable le plan des données du réseau. Malgré la variété d'architectures ciblées pour les programmes P4, les blocs nécessaires au traitement de paquets sont bien définis. L'arrivée de $P4_{16}$ a cependant ajouté un module essentiel lors de la compilation d'un programme P4, le déparseur. Cette différence est significative, car en $P4_{14}$, le graphe de déparsage dérive du graphe de parsage

et le déparseur bénéficie donc des transformations effectuées sur le graphe de parsage. Une question se pose alors : le graphe de déparsage impacte-t-il l'implémentation du déparseur ? Nous proposons une réponse à cette question dans le chapitre 5 de cette thèse.

CHAPITRE 3 IMPLÉMENTER DES PROGRAMMES P4 SUR FPGA

Dans ce chapitre, nous présentons des considérations relatives à l'implémentation de programmes P4 sur des FPGA. Nous cherchons à déterminer si les FPGA sont d'une part des cibles adaptées pour l'implémentation de programmes P4 et d'autre part si des améliorations sont possibles afin d'améliorer la prise en charge de programmes P4 sur des cibles FPGA. Comme les programmes P4 exploitent une architecture de commutateur indépendante des protocoles — *Protocol Independent Switch Architecture* (PISA), nous évaluons comment les différents blocs de PISA s'implémentent sur un FPGA. Nous commençons par analyser comment les structures élémentaires de PISA s'implémentent sur les FPGA actuels (Section 3.1). Puis, nous évaluons l'impact de différents paramètres d'implémentation de plans des données programmables sur les performances obtenues (Section 3.2). Finalement, nous proposons des solutions pour améliorer l'implémentation de programmes P4 sur FPGA (Section 3.3). La présente étude a été présentée à la conférence HPSR'20 [80].

3.1 Analyse de l'implémentation de PISA sur FPGA

Dans un pipeline PISA, le traitement d'un paquet se décompose en trois étapes : l'extraction des en-têtes, le traitement des en-têtes et la recomposition du paquet. L'extraction des en-têtes se fait dans le parseur. Le traitement des en-têtes s'implémente en intégrant différents blocs de Comparaison-Action. Un bloc de Comparaison-Action se compose d'une table de comparaison et d'un bloc d'exécution des actions spécifiées. La recomposition d'un paquet à émettre se fait dans le déparseur.

Dans cette section, nous analysons comment ces différents éléments du pipeline PISA s'intègrent dans un FPGA. Dans un premier temps, nous regardons comment s'implémente les parseurs de paquets (Section 2.1.3). Puis nous évaluons comment s'implémentent les différents types de tables de comparaisons sur FPGA (Section 3.1.2). Par la suite, nous explorons la prise en charge des opérations fondamentales composant les actions (Section 3.1.3). Finalement, nous nous intéressons à l'implémentation du déparseur sur FPGA (Section 3.1.4).

3.1.1 Le parseur de paquets

Le parseur de paquets identifie et extrait les en-têtes d'un paquet qui sont traités dans le reste du pipeline du commutateur. Gibb et al. décomposent la structure d'un parseur en trois blocs de base : l'identifieur, l'extracteur et une mémoire tampon [40]. L'identifieur permet de

déterminer l'en-tête à extraire à l'aide du graphe de parsage. Pour cela, le module requiert une machine à états. L'extracteur sélectionne les bits à extraire avec des multiplexeurs et une table de correspondance pour indiquer quelle entrée les multiplexeurs doivent sélectionner. La mémoire tampon est un vecteur de registres. Pour implémenter un parseur, il faut donc implémenter une machine à états, des multiplexeurs et des registres.

Ces différents blocs nécessaires à l'élaboration d'un parseur s'implémentent efficacement sur un FPGA. Une machine à états se compose d'une mémoire et d'éléments de calculs afin d'évaluer les transitions d'états. Sur les FPGA, l'implémentation de la mémoire peut se faire par des combinaisons de Blocs RAM (BRAM) et LUTRAM. L'évaluation des transitions d'états s'effectue en configurant les tranches du FPGA. Dans le cas du parseur, la transition d'états consiste principalement en la comparaison de la valeur d'un champ d'en-tête avec des valeurs définies lors de la compilation. Cette opération s'implémente efficacement dans les tranches du FPGA. Les bascules (FF) sont des éléments de base composant le FPGA. Les multiplexeurs s'implémentent aisément sur les tranches du FPGA [81]. Il en résulte que les FPGA actuels peuvent implémenter efficacement un parseur de paquet tel que montré à plusieurs reprises [17, 28, 30].

3.1.2 Les tables de comparaisons

Les tables de comparaison sont des conteneurs associant une clé à une valeur. La clé est recherchée pour trouver la valeur associée. En général, le résultat de la comparaison retourne un indice ou une adresse permettant de lire une mémoire à adressage direct contenant la valeur. Les différents types de table de comparaisons déterminent comment la clé en entrée est comparé avec les clés dans la mémoire. Le langage P4 permet trois types de table de comparaisons : la comparaison exacte, la comparaison ternaire et la comparaison du plus long prefixe — *Longest Prefix Match* (LPM). Dans cette section nous examinons comment ces trois types de tables peuvent être implémentées dans les FPGA actuels.

Comparaisons exactes

L'implémentation de la comparaison exacte se fait traditionnellement en utilisant des mémoires adressables par contenu — *Content Adressable Memory* (CAM). Une CAM compare la clé d'entrée avec toutes les clés qu'elle contient et retourne la valeur associée. Les FPGA ne contiennent pas de CAM, il faut donc les implémenter à partir des ressources disponibles sur le FPGA. Deux approches populaires pour l'implémentation d'une CAM sur un FPGA sont les mémoires transposées et les tables de hachage [82–84].

L'implémentation d'une CAM à l'aide de mémoires transposées consiste à utiliser l'adresse comme une clé. La valeur associée est alors enregistrée comme une donnée. L'avantage de cette solution est qu'elle permet d'obtenir le résultat et de mettre à jour la mémoire en un seul cycle, car il s'agit d'une simple opération de lecture ou d'écriture. Cependant, cette approche a une efficacité mémoire d'environ 10 %, il faut donc 10 bits de mémoire par bit de clé. Cela rend l'implémentation de CAM à l'aide de mémoires transposées coûteuse [82, 83].

La seconde approche pour implémenter des CAM est l'utilisation du hachage Cuckoo [84]. Cette approche consiste à calculer un haché de la clé à rechercher pour générer une adresse mémoire. Comme le haché est plus petit que la taille de la clé, il y a un risque de collision, à savoir deux clés différentes qui génèrent le même haché. Pour résoudre les collisions, plusieurs fonctions de hachage sont utilisées et associées à différentes mémoires. Ainsi lors d'une collision, on déplace une des clés dans une autre mémoire avec un autre haché. Pour retrouver la donnée, on compare cette dernières à toutes les données enregistrées en mémoire, ayant le même haché que la clés. Il a été montré que cette approche permet d'atteindre une efficacité mémoire de plus de 80 % [7, 85]. Cette approche requiert toutefois de gérer les collisions qui sont résolues par le déplacement des données entre les mémoires. Lors d'une collision, le logiciel responsable de l'insertion déplace la donnée déjà présente dans une autre mémoire associée à une autre table de hachage. Si la mémoire de destination possède déjà une entrée, on résout cette nouvelle collision de la même manière. On répète la résolution de collision jusqu'à ce qu'une insertion ne génère pas de collision. Ainsi, plus la table contient un nombre important d'entrées utilisées, plus le nombre de déplacements à effectuer lors de l'insertion d'une donnée risque d'être important. Par conséquent, l'utilisation du hachage Cuckoo rend l'insertion de données dans la table plus lente qu'avec des CAM.

Les deux approches présentées s'implémentent facilement dans un FPGA. Dans le cas des mémoire transposées la solution requiert uniquement des mémoire RAM. Dans le cas du hachage Cuckoo, il faut des mémoires RAM et pouvoir calculer des hachés. Les FPGA contiennent des mémoires à accès aléatoire statiques — *Static Random Access Memory* (SRAM) en quantité importante. Ces mémoires sont requises pour ces deux solutions d'implémentation de tables de comparaisons exactes. Pour calculer les hachés, l'opération est implémentées dans les tranches, il existe des fonctions de hachage qui requièrent peu de ressources du FPGA [86]. Les comparaisons exactes s'implémentent donc efficacement sur FPGA.

La comparaison ternaire

Dans la comparaison ternaire seulement des parties des clés contenues sont comparées à la clé en entrée. Ainsi, pour chaque donnée dans la mémoire, on associe un masque et une priorité.

Le masque indique quelles parties de la clé sont à considérer lors de la recherche. La priorité permet d'itentifier la valeur à retourner si plusieurs entrées de la mémoire adressable par contenu ternaire — *Ternary Content Adressable Memory* (TCAM) valides la clé en entrée. Le module matériel pour implémenter les comparaisons ternaires est la TCAM, qui n'est pas présent sur les FPGA et qui doit donc être implémentée.

Une approche pour implémenter une TCAM de largeur L et de profondeur N est de la subdiviser en P TCAM plus petites et de les implémenter en utilisant des mémoires transposées. L'implémentation d'une plus petite TCAM s'effectue en utilisant le bus d'adresse comme la clé. Chaque TCAM plus petites a donc une clée de largeur $w = L/P$. Le coût en mémoire de cette approche est de $2^w/w$ [87]. Il en résulte que le coût est minimal si w est égal à 1 ou 2, ce qui requiert d'avoir des blocs mémoires pour faire la transposition de profondeur 2 ou 4. Les FPGA contiennent deux types de blocs mémoire, soit les BRAM qui ont une profondeur minimum de 512 ou bien les LUTRAM avec une profondeur minimum de 32 [88, 89]. Par conséquent, dans la pratique l'implémentation d'une TCAM sous forme de mémoires transposées nécessite entre $8,5\times$ à $65\times$ plus de mémoire que nécessaire.

Reviriego et al. proposent de directement synthétiser les entrées de la TCAM dans les tables de conversion — *LookUp Table* (LUT) et d'utiliser la reconfiguration partielle du FPGA pour modifier les entrées [90]. Cette approche permet une meilleure utilisation des ressources mémoires que l'utilisation de mémoire transposée, les auteurs indiquent une division par 2 par rapport à l'approche des mémoires transposées sur LUTRAM. Comme les LUT sont des SRAM, cette approche résulte dans une efficacité mémoire de l'ordre de 25 %, ce qui reste faible. De plus la reconfiguration partielle est lente, par exemple il faut 7 ms pour reconfigurer une TCAM de 32 entrées avec une largeur de 40 bits. Dans des réseaux à très haut débit, ce temps de reconfiguration peut s'avérer très limitant. L'implémentation de TCAM performantes et efficientes sur FPGA reste donc un problème ouvert.

LPM

La comparaison du plus long prefixe — *Longest Prefix Match* (LPM) est un sous cas de la comparaison ternaire. Dans la LPM, on applique un masque qui permet d'ignorer, lors de la recherche, une partie des bits les moins significatifs. Si plusieurs résultats sont trouvés, on sélectionne le préfixe le plus long, donc le résultat qui a le moins de bits ignorés. Comme la LPM est un sous cas de la comparaison ternaire, les tables LPM sont souvent implémentées avec des TCAM. Cependant, comme montré précédemment, les TCAM ne s'implémentent pas efficacement sur les FPGA [83].

Il est toutefois possible d'utiliser des structures de données telles que des arbres binaires pour implémenter des tables de LPM, tel que proposé par Xilinx [91]. La solution de Xilinx permet de doubler la fréquence d'horloge par rapport aux approches utilisant des mémoires transposées, tout en réduisant significativement la consommation de ressources. Toutefois, le temps de mise à jour est proportionnel au nombre d'entrées présentes dans la table.

Finalement, plusieurs solutions proposées dans la littérature consistent à prendre parti des caractéristiques spécifiques des règles contenues dans la table [92–94]. Ces approches permettent une meilleure efficacité mémoire. Cependant, dans le cas des plans des données programmables, les données à insérer dans les tables ne sont pas connues à l'avance.

3.1.3 Les actions

En P4, les actions sont éxprimées comme des compositions de primitives. On distingue trois grandes classes de primitives en P4 : les opérations arithmétiques et logiques, les structures conditionnelles et le décalage de bits.

Les opérations arithmétiques et logiques

À l'exception de l'opération de division, les opérations arithmétiques et logiques s'implémentent efficacement dans un FPGA [82]. De plus il y a des structures intégrées aux FPGA permettant de supporter des additions et des multiplications rapides à l'aide de blocs DSP et des chaînes de retenus dédiées. L'opération de division est toutefois coûteuse en termes de ressources, mais est rarement utilisée dans le cas des applications de traitement de paquet. Finalement, le tissu d'interconnexions du FPGA permet de connecter ses différentes ressources dans un pipeline et donc de construire des actions plus complexes. Ainsi il est possible d'envisager le support d'une grande variété de combinaisons d'opérations arithmétiques et logiques sur les FPGA.

Les structures conditionnelles

L'implémentation d'une structure conditionnelle dans un FPGA se fait par l'implémentation d'un multiplexeur. Les multiplexeurs s'implémentent efficacement sur les FPGA. Une tranche d'un FPGA Xilinx peut implémenter un multiplexeur 16 :1, il faut cependant combiner plusieurs tranches si plus de possibilités sont à prendre en compte[81]. Ceci peut alors limiter la fréquence maximale pouvant être atteinte.

Le décalage de bits

Deux cas sont à prendre en compte pour l'implémentation du décalage de bits : le décalage connu à la compilation et le décalage connu à l'exécution. Lorsque le décalage est connu à la compilation, il s'agit d'une opération non coûteuse sur un FPGA, car il s'agit d'une simple opération de connexions. Lorsque le décalage à effectuer est connu uniquement lors de l'exécution, cela requiert d'implémenter un décaleur barillet. Les décaleurs barillet sont en général implémentés sur FPGA en utilisant des chaînes de multiplexeurs. Bien que le multiplexeur s'implémente bien sur le FPGA, le nombre très important de multiplexeurs requis amène à une utilisation importante des ressources. Cette opération est toutefois peu commune dans le traitement des paquets.

3.1.4 Le déparseur de paquets

Le déparseur effectue l'inverse du parseur, il insère les en-têtes dans le paquet à émettre. L'insertion des en-têtes s'effectue selon un ordre précis connu au moment de l'implémentation. Comme il s'agit d'effectuer l'inverse de l'opération de parsage, les méthodes permettant d'implémenter efficacement un parseur sur un FPGA peuvent être réutilisées. L'implémentation d'un déparseur consiste donc à sélectionner les données à insérer au bon moment et à la bonne place les en-têtes du nouveau paquet. Cette opération se fait par l'implémentation d'une machine à états et la sélection et le décalage fixe de bits. La sélection de bits s'implémente à l'aide de multiplexeurs et le décalage fixe de bits se fait par une connexion entre les modules. Toutefois, on constate que les différents déparseurs proposés consomment beaucoup de ressources en comparaison des parseurs. Par exemple le déparseur proposé par Benáček et al. utilise de 3 à 4 fois plus de tranches que leur parseur et il représente plus de 75 % des ressources utilisées par une application complète [17].

En conclusion, l'implémentation du déparseur sur FPGA reste un problème ouvert, mais devrait pouvoir s'effectuer avec les architectures de FPGA actuelles. En effet, l'implémentation d'une machine à états est efficace sur FPGA. La sélection de bits se fait à l'aide de multiplexeur qui s'implémente également bien sûr FPGA. Finalement, le décalage fixe de bits consiste en un routage des connexions qui peut être considéré quasi gratuite sur un FPGA.

3.2 Évaluation des FPGA comme commutateurs réseaux

Les FPGA actuels peuvent implémenter des modules Ethernet de plus de 100 Gb/s et atteindre un débit agrégé de plus de 1 Tb/s [95]. Ces très hauts débits supportés par les FPGA ne prennent pas en compte le traitement possible sur la puce. Il se pose donc la question des

traitements qu'un FPGA peut effectuer sur les paquets tout en supportant les hauts débits demandés.

Dans cette section, nous proposons une évaluation de l'implémentation de différents programmes P4 sur FPGA afin de définir les limites de l'implémentation de PISA sur FPGA. Dans un premier temps, nous présentons la méthode d'évaluation (Section 3.2.1). Puis, nous regardons l'impact de la largeur du bus d'entrée/sortie sur la fréquence maximale (Section 3.2.3). Finalement, nous évaluons l'impact de différents modules PISA sur la consommation de ressources ainsi que la fréquence maximale (Section 3.2.2).

3.2.1 Méthodologie d'évaluation du FPGA comme commutateur

Un pipeline PISA s'implémente sur FPGA avec une structure par flux et avec le concept de propagation par l'avant. Il en résulte que le débit d'un pipeline PISA est égal à la largeur du bus d'entrée-sortie multiplié par la fréquence. Pour augmenter le débit, on peut donc soit augmenter la fréquence ou bien la largeur du bus. Dans cette section, nous présentons la méthode et les programmes utilisés afin de déterminer les limites de l'implémentation de l'architecture PISA sur FPGA.

Afin d'évaluer l'implémentation de l'architecture PISA sur FPGA, plusieurs programmes P4 sont compilés et implémentés sur FPGA. La compilation des programmes P4 est faite en utilisant le compilateur Xilinx SDNet 2017.4. La synthèse et l'implémentation est faite avec Vivado 2018.2 en ciblant un FPGA Xilinx Ultrascale+ *xcvu9p* avec une contrainte d'horloge pour le pipeline PISA de 500 MHz.

Nous implémentons six programmes P4 différents afin de tester différents aspects de l'architecture PISA. Le tableau 3.1 présente les caractéristiques des différents programmes que nous implémentons et l'élément de PISA que nous testons. Le programme T0 est le cas de base, il s'agit d'un pipeline PISA qui parse un paquet puis le déparse sans faire aucune modification. Ce programme parse 3 en-têtes totalisant 74 octets. Le programmae T1 contient 8 en-têtes de plus que T0 pour un total de 114 octets d'en-têtes à parser et déparser, il est utilisé pour évaluer le parseur et le déparseur. L'implémentation des actions est évaluée avec les programmes T2 et T3. Le programme T2 calcule une somme de contrôle pour l'en-tête IPv4 qui nous permet de tester des opérations logiques et artithmétiques. Le programme T3 permet d'évaluer l'impact des conditions. Les programmes T4 et T5 sont utilisés pour évaluer l'implémentation des tables de comparaison exacte ainsi que des TCAM. Dans les deux cas, les tables ont des clés de 128 bits pour accéder à des données de 9 bits. La table de correspondance exacte a 64 k entrés et la TCAM a 4 k entrées.

Tableau 3.1 Programmes pour l'évaluation du FPGA pour l'implémentation de PISA

Programme	Description	Bloc PISA évalué
T0	3 en-têtes, 74 octets d'en-têtes	
T1	T0 + 8 en-têtes, 114 octets d'en-têtes	Parseur et Déparseur
T2	T1 + somme de contrôle IPv4	Actions
T3	T1 + une chaîne de 4 conditions	Actions
T4	T2 + comparaison exacte 64 k entrée x 128 bits	Table de comparaisons
T5	T2 + TCAM 4k entrée de 128 bits	Table de comparaisons

3.2.2 Évaluation de l'implémentation des différents blocs de PISA

L'augmentation de la fréquence peut permettre d'accroître le débit d'un pipeline PISA. Cela peut être fait en augmentant la profondeur du pipeline. En théorie, un certain nombre de modules PISA peuvent être pipelinés afin d'accroître la fréquence. Cependant, il est rare d'avoir des implémentations sur FPGA atteignant des fréquences de plus de 500 MHz [96]. Dans cette section, nous regardons comment évolue la consommation de ressources ainsi que la fréquence de différents modules de PISA. Pour cela, nous implémentons les six programmes P4 présentés dans la section 3.2.1. Ces programmes sont implémentés avec un bus d'entrée-sortie de 2048 bits.

Les résultats d'implémentation des six programmes sont présentés dans la figure 3.1. La ligne pointillée indique la période d'horloge permettant d'atteindre un débit de 600 Gb/s. On constate, en comparant à T0, que les implémentations de T1, T2 et T3 atteignent des périodes d'horloge similaires. Donc, augmenter la taille du parseur et du déparseur ainsi que les actions à effectuer n'affectent pas significativement le débit. On remarque par les résultats de T4 et T5 que les tables de comparaison augmentent de plus de 60 % la période d'horloge. Cette augmentation de la fréquence provient de la distribution des BRAM dans le FPGA. En effet, les BRAM sont organisées en colonnes et pour implémenter des mémoires larges, plusieurs colonnes sont utilisées augmentant ainsi le délai de routage.

On constate que la consommation de ressources est plus importante pour T1 que pour T0, avec 80% des ressources utilisées par le déparseur. L'implémentation de T2 consomme environ 10 k FF et 7 k LUT de plus que T1, T3 requiert environ 7 k et 3 k FF de plus T1, T2 et T3 consomment autant de BRAM que T1. Cela confirme l'efficacité de l'implémentation des actions. L'utilisation de BRAM augmente pour T4 et T5, car on implémente des tables de comparaisons, mais on constate une l'augmentation de l'utilisation de LUT et FF pour T5. Ces résultats suggèrent que les tables de comparaison exacte s'implémentent bien sur FPGA et que les TCAM ne s'implémentent pas efficacement.

Finalement, lorsqu'on regarde les résultats d'implémentation de T1 et T0, on constate que le déparseur représente plus de la moitié des ressources consommées. Ceci indique qu'une architecture de déparseur est nécessaire pour implémenter plus efficacement des programmes P4 sur FPGA. Le fait que le déparseur consomme beaucoup plus de ressources que la solution proposée par [17], peut s'expliquer par la version de P4 utilisée. En effet, en $P4_{16}$, le graphe de déparsage n'est pas le même que le graphe de parsage comme c'est le cas avec $P4_{14}$ (section 2.2.2). De plus, les compilateurs ne proposent pas de réduction du graphe de déparsage. Il est donc possible que la consommation importante de ressources découle du fait que le déparseur émette plus de combinaisons d'en-têtes.

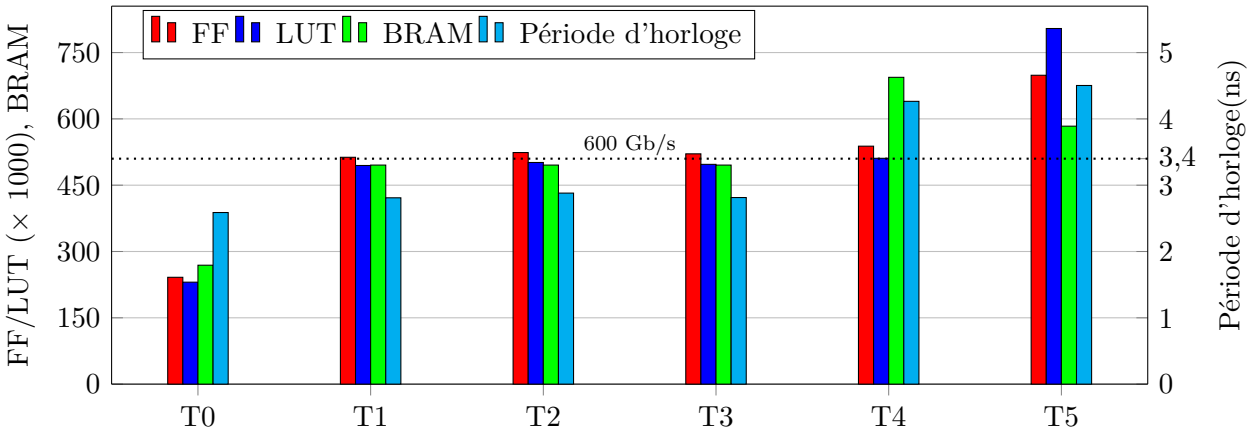


Figure 3.1 Résultats d'implémentation des programmes du tableau 3.1 avec un bus d'entrée-sortie de 2048 bits

3.2.3 Impact de la largeur du bus sur la fréquence d'opération

Une solution pour accroître le débit d'un pipeline PISA est d'augmenter la largeur du bus d'entrée-sortie. En effet, le débit maximal est égal à la fréquence multipliée par la largeur du bus. Toutefois, en augmentant la largeur du bus d'entrée-sortie, il faut synchroniser et traiter plus de bits. Cela résulte en une utilisation plus importante de ressources et en une dégradation de la fréquence d'horloge maximale. De plus, l'augmentation de la taille des bus augmente le risque de congestion lors du routage, ce qui peut dégrader encore plus la fréquence maximale d'horloge. Dans cette section, nous évaluons l'impact de l'augmentation de la largeur du bus d'entrée-sortie sur la fréquence d'horloge pouvant être atteinte par un pipeline PISA.

Pour évaluer l'impact de la largeur du bus sur la fréquence, nous implémentons le programme T0 (Tableau 3.1) pour des bus de 64, 128, 256, 512, 1024, 1280 et 2048 bits de large. Les

résultats d'implémentation sont montrés dans la figure 3.2. On constate que l'utilisation de ressources est proportionnelle à la largeur du bus. La fréquence d'horloge est relativement stable pour des bus de 64 à 1024 bits autour de $2ns$ (500 MHz), mais se détériore pour des bus de 1280 bits. Cette dégradation provient d'une grande sortance (fan-out) des liens ainsi que d'une congestion pour le routage. On obtient ainsi un débit maximal de 786 Gb/s dans le cas d'un bus de 2048 bits.

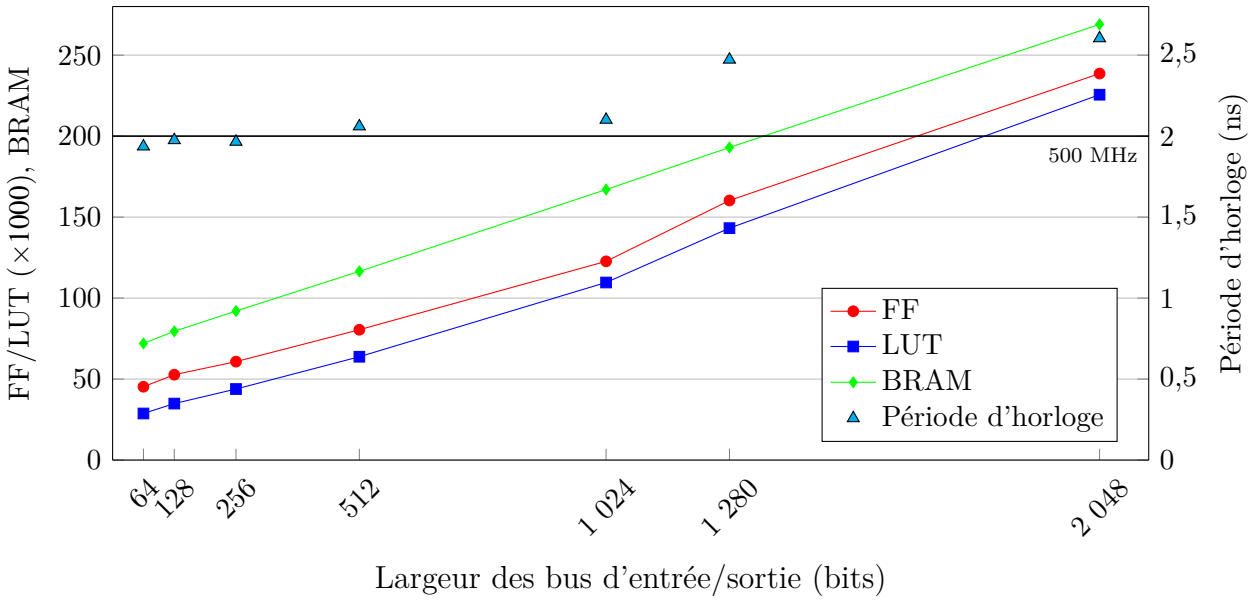


Figure 3.2 Résultats d'implémentation de T0 pour différentes largeurs du bus d'entrée-sortie

3.3 Les solutions pour améliorer l'implémentation de PISA sur FPGA

L'efficacité d'implémentation de PISA est actuellement limitée par l'architecture du FPGA, notamment pour les comparaisons. Aussi, on constate que pour des bus plus larges permettant d'atteindre de plus grands débits, la fréquence d'horloge se dégrade. Également, nous avons vu que l'implémentation des déparseurs consomme beaucoup de ressources.

Dans cette section, nous proposons des solutions pour améliorer l'implémentation de PISA sur les FPGA. Dans un premier temps, nous proposons des modifications à l'architecture du FPGA (Section 3.3.1). Puis nous proposons des axes afin d'améliorer l'implémentation du déparseur sur FPGA (Section 3.3.2).

3.3.1 Intégrer de nouveaux blocs aux FPGA

L'intégration de nouveaux blocs à la microarchitecture des FPGA doit être faite avec prudence. En effet, les FPGA sont utilisés dans un large éventail d'applications tel que le traitement du signal, le chiffrement ou encore l'émulation des systèmes logiques [97]. Nous proposons ici trois modifications pour améliorer l'implémentation de PISA sur FPGA. La première modification permet un meilleur support des tables de comparaison par l'intégration de mémoires TCAM. Les deux autres modifications ont pour objectif de limiter la congestion en modifiant la structure de routage du FPGA.

Des TCAM dédiées

L'utilisation de TCAM dédiées aiderait à améliorer l'implémentation des comparaisons ternaires. Des TCAM ont déjà été implémentées par le passé dans des FPGA Lattice [98], mais de telle TCAM ne sont plus dans les FPGA actuels. Nous évaluons ici le bénéfice de l'intégration de TCAM en calculant le nombre de transistors requis pour l'implémentation d'une TCAM de profondeur 128 et d'une largeur de 48 bits. Ces TCAM correspondent aux TCAM qui étaient implémentées dans les FPGA Lattice [98].

Une cellule TCAM de un bit requiert 16 transistors [99] et une cellule SRAM de 1 bit requiert 6 transistors. Comme l'efficacité d'une TCAM implémentée avec les éléments du FPGA a un surcoût mémoire de $8,5\times$ au minimum (Section 3.1.2), il faut $8,5 \times 6 = 51$ transistors par bit de TCAM implémentée sur FPGA. Ainsi, en implémentant des TCAM sur FPGA, on peut réduire de plus de $3\times$ ($\frac{51}{16} \approx 3.19$) le nombre de transistors utilisés lors de l'implémentation d'une cellule TCAM sur FPGA.

Outre les applications réseaux, les TCAM peuvent être utilisées pour diverses applications, telles que la compression et le chiffrement des données [98]. Les TCAM peuvent aussi être utilisées pour des algorithmes de recherche du plus proche voisin [100]. Également, l'utilisation de TCAM en tant que CAM pourrait permettre d'améliorer l'implémentation de l'exécution hors d'ordre pour des processeurs implémentés sur FPGA [82].

De nouvelles structures de routage

La structure de routage des FPGA est également un élément limitant. En effet, pour augmenter le débit maximal, il faut utiliser des bus plus larges. Cependant, des bus trop larges rendent le routage complexe ce qui résulte en une dégradation de la fréquence d'horloge. Nous proposons deux axes complémentaires pour aider au routage : l'intégration de réseaux sur puce — *Networks on Chip* (NoC) et l'ajout de bus aux ressources de routage.

L'intégration de NoC consiste à intégrer une grille au FPGA permettant de connecter différentes portions du FPGA entre elles [101]. Comme le NoC est implémenté en dur, il peut fonctionner à haute fréquence ce qui permet des interconnexions rapides entre les modules du FPGA [102]. Également, l'utilisation d'un NoC réduit la complexité du routage en réduisant le champ de routage. Le NoC peut être utilisé pour transmettre les données entre les différents blocs, mais aussi pour envoyer directement les données vers les interfaces d'entrée-sortie. En effet, en ayant un NoC, les interfaces d'entrée sortie peuvent être directement connectés au NoC, ce qui permet par la suite de limiter les contraintes pour le placement de modules de traitement [103].

Notre seconde proposition pour améliorer le routage sur le FPGA est d'intégrer des bus aux ressources de routage. L'idée est d'intégrer des bus qui sont considérés comme une seule instance ainsi au lieu de router n fils de 1 bit, on route un bus de n bits. Cette approche réduit le coût de l'interconnexion car un seul mot de configuration route plusieurs fils [104]. Ceci bénéficierait à une variété d'applications, car beaucoup traitent des données représentées sur plusieurs bits.

Gaide et al. proposent une nouvelle architecture de FPGA contenant un NoC [103]. Le NoC proposé a une topologie en colonne avec un bus en haut et en bas des colonnes, tel que montré dans la figure 3.3. L'utilisation du NoC permet de placer les éléments de traitement indépendamment de la position des interfaces sur la puce car il est possible de déplacer les données efficacement sans avoir des problèmes de timing. Le débit du NoC proposé pourrait toutefois être limitant, les auteurs annoncent un débit permettant de supporter la bande passante de la mémoire DDR4 soit d'environ 200 Gb/s [105], ce qui est bien inférieur au débit total des interfaces réseau des FPGA de plus de 1 Tb/s (Section 3.2). Les auteurs proposent aussi d'améliorer le routage en augmentant la taille et des blocs de logique configurable — *Configurable Logic Bloc* (CLB) et leur capacité de routage interne [103]. Également, Gaide et al. intègrent des structures de routage augmentant l'interconnexion entre les CLB. La nouvelle structure proposée semble améliorer le routage, notamment en augmentant le routage interne des CLB, le tissu de routage entre les blocs est ainsi moins sollicité.

3.3.2 Revoir l'implémentation du déparseur

Le déparseur est le module responsable de recomposer le paquet après le traitement des en-êtes dans le pipeline PISA, ce qui le rend essentiel à toute application réseau. Il est considéré que le déparseur est l'inverse du parseur [17]. Il a été montré que le parseur consomme peu de ressources sur un FPGA [28], l'implémentation du déparseur devrait donc elle aussi utiliser peu de ressources du FPGA. Cependant, nous avons constaté que le déparseur consomme

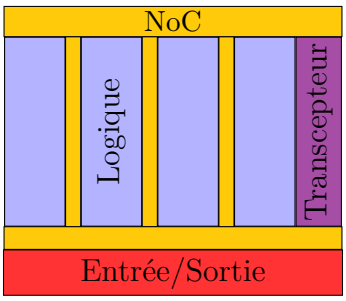


Figure 3.3 Présentation de la nouvelle architecture de FPGA proposé par [103]

une part significative des ressources du FPGA lors de son implémentation. Il y a donc des possibilités permettant de réduire le coût d'implémentation du déparseur sur FPGA. Pour cela, nous pensons que deux éléments sont à travailler : proposer une architecture de déparseur mieux adaptée aux FPGA et transformer le graphe de déparsage pour réduire le coût d'implémentation de cette architecture.

Le premier élément qui permettrait de diminuer les ressources utilisées par le déparseur sur le FPGA est de proposer une nouvelle architecture de déparseur. La nouvelle nouvelle architecture doit prendre avantage du tissu d'interconnexions du FPGA ainsi que de sa configurabilité. Le rôle du déparseur est de déplacer des données afin de les organiser sur le bus de sortie. Cette organisation se fait par un mouvement de données spatial, mais aussi temporel. Il faut donc une machine à états ainsi que des structures pour déplacer les données. Faire des décaleurs de bits peut s'avérer très coûteux, principalement si l'on veut des décalages de taille variable. Nous avons toutefois vu que les multiplexeurs peuvent s'implémenter assez efficacement sur les ressources du FPGA. Nous pensons donc qu'une architecture de déparseur doit se structurer autour de connexions fixes vers des multiplexeurs. Cela permet d'utiliser au mieux le tissu de routage du FPGA. Également, comme pour le parseur, il faut une machine à états afin de contrôler les multiplexeurs, cette machine à états dérive du graphe de déparsage.

Le second élement qui permettrait d'utiliser plus efficacement les ressources du FPGA par le déparseur est la transformation du graphe de déparsage pour qu'il n'émette que des combinaisons d'en-têtes possibles. Dans le travail de Benáček et al., le déparseur est construit à partir du graphe de parsage [17]. Leur déparseur consomme moins de ressources que le déparseur de Xilinx qui ne dérive pas du graphe de parsage. Ainsi, des modifications au compilateur P4 seraient requises pour pouvoir transformer le graphe de déparsage. Cela permettrait d'implémenter une machine à états plus petite, et de réduire le nombre de décalages à réaliser, ce qui permettrait une diminution des ressources consommées.

Ces deux propositions pour réduire le coût de l'implémentation du déparseur sur les FPGA sont explorées dans cette thèse. L'architecture est vue en détail dans le chapitre 4. La transformation du graphe de déparsage est explorée dans le chapitre 5.

CHAPITRE 4 UNE MICROARCHITECTURE DE DÉPARSEUR POUR FPGA

Dans ce chapitre, nous présentons deux contributions de cette thèse. Nous proposons la transformation du graphe de déparsage d'un programme P4 afin de pouvoir mieux en extraire le parallélisme. Nous proposons également une architecture FPGA générée à partir du graphe de déparsage transformé. Cette architecture s'intègre dans le pipeline d'une architecture de commutateur indépendante des protocoles — *Protocol Independent Switch Architecture* (PISA) et tire avantage de la capacité de reconfiguration et du tissu d'interconnexions des FPGA. Le résultat de ce travail a été publié à la conférence FPGA'21 [106].

Le déparseur est responsable de recomposer le paquet après le traitement de ses en-têtes dans le commutateur. Il s'agit de concaténer les en-têtes modifiés et la charge utile du paquet afin de générer le paquet de sortie. L'ordre dans lequel les en-têtes sont insérés est défini par le graphe de déparsage qui est obtenu par la compilation du code P4.

Dans un premier temps, nous détaillons la transformation du graphe de déparsage de P4 afin d'accroître le parallélisme de traitement dans le déparseur (Section 4.1). Par la suite, nous présentons les principaux blocs de la microarchitecture de déparsage (Section 4.2). Nous pourrons ainsi présenter la transformation du graphe de déparsage vers l'architecture (Section 4.3). Finalement, les résultats d'implémentation de l'architecture générée ainsi qu'une comparaison avec d'autres architectures de déparseurs sont présentés (Section 4.4).

4.1 Le graphe de déparsage

Le graphe de déparsage permet d'indiquer l'ordre dans lequel les en-têtes sont émis. Dans le cas des programmes P4, le graphe de déparsage est un GDA. Dans cette section, nous proposons d'effectuer une fermeture transitive du GDA de déparsage afin de pouvoir extraire le parallélisme dans les transitions d'états. Dans un premier temps, nous présentons la représentation d'un déparseur P4 (Section 4.1.1). Par la suite nous présentons la fermeture transitive du GDA de déparsage proposée (Section 4.1.2).

4.1.1 La représentation P4 d'un déparseur

Le déparseur se décrit en P4 en indiquant l'ordre d'émission des en-têtes dans une succession d'appels à la fonction `emit` [107]. Le Listage 4.1 montre un exemple de déparseur écrit en P4 pour l'émission des en-têtes `ethernet`, `ipv4` et `tcp`. La fonction `emit`, comme définie dans

la spécification de $P4_{16}$, doit vérifier pour chacun des en-têtes si son champ de validité est vrai. Si le champ de validité est vrai alors l'en-tête est inséré dans le paquet, sinon l'en-tête suivant est émis. Le code présenté dans le Listage 4.1 peut être converti dans le graphe de la Figure 4.1. Dans ce GDA, les sommets exécutent la fonction `emit` sur l'en-tête à l'exception des sommets «start» et «end», qui sont les sommets de début et fin d'émission d'en-tête.

Le graphe présenté dans la Figure 4.1 peut être transformé en découpant chaque sommet en deux sommets différents : un qui vérifie la validité de l'en-tête, l'autre qui insère l'en-tête. La vérification de la validité de l'en-tête s'effectue en utilisant la fonction P4 `isValid`. Si l'en-tête est valide on insère l'en-tête sinon on évalue la validité de l'en-tête suivant. Cette transformation génère le GDA présenté dans la Figure 4.2.

```
control deparser(packet_out Pkt_out, in header_s PHV) {
    apply {
        Pkt_out.emit(PHV.ethernet);
        Pkt_out.emit(PHV.ipv4);
        Pkt_out.emit(PHV.tcp);
    } }
```

Listage 4.1 Exemple d'un code P4 de déparseur pour les en-têtes `ethernet`, `ipv4` et `tcp`

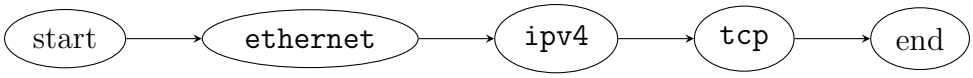


Figure 4.1 Graphe de déparsage résultant du Listage 4.1

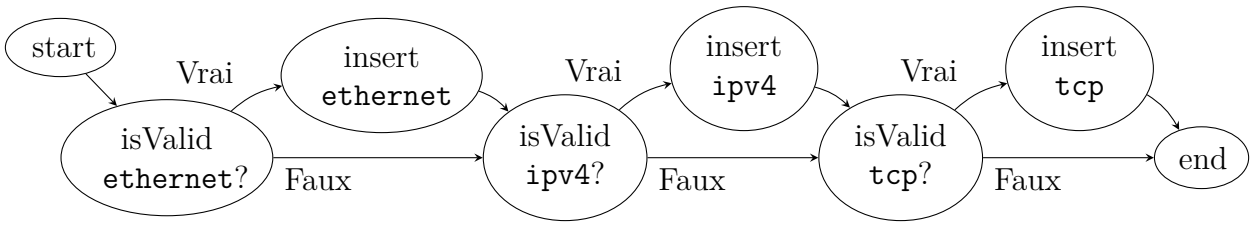


Figure 4.2 Graphe de déparsage après développement du GDA de la Figure 4.1

4.1.2 Transformer le graphe de déparsage

La représentation du GDA de déparsage présentée précédemment implique une forte dépendance dans le graphe de contrôle. En effet, les en-têtes doivent être insérés dans un ordre précis déterminé par le programmeur. Lorsque l'on analyse la figure 4.2, on remarque que

les validités des en-têtes sont évaluées séquentiellement et les résultats de cette évaluation déterminent ce que l'on fait ensuite. Ainsi, si l'on émet un paquet pour lequel seul l'en-tête `ethernet` est valide, il faut quand même évaluer les en-têtes `ipv4` et `tcp`. Nous proposons donc une transformation du GDA de déparsage afin de définir les conditions exactes permettant d'accéder à un sommet.

Pour cela, nous utilisons le concept de fermeture transitive qui consiste à insérer une arête entre les sommets du graphe ayant un chemin dans le graphe d'origine [108]. Lors de la fermeture transitive du graphe, nous ajoutons à chaque nouvelle arête une condition à respecter pour suivre cette arête. Ainsi, à partir de chacun des sommets du GDA, il est possible de déterminer l'arête à suivre en évaluant la condition de toutes les arêtes. L'algorithme 4.1 est utilisé pour faire la fermeture transitive que nous utilisons. Un exemple de résultat de transformation est présenté dans la figure 4.3.

L'algorithme 4.1 crée un nouveau GDA transitivement fermé avec les conditions associées à chacune des arêtes. Le GDA transitivement fermé est construit en commençant par le dernier sommet dans le GDA d'origine. On connecte chaque nouveau sommet inséré aux sommets suivants possibles. On construit la liste de sommets suivants comme une pile. Le dernier sommet ajouté est le premier connecté. La condition est construite afin de respecter l'ordre de parcours du GDA de déparsage initial.

Algorithme 4.1 : Fermeture transitive avec condition d'un Graphe Dirigé Acyclique de déparsage

Entrée : G_o : Graphe de déparsage
Sortie : G_c : Graph de déparsage transitivement fermé

```
S ← liste(G_ori) ;                          // Liste des sommets en ordre inverse
E ← [] ;                                    // Liste pour les états suivants
G_c ← newDAG() ;                            // Nouveau GDA, vide au départ
Pour Chaque n in S[: −1] Faire              // Itération sur les sommets en ordre
 inverse
    cond ← "";
    G_c.AjoutNoeud(n);
    Pour Chaque es in E Faire               // On itère sur les sommets suivant
        cond ← es[1] + cond;
        G_c.AjoutArête(n, es[0], cond) ;    // arête de n vers es avec condition
        cond ← "AND not" + cond;
    FinPourCh
    // On insère l'état suivant avec une condition associée;
    Si n is end Alors E.insertFront((n, true)) Sinon E.insertFront((n, n + .isValid));
FinPourCh
return G_c
```

L'application de l'algorithme 4.1 au graphe de déparsage de la figure 4.1 génère le graphe de la figure 4.3. Dans ce graphe chacun des sommets effectue l'insertion d'un en-tête. Les étiquettes des arêtes indiquent la condition à respecter pour pouvoir emprunter l'arête correspondante. Ainsi, pour pouvoir aller du sommet «start» au sommet «end», il faut que le champ de validité des en-têtes `ethernet`, `ipv4` et `tcp` soit invalide. Si on est au sommet «start» et que l'en-tête « `ethernet` » est valide, alors le prochain noeud sera `ethernet`.

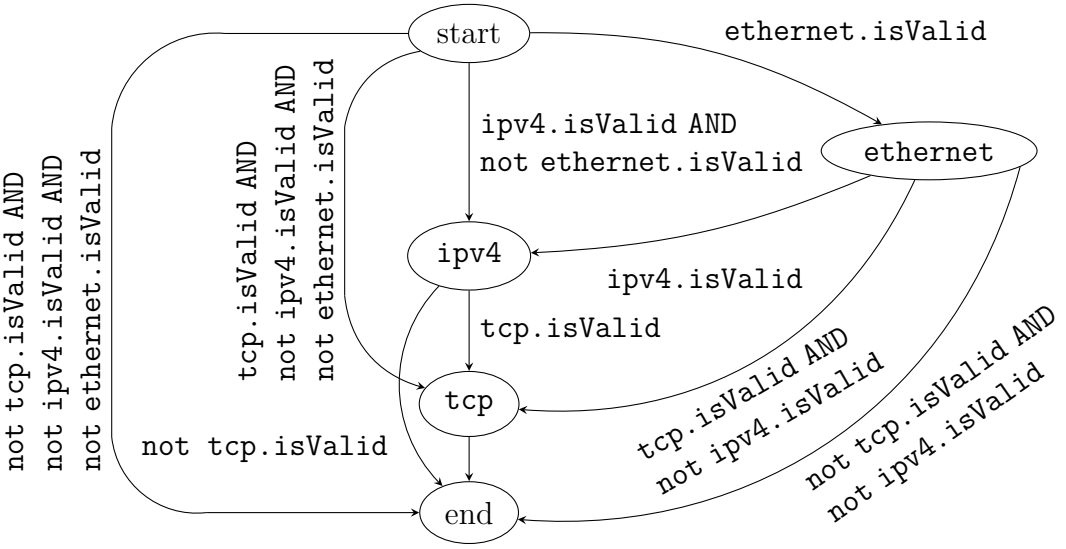


Figure 4.3 Fermeture transitive du GDA de déparsage de la figure 4.1 en utilisant l'algorithme 4.1

4.2 Les composants de l'architecture

L'architecture proposée adopte une structure par flux avec le concept de propagation par l'avant, il n'y a donc pas de boucle de rétroaction. Ceci est conforme à la structure de PISA [7]. Par conséquent, le débit de sortie du déparseur est garanti.

Le déparseur recompose le paquet de sortie en associant les nouveaux en-têtes du paquet avec la charge utile du paquet en entrée. Il prend donc en entrée le PHV ainsi que la charge utile du paquet original et les associe afin de générer un nouveau paquet en sortie. Comme nous adoptons une structure de flux, le paquet en sortie se compose de la succession des vecteurs de bits émis par le déparseur.

Afin de maximiser la bande passante, l'architecture doit émettre un vecteur de bits à chaque cycle d'horloge. Il faut donc déterminer à chaque cycle la valeur prise par chacun des bits du vecteur de sortie. Par conséquent, l'architecture de déparseur sélectionne le bit d'entrée à émettre pour chacun des bits de sortie à un cycle donné. Ainsi, nous proposons l'architecture

de la figure 4.4 qui connecte chaque bit de sortie à différents bits du vecteur d'en-tête et de la charge utile du paquet. La connexion se fait à l'aide de trois blocs principaux : le sélecteur de bit d'en-tête (Section 4.2.1), le sélecteur de bits de la charge utile (Section 4.2.2), et le sélecteur de bit de sortie (Section 4.2.3).

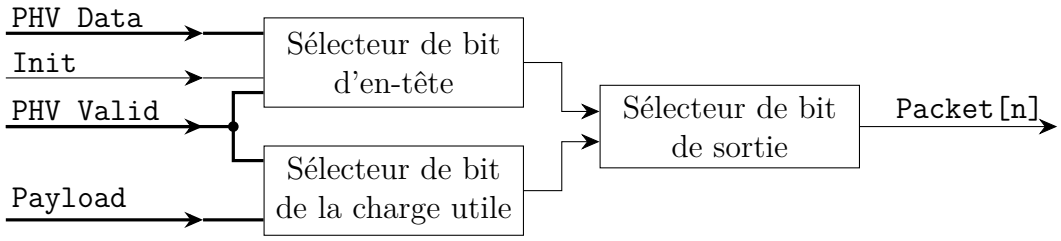


Figure 4.4 Structure d'un bloc de déparseur sur FPGA pour reconstituer le bit `n` du paquet.

4.2.1 Le sélecteur de bit d'en-tête

Un des éléments principaux de l'architecture de déparsage est le sélecteur de bit d'en-tête. Le rôle du sélecteur de bit d'en-tête est d'émettre les bons bits d'en-tête pour chacun des bits du vecteur de sortie. La figure 4.5 présente la vue d'ensemble du sélecteur de bit d'en-tête. Le sélecteur de bit d'en-tête prend en entrée un signal de départ `Init` et le vecteur d'en-tête PHV composé de deux parties : les données des en-têtes `PHV data` et les bits de validité des en-têtes `PHV valid`. Le sélecteur génère en sortie un bit de donnée `Header data`, un bit de validité `Header valid`, indiquant si le bit est valide, et un bit indiquant si l'on émet le dernier bit d'en-tête `Header last`.

Le bit `Header Data` est choisi du vecteur `PHV Data` à l'aide d'un multiplexeur contrôlé par la machine à états. La machine à états prend en entrée le vecteur `PHV valid` et le signal `Init` et détermine la valeur des signaux `Header valid` et `Header last`. La génération de la machine à états est détaillée dans la section 4.3. Ce module sera par la suite répliqué pour chacun des bits de sortie du déparseur. Comme le module est répliqué, il est possible de réduire la taille du multiplexeur. En effet, les bits de l'en-tête sont toujours émis dans le même ordre. Ainsi, si nous considérons un bus de sortie de 64 bits et deux en-têtes de 64 bits, alors chacun des bits de chacun des en-têtes se connecte uniquement à un seul bit de sortie. Le multiplexeur de chacun des sélecteurs de bit d'en-tête est donc un multiplexeur 2-1.

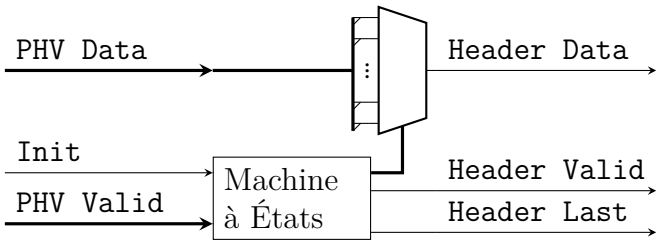


Figure 4.5 Le sélecteur de bit d'en-tête pour un bit de sortie

4.2.2 Le sélecteur de bit de la charge utile du paquet

Le bloc de sélection de bit de la charge utile de paquet aligne la charge utile du paquet sur le bus de sortie. La figure 4.6 montre l'architecture proposée pour la sélection d'un bit de charge utile. Ce bloc de sélection prend en entrée le bus de charge utile composé de deux signaux principaux : `Payload Data In` et `Payload Keep In` contenant respectivement des bits de donnée et leur validité. Il prend également en entrée le vecteur de validité des en-têtes : `PHV valid`. Le sélecteur de bit de la charge utile sort un bit de donnée `Payload Data Out` avec un bit de validité associé `Payload Keep`.

Les vecteurs de bit `Payload Data In` et `Payload Keep In` sont connectés aux multiplexeurs 1 et 3 respectivement. Ces multiplexeurs sont contrôlés par un signal en sortie du bloc «Décodeur» qui, en fonction des en-têtes valides, détermine le bit à sélectionner. En sortie des multiplexeurs 1 et 3, une bascule D est insérée, cette bascule permet de décaler la sortie d'un cycle d'horloge. Finalement `Payload Data Out` et `Payload Keep Out` sont déterminés respectivement par les multiplexeurs 2 et 4 qui sont également contrôlés par le bloc «Décodeur». Les multiplexeurs 2 et 4 déterminent si le bit sélectionné doit être décalé d'un cycle ou non. Le bloc «Décodeur» est configuré de manière à associer pour chacune des valeurs possible de `PHV valid` une valeur pour le contrôle des multiplexeurs. La détermination des valeurs de configuration du décodeur est expliquée dans la section 4.3.

4.2.3 Le sélecteur de bit de sortie

Le dernier bloc de l'architecture est le sélecteur de bit de sortie présenté à la figure 4.7. Ce bloc sélectionne entre les bits d'en-têtes et les bits de charge utile pour la sortie du paquet. Le sélecteur prend en entrée les sorties du bloc de sélecteur d'en-tête et du sélecteur de bit de charge utile ainsi que les signaux `Payload Last` et `Has Payload`. Il sort les signaux : de la donnée du paquet `Packet Data`, de validité de la donnée `Packet Keep` et de fin de paquet `Packet Last`. La sélection du signal de donnée se fait en fonction de la valeur du signal `Header Valid`. Quand le signal est à 0, on émet les bits de donnée. La sortie de donnée est

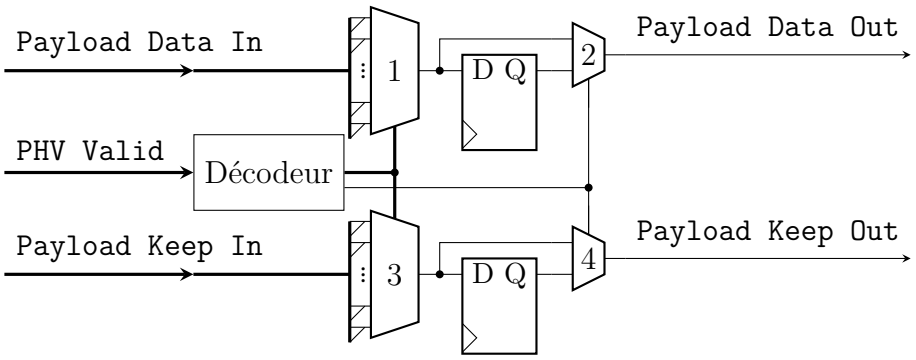


Figure 4.6 Le sélecteur de bit de la charge utile du paquet pour un bit de sortie

valide si `Header Valid` ou `Payload Keep Out` sont valides. Si le paquet n'a pas de charge utile, `Has Payload` = 0, alors la dernière émission d'en-tête indique la fin du paquet, sinon la dernière trame de charge utile est utilisée comme fin de paquet. Ce module est dupliqué pour chaque bit de sortie du paquet.

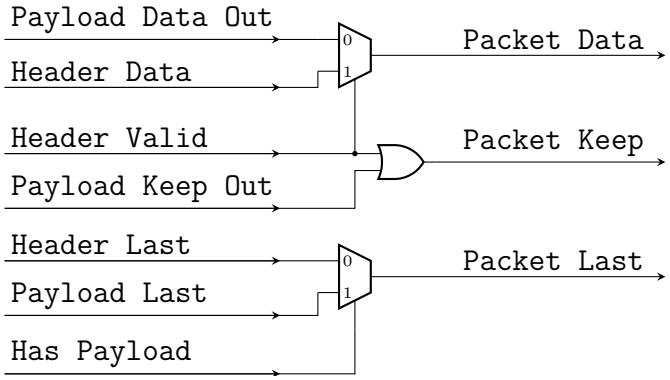


Figure 4.7 Le sélecteur de bit de sortie.

4.3 Générer l'architecture

La section 4.1 présente la transformation du graphe de déparsage. Cette transformation permet de déterminer le prochain en-tête à émettre en fonction de l'en-tête en train d'être émis et des en-têtes valides. Cette section présente la génération le déparseur à partir du GDA transitivement fermé. Tout d'abord, nous présentons le découpage que nous effectuons sur le GDA de déparsage. Puis nous expliquons comment les blocs de connexions du PHV et du sélecteur de bit de la charge utile sont générés.

4.3.1 La décomposition du graphe de déparsage

L'architecture de déparsage proposée se structure autour de blocs générés pour chacun des bits de sortie du bus de données. Ainsi, pour chacun de ces bits il faut déterminer les connexions possibles au PHV et à la charge utile ainsi que l'ordre dans lequel les bits doivent être insérés sur le bus de sortie. Pour cela, nous décomposons le GDA de déparsage transitivement fermé afin de généré des sous GDA pour chaque bit de sortie sur le bus de donnée.

L'algorithme 4.2 présente le générateur de sous-graphes. L'algorithme prend deux paramètres : le graphe de déparsage à séparer ainsi que le nombre de sous-graphes à générer. Il génère en sortie les sous-graphes ; chacun des sous-graphes correspond à un bit de sortie du déparseur. Le graphe de déparsage fourni en entrée doit avoir les conditions de transition sur les arêtes. Pour générer les sous-graphes, on parcourt chacun des chemins entre les sommets « start » et « end » du GDA de déparsage.

Lors du parcours d'un chemin p du graphe de déparsage, les différentes arêtes A du chemin sont évaluées. Une arête est un objet dont le premier élément est le sommet d'origine et le second le sommet de destination. L'arête peut également avoir une condition qui lui est associée. Avant le parcours du chemin, nous mettons l'état actuel, « prev_node » de chacun des sous-graphes comme étant l'état de départ, « start », du GDA de déparsage. Pendant le parcours du chemin, nous évaluons l'en-tête h que le sommet de destination insère, puis pour chacun des bits de cet en-tête un sommet $Noeud$ est créé. À partir de ce nouveau sommet, une arête $Arête$ reliant l'ancien sommet du sous-graphe avec le nouveau sommet $noeud$ est créée. Si le bit d'en-tête émis est plus petit que le nombre de sous-graphes alors on ajoute la condition de A à l'arrête, sinon aucune condition n'est nécessaire. Finalement, l'arête est insérée dans le sous-graphe, et le sommet précédent, prev_node, du sous-graphe est mis à jour puis le sous-graphe suivant est sélectionné pour évaluer le prochain bit d'en-tête.

La figure 4.8 montre le sous-graphe pour le bit numéro 2 d'un bus de sortie de 256 bits obtenue lors de la séparation du graphe présenté dans la figure 4.3. Pour une meilleure lisibilité, seuls les noms des en-têtes à tester sur les arêtes ont été mis. Lors de la construction de ce sous-graphe, certaines conditions ont été enlevées tel que la condition entre `ipv4[146]` et «end». Cette condition est retirée, car lors de l'émission du bit 146 de `ipv4`, il n'y aura plus de bit d'en-tête à insérer sur le paquet de sortie. Dans le graphe d'origine, après l'émission de l'en-tête `ipv4`, il est possible soit d'aller à l'émission de `tcp` ou bien d'aller à «end». Donc lors de l'émission `ipv4[146]`, il reste à émettre au maximum, le reste de l'en-tête `ipv4` et l'en-tête `tcp`, qui ont une largeur de 160 bits chacune. Le reste de l'en-tête `ipv4` est donc de $160 - 146 = 14$ bits, en y ajoutant la largeur de `tcp`, le déparseur doit encore émettre $14 + 160 = 174$ bits d'en-tête. Comme nous avons un bus de sortie de 256 bits, les 174 bits

restants seront insérés sur les autres bits du bus. Après l'insertion du bit 146 de `ipv4`, il n'y aura plus de bits d'en-tête à émettre sur le bit 2 du bus de sortie.

Algorithme 4.2 : Génération des sous-graphes de déparsage

Entrée : G_d : le graphe de déparsage
Entrée : w : nombre de sous-graphes à générer
Sortie : Sg : liste de w sous-graphe
$Sg \leftarrow$ liste de w GDA vide;
Pour Chaque p *dans* $G_d.path()$ **Faire**
    $Sgi \leftarrow 0$; // Numéro du sous-graphe
    prev_node $\leftarrow$ [] ; // Sommet précédent pour chaque sous-graphe
    // Insertion du premier noeud de p pour chaque sous-graphe
    Pour $i \leftarrow 0$ **to** w **Faire** prev_node.append($p.Nodes[0]$);
    Pour Chaque A *dans* $p.Edges$ **Faire**
        $h \leftarrow getHeader(A[1])$; // En-tête du noeud de destination
        Pour $i \leftarrow 0$ **to** *size(h)* **Faire** // Chaque bit de l'en-tête
            // Construction des arêtes du sous-graphe
            $Noeud \leftarrow$ new extractNode(h, i) ; // Noeud pour extraire le bit i de h
            $Arête \leftarrow$ new edge(prev_node[Sgi], $Noeuds$) ; // Arête prev $\rightarrow$ Noeud
            // Début de l'en-tête, on insère la condition de A
            Si $i < w$ **Alors** $Arête$.Ajout_cond(A.cond());
            $Sg[Sgi]$.insert($Arête$) ; // Ajout $Arête$ au sous-graphe Sgi
            prev_node[Sgi] $\leftarrow$ $Noeud$; // Mise à jours des sommets précédents
            $Sgi \leftarrow (Sgi + 1)$ mod w ; // Sous-graphe suivant
        FinPour
    FinPourCh
FinPourCh
return Sg

4.3.2 Génération du sélecteur de bit d'en-tête

Chacun des sous-graphes générés avec l'algorithme 4.2 permet de construire un sélecteur de bit d'en-tête. Le bloc de PHV contient un multiplexeur et une machine à états. Le multiplexeur est généré en fonction des bits à connecter qui découle des informations fournies dans les noeuds du sous-graphe. Comme chaque sommet du sous-graphe est associé à un unique bit d'en-tête à émettre, le nombre d'entrées du multiplexeur est égal au nombre de sommets du sous-graphe en excluant les états « start » et « end ». Lors de la connexion des bits du multiplexeur, une table de correspondance est construite entre la position des connexions sur le multiplexeur et le sommet du sous-graphe correspondant. Cette table de correspondance est ensuite utilisée pour générer la machine à états.

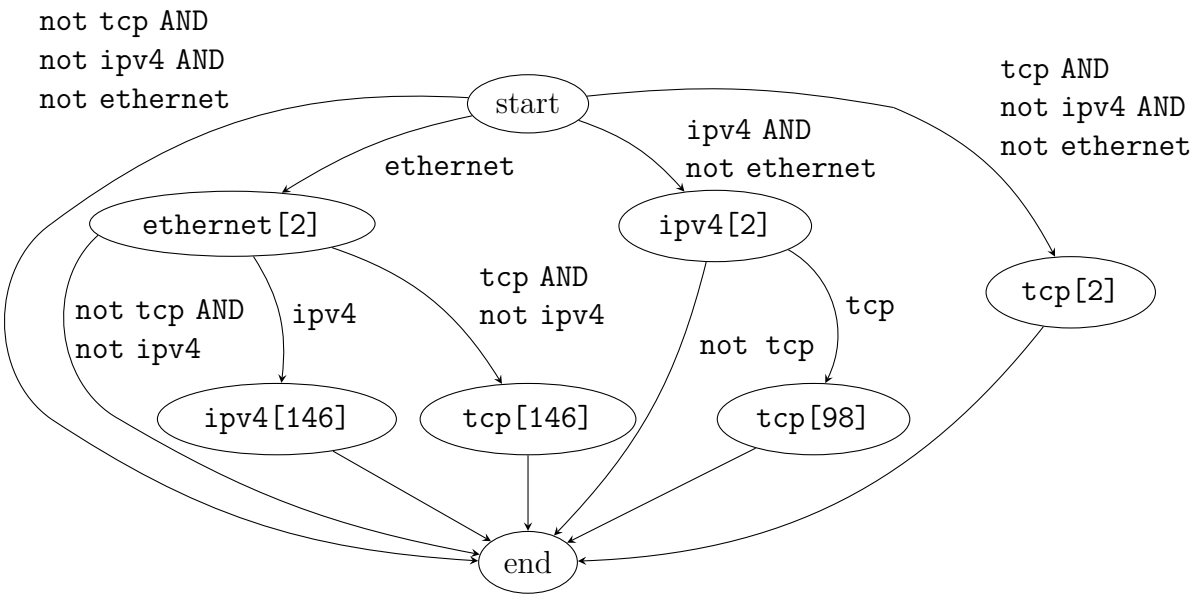


Figure 4.8 Sous-graphe pour le bit de sortie numéro 2 résultant de la décomposition du graphe de la figure 4.3 pour un bus de sortie de 256 bits

La machine à états est dérivée du sous-graphe. Chacun des sommets du sous-graphe représente un état et chaque arête une transition. Chacun des états de la machine à états détermine l'entrée à sélectionner sur le multiplexeur et les signaux `Header Valid` et `Header last`. La table de correspondance, construite lors de la création du multiplexeur, est utilisée pour associer à chacun des états de la machine à états le bit du multiplexeur à sélectionner.

Un exemple de sélecteur de bit d'en-tête, pour le bit de donnée de sortie «2» est montré dans la figure 4.9, il est construit en utilisant le GDA de la figure 4.8. La table de correspondance indique pour un état l'entrée du multiplexeur à sélectionner. Comme expliqué précédemment, la machine à états est directement dérivée du sous-graphe de déparsage. Le signal `Header Last` est activé si le prochain état est «end». Par exemple, si l'on est dans l'état `ethernet[2]` et que `tcp` et `ipv4` ne sont pas valides, alors le signal `Header Last` est mis à «1». Le signal `Header Valid` est à «0» dans les états «start» et «end», et à «1» dans les autres états.

4.3.3 La génération du bloc de sélection de la charge utile

Pour générer les sélecteurs de charge utile, il faut déterminer la position du dernier bit émis par la machine à état du déparseur pour chacune des combinaisons possibles d'en-têtes. Pour cela, l'algorithme 4.3 est utilisé. Dans cet algorithme, tous les chemins entre « start » et « end » du graphe de déparsage transitivement fermé sont parcourus. Pour chacun des chemins, le nombre total de bits d'en-tête insérés, HIw, est calculé ainsi que la liste, PHV, des en-têtes

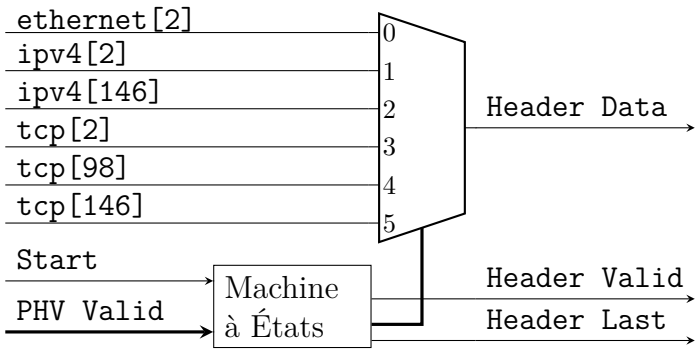


Table de correspondance

sommet	Position Mux
ethernet[2]	0
ipv4[2]	1
tcp[2]	3
ipv4[146]	2
tcp[98]	4
tcp[146]	5

Figure 4.9 Sélecteur de bit d'en-tête en utilisant le graphe de la figure 4.8

émis qui doivent donc être valides. Finalement, une liste contenant la liste des *PHV* avec le décalage à appliquer est retournée. Cette liste est ensuite utilisée pour générer les blocs de sélecteur de bit de charge utile en utilisant l'algorithme 4.4.

L'algorithme 4.4 permet de générer chaque sélecteur de bit de la charge utile. L'algorithme prend deux paramètres : une liste indiquant pour un ensemble d'en-têtes valides le décalage à appliquer, *PHVoff* et la position du sélecteur de bit de charge utile sur le bus de sortie, *Pos*. L'algorithme retourne une structure contenant les connexions de charge utile du paquet au multiplexeur : «PayloadShift.ConnexionMux», ainsi que la configuration du décodeur : «PayloadShift.Décodeur». Pour le décodeur, une association est construite indiquant pour chacune des combinaisons d'en-tête les valeurs que doivent prendre les multiplexeurs du sélecteur de bit d'en-tête. L'émission est décalée d'un cycle si le nombre de bits de décalage de la charge utile, *off*, est inférieur à la position du multiplexeur.

Un exemple de bloc de sélecteur de bit de charge utile, pour le bit de donnée numéro 2, est présenté dans la figure 4.10 avec la table de décodage associée. La table de décodage indique pour la valeur des signaux de validité d'en-tête `tcp` (T), `ipv4` (I) et `ethernet` (E) dans la colonne «TIE», la valeur que doivent prendre les multiplexeurs de sortie. Par exemple, si aucun en-tête n'est valide (TIE=000), le multiplexeur 1 prend le bit de charge utile 2 connecté à l'entrée 0 du multiplexeur et le multiplexeur 2 sort le signal sans un cycle de décalage. En effet si aucun en-tête n'est émis, alors seules les données de la charge utile sont émises et donc ces dernières n'ont pas besoin d'être décalées. Si les en-têtes `ethernet` et `ipv4` sont valides (TIE=011) alors on décale le signal de sortie d'un cycle en mettant à «1» la sortie du multiplexeur 2, et on sélectionne le bit de charge utile numéro 242 en sélectionnant le signal numéro 5 du multiplexeur 1. Dans ce cas, il faut émettre 112 bits pour `ethernet` et 160 bits pour `ipv4` ce qui fait un total de 272 bits à émettre. Comme le bus de sortie fait

256 bits, on décale de 272 mod 256 = 16 bits. Le bit de donnée numéro 2 sort donc le bit $256 - 16 + 2 = 242$ de la charge utile décalé de 1, car $2 < 16$.

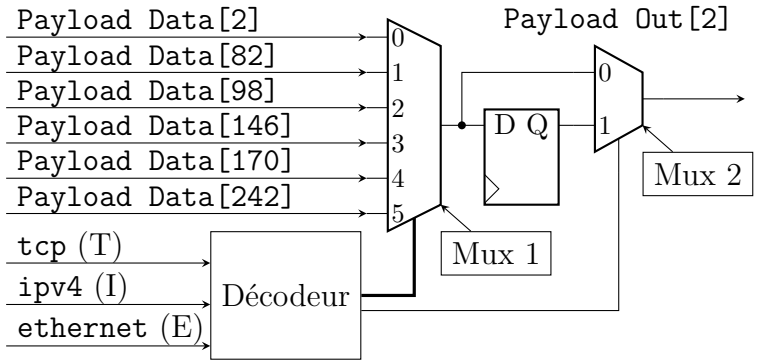


Table de décodage

TIE	Mux 1	Mux 2
000	0	0
001	3	1
010	2	1
011	5	1
100	2	1
101	5	1
110	4	1
111	1	1

Figure 4.10 Bloc de sélection de bit de la charge utile avec la table d'association du décodeur.

Algorithme 4.3 : Génération de la liste des décalage pour la charge utile

Entrée : G_d : GDA de déparsage
Entrée : w : largeur du bus de sortie
Sortie : *PHVoff* : liste d'association `PHV_valid` et décalage appliqué
PHVoff ← [];
Pour Chaque *p dans* G_d*.path()* **Faire**
  $HIw \leftarrow 0$; `// Nombre de bits d'en-tête émis`
  PHV ← G_d.getPHVValid(False) ; `// Liste d'en-têtes initialisés à Faux`
  Pour Chaque *S dans p* **Faire**
    $h \leftarrow$ getHeader(S);
    $HIw+ =$ size(h);
    PHV[h] ← Vrai;
  FinPourCh
  PHVoff.insert((PHV, HIw mod w))
FinPourCh
return *PHVoff*

4.4 Résultats d'implémentation de l'architecture

Dans cette section nous présentons les résultats d'implémentation de l'architecture générés avec Vivado 2019.1 en ciblant un FPGA Xilinx UltraScale+ *xcvu3p*. Dans un premier temps nous présentons des paramètres impactant la consommation de ressources. Puis nous présentons les tests effectués afin de regarder l'impact des différents paramètres lors de la

Algorithme 4.4 : Génération d'un bloc de sélection de bit de la charge utile de paquet

Entrée : *PHVoff* : Liste de décalage avec les valeur d'en-têtes valides
Entrée : *Pos* : position sur le bus de sortie
Sortie : PayloadShift : les paramètres d'un sélecteur de bit de charge utile
PayloadShift.Décodeur $\leftarrow$ [] ; $T_{off} \leftarrow$ [] ; $posMux \leftarrow 0$;
Pour Chaque PHV_{valid}, *off dans PHVoff* **Faire**
  Si *off not in* T_{off} **Alors** $T_{off}[off] \leftarrow posMux$; $posMux++$;
  $P.PHV \leftarrow \text{PHV}_{valid}$;
  $P.décale \leftarrow 0$; `// décalage d'un cycle faux`
  $P.pos \leftarrow T_{off}[off]$; `// position dans la charge utile`
  Si $Pos < off$ **Alors** $P.décale \leftarrow 1$; `// on doit décaler de 1 cycle`
  PayloadShift.Décodeur.insert(P);
FinPourCh
PayloadShift.ConnexionMux $\leftarrow T_{off}$;
return *PayloadShift*

génération de l'architecture. Finalement les résultats des différents tests sont présentés ainsi qu'une comparaison avec l'état de l'art.

4.4.1 Les paramètres impactant la génération du déparseur

La génération de l'architecture du déparseur dépend de trois paramètres : la taille des en-têtes, le nombre de chemins dans le graphe de déparsage et la largeur du bus de sortie. La taille des en-têtes ainsi que le nombre de chemins dans le graphe de déparsage découlent directement du programme P4 compilé ; la réduction possible du nombre de chemins dans le programme P4 est discutée dans le chapitre 5. La largeur du bus de sortie dépend du débit que l'on veut pouvoir supporter associé à la fréquence d'horloge que l'on peut obtenir.

L'architecture globale du déparseur résulte de la génération pour chacun des bits de sortie des sélecteurs de bit : d'en-tête, de charge utile et de sortie. La consommation de ressources d'un sélecteur de bit de sortie est fixe, le coût global des sélecteurs de bit de sortie dépend donc directement de la largeur du bus de sortie. Les sélecteurs de bit d'en-tête et de charge utile dépendent eux de la largeur du bus de sortie, du nombre de chemins dans le graphe et de la taille de chacun des en-têtes. La taille de chacun des en-têtes affecte la taille totale du PHV. Comme nous implémentons le déparseur depuis le graphe transitivement fermé, le nombre de chemins est 2^N pour N en-têtes. Ainsi pour évaluer l'implémentation de notre déparseur nous allons effectuer des tests qui feront varier la largeur du bus de sortie, des tests avec un nombre d'en-têtes variable, mais un PHV de taille fixe et des tests avec un nombre d'en-têtes fixe et des tailles de PHV variables.

4.4.2 Tests et résultats

Les tests et leurs caractéristiques sont présentés dans le tableau 4.1, ils ont tous été implémentés avec contrainte de période d'horloge de 1,6 ns, une interdiction d'utilisation de BRAM et des largeurs de bus de sortie de 64, 128, 192, 256, 320, 384, 448, 512 et 576 bits en ciblant un FPGA Xilinx UltraScale+ *xcvu3p* avec Vivado 2019.1. Les tests **T0**, **T1**, **T2** et **T3** utilisent des en-têtes de protocoles respectant les tailles des standards présentés dans le tableau 4.2. Dans le reste de cette section, nous présentons les résultats d'implémentations et discutons de l'impact des différents paramètres présenté précédemment.

Nous commençons par évaluer l'impact de la taille du bus de sortie en regardant les résultats pour les tests **T0**, **T1**, **T2**, **T3**. Puis nous évaluons l'impact du nombre de chemins est faite en comparant les résultats des tests **T0-1008**, **T1-1008**, **T2-1008** et **T3** pour lesquels le PHV a une largeur de 1008 bits. Par la suite, l'impact de la largeur du PHV est évalué en comparant les résultats de **T1**, **T1-1008**, **T1-2**, **T1-3**, **T1-4**. Puis nous regardons l'impact de l'interdiction de l'utilisation de BRAM dans le cas de **T3** en montrant les résultats avec et sans la contrainte sur les BRAM. Finalement, nous comparons les ressources utilisées par des déparseurs générés par Xilinx SDnet [53], ainsi que des déparseurs générés par Benáček et al. [17] par rapport aux résultats obtenus avec notre architecture de déparseur.

Tableau 4.1 Liste et caractéristique des cas de test pour l'implémentation du déparseur

Test	Nombre en-tête	Nombre chemin	Largeur PHV (bits)	Liste des en-têtes
T0	3	8	432	`ethernet`, `ipv4`, `tcp`
T1	5	32	816	**T0** + `tcp`, `udp`
T2	7	128	880	**T1** + `icmp`, `icmpv6`
T3	11	2048	1008	**T2** + 2× `vlan`, 2× `mpls`
T0-1008	3	8	1008	
T1-1008	5	32	1008	
T2-1008	7	128	1008	
T1-2	5	32	1632	
T1-3	5	32	2448	
T1-4	5	32	3264	

Impact de la largeur du bus de sortie

Les figures 4.11 et 4.12 présentent les résultats d'implémentation pour les différents cas en fonction de la largeur du bus de sortie. Dans la figure 4.11, le graphe représente pour chaque

Tableau 4.2 Taille des en-têtes de différents protocoles standards

Nom	Taille (Bits)	Taille (Octets)
`ethernet`	112	14
`ipv4`	160	20
`ipv6`	320	40
`tcp`	160	20
`udp`	64	8
`mpls`	32	4
`vlan`	32	4
`icmp`	32	4
`icmpv6`	32	4

déparseur la consommation de LUT et FF en fonction de la largeur du bus de sortie. On constate que pour chaque déparseur l'augmentation de la largeur du bus de sortie influence l'utilisation de FF et de LUT. L'utilisation de FF croît linéairement avec la largeur du bus de sortie. L'utilisation de LUT quant à elle augmente lorsque la largeur du bus de sortie augmente, mais est aussi dépendante du déparseur implémenter. Cela provient de la taille des différents en-têtes à déparser. Par exemple, pour **T3** dans le cas d'un bus de 512 bits, on constate une diminution du nombre de LUT par rapport à un bus de sortie de 448 bits. Cette variation est due à la taille des différents en-têtes à déparser, et à leur alignement sur le bus de sortie.

Dans la figure 4.12, le graphe représente pour chaque déparseur la fréquence et le débit maximal en fonction de la largeur du bus de sortie. On remarque que la fréquence d'horloge décline lorsque la largeur du bus de sortie augmente, cependant la fréquence est relativement stable pour les différents déparseurs jusqu'à un bus de sortie de 384 bits. Pour un bus de sortie de 384 à 576 bits, la fréquence maximale varie beaucoup plus en fonction du déparseur implémenté. Jusqu'à un bus de sortie de 512 bits, le débit augmente pour tous les déparseurs implémentés. Pour le bus de sortie de 576 bits, les déparseurs **T1** et **T3** on un débit maximal inférieur à leur implémentation pour un bus de sortie de 512 bits.

Impact du nombre de chemins

Pour évaluer l'impact du nombre de chemin dans le graphe de déparsage, nous avons implémenté **T0-1008**, **T1-1008**, **T2-1008** et **T3**. Chacun de ces déparseurs prend en entrée un PHV de 1008 bits, ils ont respectivement 8, 32, 128 et 2048 chemins dans leur graphe de déparsage. Le tableau 4.3 présente les résultats d'implémentation de **T0-1008** (8 chemins)

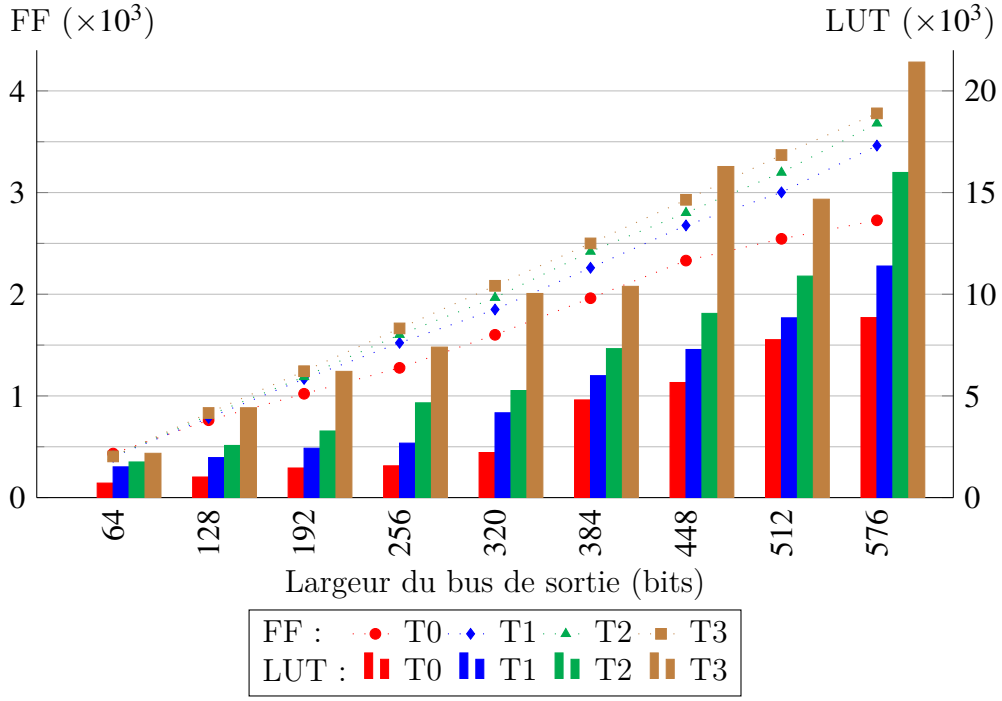


Figure 4.11 Utilisations de LUT et FF par l'implémentation de **T0**, **T1**, **T2** et **T3** pour différentes tailles de bus

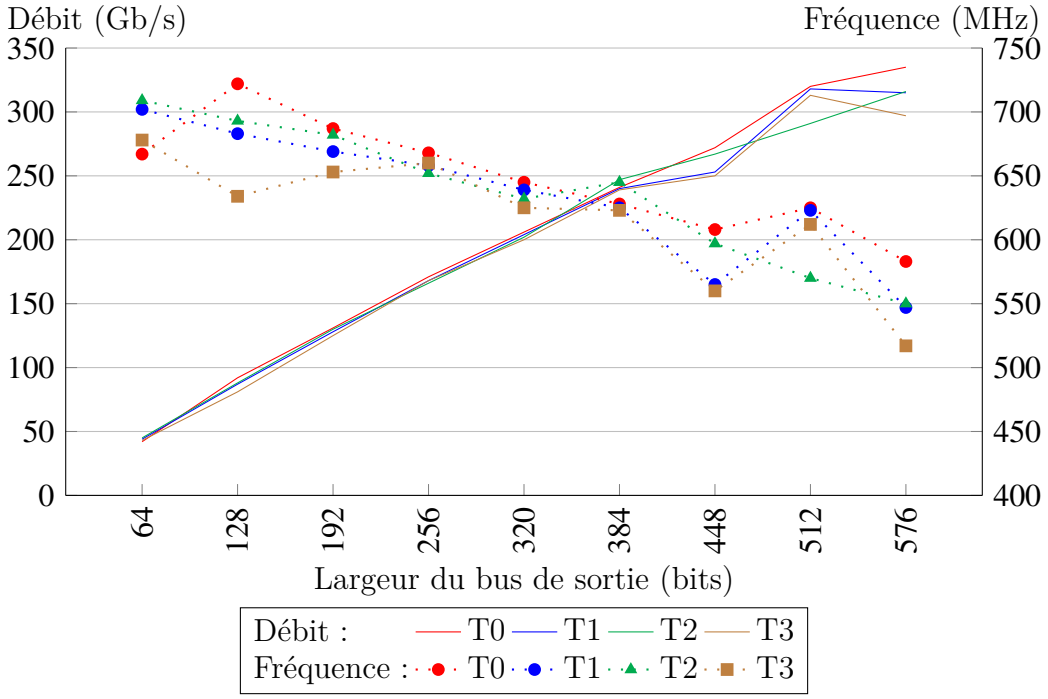


Figure 4.12 Fréquence et débit maximum obtenues après l'implémentation de **T0**, **T1**, **T2** et **T3** pour différentes tailles de bus

pour les différents bus de sortie. La figure 4.13 montre l'écart de consommation pour les différentes largeurs de bus de sortie de **T1-1008**, **T2-1008** et **T3** par rapport à **T0-1008**.

On constate que la fréquence reste relativement stable lorsque le nombre de chemins augmente. Aussi, la consommation de FF reste équivalente entre les différents déparseurs en fonction du nombre de chemins, à part pour le cas avec le bus de sortie de 64 bits. Dans le cas du bus de 64 bits, **T1-1008, *T2-100** et **T3** requièrent la moitié de FF par rapport à **T0-1008**. cette différence s'explique par un alignement des différents en-têtes avec le bus de 64 bits. Les LUT sont les éléments étant le plus affecté par l'augmentation du nombre de chemins avec une augmentation maximale dans le cas de **T3-1008** avec un bus de sortie de 256 bits, le déparseur implémenté requiert $3,5\times$ plus de LUT que **T0-1008** avec la même largeur de bus de sortie. Cette différence provient du fait que chacun des multiplexeurs des sélecteurs de bits d'en-tête et de charges utiles sont plus gros, car plus de combinaisons sont possibles. Globalement, on constate que l'utilisation de LUT augmente lorsque le nombre de chemins augmente.

Tableau 4.3 Résultats d'implémentation de **T0-1008** pour différentes largeurs de bus de sortie

Bus de sortie (bits)	64	128	192	256	320	384	448	512	576
FF	689	806	1253	1480	1876	2111	2420	2805	2980
LUT	1480	1777	2250	2091	4092	5715	6437	8523	9749
Fréquence (MHz)	681	660	664	627	630	626	625	630	589

Impact de la taille du PHV

Afin d'évaluer l'impact de la taille du PHV sur la consommation de ressources, nous avons implémenté des cas de test avec le même nombre de chemins, mais avec un nombre total de bits de PHV différents. Les tests implémentés sont **T1**, **T1-1008**, **T1-2**, **T1-3** et **T1-4**. Tous ces tests ont un graphe de déparsage avec 32 chemins, mais des PHV de respectivement, 816, 1008,1632, 2448, 3264 bits. Les résultats d'implémentation de **T1** pour différentes largeurs de bus sont présentés dans le tableau 4.4. Nous comparons le ratio des FF, LUT et la fréquence par rapport **T1** pour **T1-1008**, **T1-2**, **T1-3** et **T1-4**, la figure 4.14 présente les ratios pour chaque déparseur et les différentes largeurs de bus.

On constate que l'utilisation de FF est stable pour les différentes tailles de PHV. Également la fréquence maximale varie légèrement pour les tests par rapport à **T1**, elle ne diminue jamais de plus de 10% et peut augmenter jusqu'à 20%. L'utilisation de LUT est impactée par la taille du PHV. Leur utilisation est par exemple $2,5\times$ supérieur pour **T1-3** par rapport à

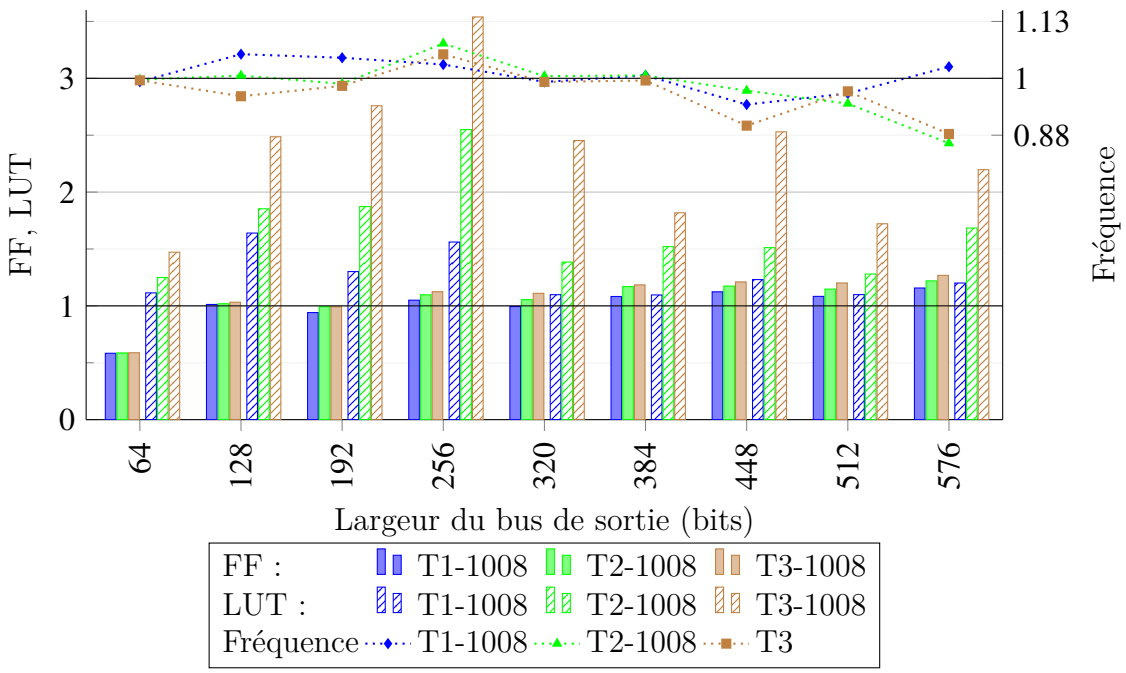


Figure 4.13 Ratios des FF, LUT et de la fréquence de **T1-1008**, **T2-1008** et **T3** par rapport à **T0-1008**, pour différentes largeurs de bus de sortie

T1 dans le cas d'un bus de sortie de 256 bits. Le cas **T1-3** est le cas ou l'augmentation de l'utilisation de LUT est la plus importante pour toutes les largeurs de bus de sortie bien que le PHV soit moins large que celui de **T1-4**. Il n'y a donc pas de corrélation entre la taille du PHV et l'utilisation de LUT. Aussi pour des bus de sortie de plus de 256 bits, l'utilisation de LUT est moins impactée que pour des bus de sortie de moins de 256 bits. Cela provient du fait que pour des bus de sortie plus large, les en-têtes peuvent être insérés en entier sur le bus. Ainsi, il y a moins de bits d'entrée sur les multiplexeurs. Globalement, l'augmentation de la taille du PHV a moins d'impact que l'augmentation du nombre de chemins.

Tableau 4.4 Résultats d'implémentation de **T1** pour différentes largeurs de bus de sortie

Bus de sortie (bits)	64	128	192	256	320	384	448	512	576
FF	402	784	1167	1522	1850	2260	2677	3002	3462
LUT	1511	1967	2428	2678	4170	5996	7283	8840	11387
Fréquence (MHz)	702	683	669	658	639	625	565	623	547

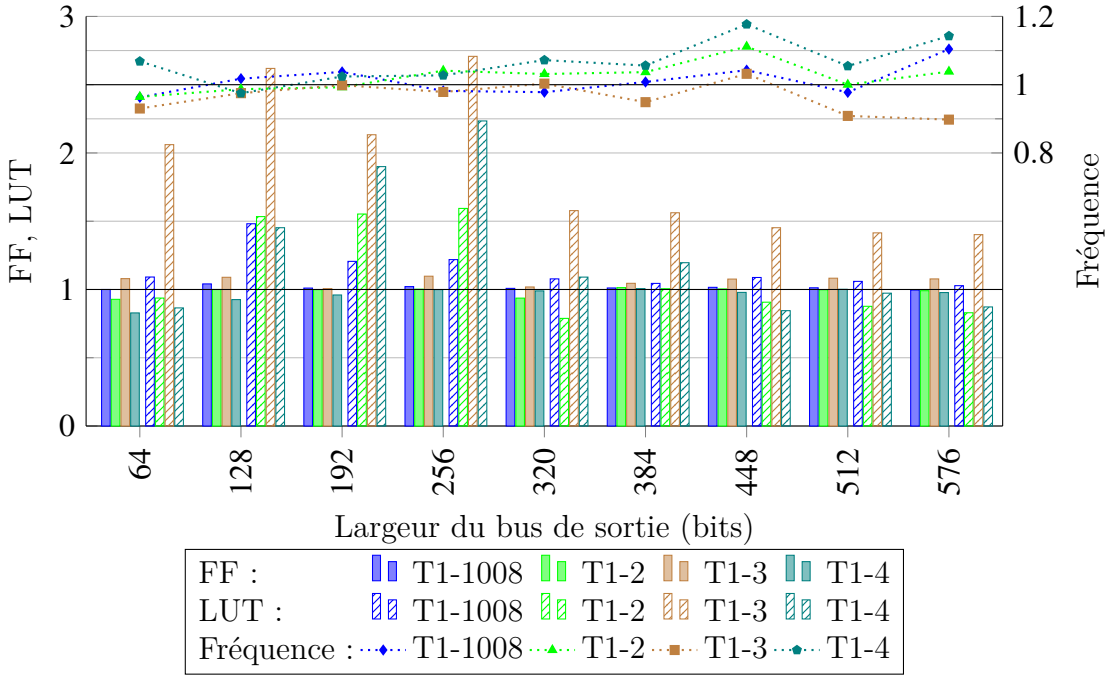


Figure 4.14 Ratios des FF, LUT et de la fréquence de **T1-1008**, **T1-2**, **T1-3** et **T1-4** par rapport à **T1** pour différentes largeurs de bus de sortie

Impact de la contrainte sur les BRAM

Tous les résultats présentés précédemment ont été effectués en indiquant à Vivado de ne pas utiliser de BRAM. Dans la figure 4.15, nous présentons le gain dans le cas de **T3** lors de l'implémentation sans BRAM par rapport au résultat d'implémentation si l'on laisse Vivado utiliser de BRAM. Aussi, nous montrons le nombre de BRAM18k utilisés par la version implémentée avec BRAM. Les résultats obtenus montrent que de ne pas utiliser de BRAM permet d'augmenter la fréquence d'horloge d'au moins 10 % et en moyenne de 22 %. Cela s'accompagne d'une augmentation du nombre de LUT et FF. Dans le cas des LUT, l'augmentation peut aller jusqu'à 24 %, avec une augmentation moyenne de 9 % par rapport à la version avec BRAM. Pour ce qui est des FF, l'augmentation maximale est de 11 % avec une augmentation moyenne de 8 %. Cependant, ces résultats sont à mettre en perspective avec le nombre de BRAM utilisés. Par exemple dans le cas d'un bus de sortie de 512 bits, la version avec BRAM utilise 46 BRAM de 18k, et utilise 994 LUT et 292 FF de moins que la version sans BRAM. Le choix de l'implémentation du déparseur avec ou sans BRAM dépend donc de la ressource la plus limitante dans le pipeline global qui sera implémenté sur un FPGA. Toutefois, pour maximiser le débit, il faut mieux interdire à l'outil d'utiliser des BRAM dans le déparseur.

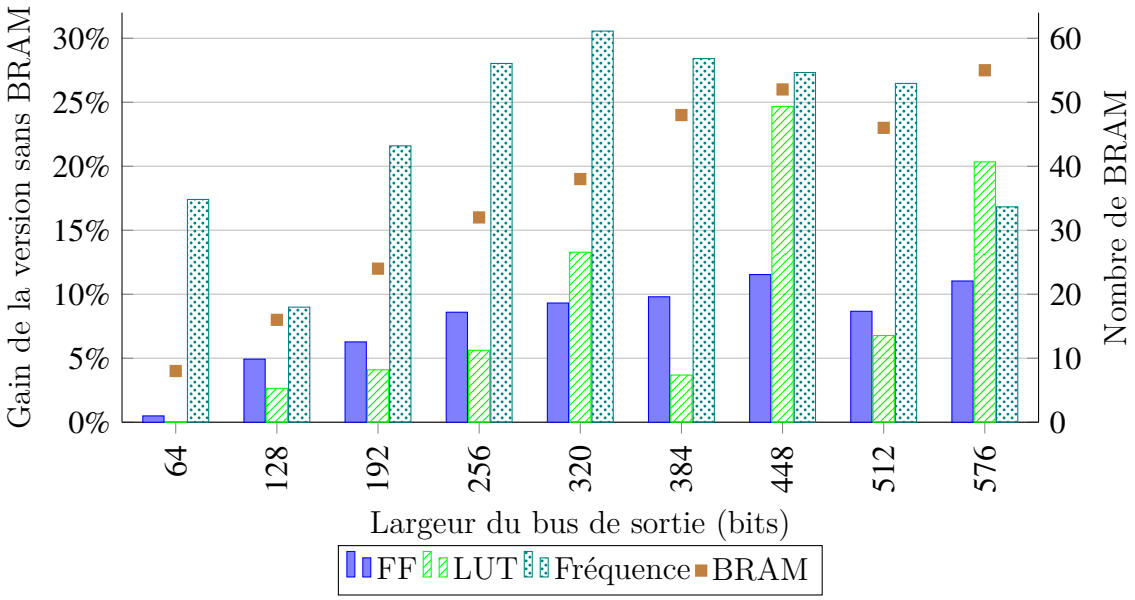


Figure 4.15 Résulats d'implémentation de **T3** avec et sans BRAM

Comparaison avec d'autres travaux

Le déparseur proposé a été comparé, pour un bus de 512 bits, avec les travaux de Benáček et al.[17] ainsi qu'avec le déparseur généré par la version 2017.4 de Xilinx SDNet. Les résultats d'implémentation des différentes implémentations de déparseur sont présentés dans le tableau 4.5. On constate par rapport à la solution proposée par Xilinx SDNet une division par plus de 30× de la consommation de bascules ainsi qu'une division par 9× de l'utilisation de LUT. Également, le déparseur de Xilinx utilise des BRAM en grande quantité. Même lors de l'implémentation de notre architecture avec BRAM montrée dans la figure 4.15, nous utilisons 46 BRAM pour **T3** avec un bus de sortie de 512 bits, la solution de Xilinx en utilise $3,5\times$ plus. En plus de la baisse de consommation de ressources, nous avons un débit plus élevé que la solution de Xilinx SDNet. Également, en comparant avec le travail proposé par Benáček et al., le déparseur présenté dans cette thèse consomme de 4× à 5× moins de ressources et permet d'atteindre un débit plus de 2× supérieur.

Nous n'avons pas accès aux détails d'implémentation du déparseur de Xilinx SDNet, nous ne savons donc pas pourquoi leur architecture consomme autant de ressources. Dans le cas du travail de Benáček et al., le déparseur proposé se structure autour d'un décaleur de bits qui est paramétré lors de l'exécution pour sélectionner les bits de PHV à insérer. Dans le cadre de l'architecture de déparseur que nous proposons, nous fixons la sélection des bits de PHV lors de la synthèse, cela a pour effet de tirer avantage du tissu d'interconnexions du FPGA.

Nous pensons donc que c'est pour cela que nous arrivons à réduire le coût d'implémentation du déparseur par rapport au travail de Benáček et al. [17].

Tableau 4.5 Comparaison de l'implémentation de déparseurs avec d'autres travaux pour un bus de 512 bits.

Test	Travail	Tranches équivalentes	LUT	FF	BRAM	Débit
T2	Ce travail	4058	11k	3.2k	0	291 Gb/s
	Xilinx SDNet	N/A	98 k	119 k	149.5	240 Gb/s
	Benáček et al. [17]	20 k	N/A	N/A	N/A	120 Gb/s
T3	Ce travail	5252	15k	3.4k	0	313 Gb/s
	Xilinx SDNet	N/A	139 k	165 k	229.5	220 Gb/s
	Benáček et al. [17]	24 k	N/A	N/A	N/A	120 Gb/s

CHAPITRE 5 SIMPLIFIER L'IMPLÉMENTATION D'UN DÉPARSEUR

Dans le chapitre 4, nous avons proposé une transformation du graphe de déparsage ainsi qu'une architecture de déparseur permettant de prendre avantage de cette transformation. Également, nous avons montré que le nombre de chemins dans le graphe de déparsage impacte la consommation de ressources de l'architecture générée. Dans ce chapitre, nous proposons de spécialiser un programme P4 afin de réduire le nombre de chemins dans le graphe de déparsage pour réduire le coût d'implémentation du déparseur sur FPGA. Ce travail a été soumis dans la revue «ACM Transactions on Architecture and Code Optimization» [109].

Tout d'abord, nous présentons un exemple de simplification du graphe de déparsage en étudiant un programme P4 (Section 5.1). Par la suite, nous présentons une solution pour faire l'analyse symbolique d'un programme P4 afin de déterminer les états possibles du vecteur d'en-têtes d'un programme P4 (Section 5.2). Ceci nous permet de proposer une solution pour simplifier le Graphe Dirigé Acyclique (GDA) de déparsage (Section 5.3). Finalement, les gains d'implémentation obtenus grâce à la solution proposée sont présentés (Section 5.3.3).

5.1 Exemple d'optimisation

Dans cette section, nous présentons un exemple d'optimisation d'un programme P4. Nous considérons un code P4 composé uniquement d'un parseur de paquets suivi d'un déparseur. Le code du parseur est montré dans le listage 5.1 et celui du déparseur dans le listage 4.1. Ce programme parse un paquet ayant les en-têtes `ethernet`, `ipv4` et `tcp`, puis effectue l'émission du paquet sans le modifier.

5.1.1 Présentation du code de parsage

Le parseur présenté dans le listage 5.1 commence le parsage à l'état `start`. La première chose que fait le parseur est l'extraction de l'en-tête `ethernet`. Lors de l'extraction de l'en-tête, son champ de validité est mis à vrai. Puis, si le champ `ethType` est égal à `0x800`, alors le parseur passe à l'état `parse_ipv4` ; dans les autres cas, le parsage se termine avec la transition `accept`. À l'état `parse_ipv4`, l'en-tête `ipv4` est extrait, puis si le champ `protocol` de l'en-tête extrait est égal à 6, alors on va à l'état `parse_tcp`, sinon on va à l'état `accept`. Finalement, l'état `parse_tcp` extrait l'en-tête `tcp` puis va à l'état `accept`. Le graphe de parsage résultant de ce code est présenté dans la figure 5.1.

Si on suit le processus de parsage, on constate qu'à la fin du parseur le vecteur d'en-têtes peut contenir uniquement trois combinaisons d'en-têtes active en même temps. Soit uniquement `ethernet` est valide, soit seulement `ethernet` et `ipv4` sont valides, soit les trois en-têtes `ethernet`, `ipv4` et `tcp` sont valides. Comme aucune autre modification n'est faite sur les en-têtes, le déparseur déparsera uniquement ces trois combinaisons d'en-têtes.

```
parser MyParser(packet_in Pkt_in, out headers PHV) {
  state start {
      Pkt_in.extract(PHV.ethernet);
      transition select(PHV.ethernet.ethType) {
          0x800   : parse_ipv4;
          default : accept;} }
  state parse_ipv4 {
      Pkt_in.extract(PHV.ipv4);
      transition select(PHV.ipv4.protocol) {
          6       : parse_tcp;
          default : accept;} }
  state parse_tcp { Pkt_in.extract(PHV.tcp);
      transition accept; } }
```

Listage 5.1 Un parseur de paquets écrit en P4 pour les en-têtes `ethernet`, `ipv4` et `tcp`

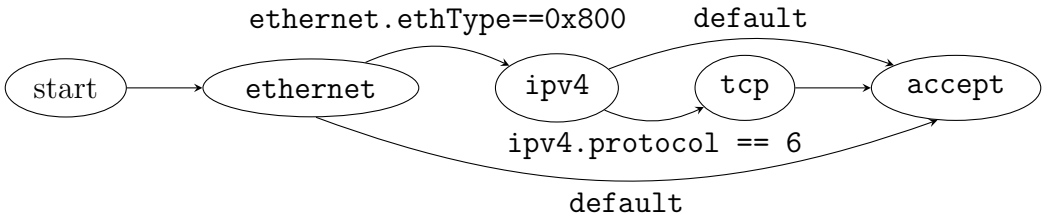


Figure 5.1 Graphe de parsage résultant du listage 5.1

5.1.2 Un graphe de déparsage réduit valide

Si on considère le parseur présenté précédemment et un code qui ne modifie jamais les en-têtes, alors le déparseur ne traitera que les trois combinaisons d'en-têtes présentées précédemment : `ethernet`, `ethernet` et `ipv4` ou `ethernet`, `ipv4` et `tcp`. Si l'on prend le cas du déparseur présenté dans le listage 4.1 avec son graphe de déparsage transitivement fermé présenté dans la figure 4.3, alors le GDA de déparsage transitivement fermé peut être réduit. En effet, l'en-tête `ethernet` étant toujours valide, seule la transition entre «start» et `ethernet` doit

être gardée, et comme une seule transition est possible, la condition qui lui est associée peut être enlevée. De plus, la condition de l'arête `ethernet-tcp` n'est jamais valide puisque dans le seul cas où `tcp` est valide, `ipv4` l'est également, cette arête peut donc être enlevée. Ces modifications permettent d'obtenir le graphe de déparsage réduit présenté dans la figure 5.2.

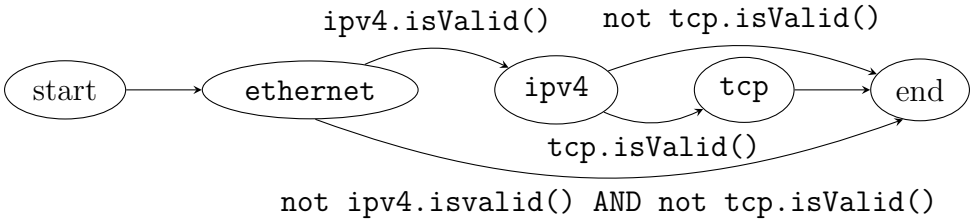


Figure 5.2 Exemple d'un graphe de déparsage réduit

Pour construire ce GDA de déparsage, il est nécessaire de connaître les combinaisons d'en-têtes pouvant être émis par le déparseur, puis de faire une réduction du GDA de déparsage transitivement fermé. Dans le reste de ce chapitre, nous présentons une approche pour l'analyse du vecteur d'en-têtes afin de déterminer les combinaisons d'en-têtes valides, puis nous présentons la méthode proposée pour effectuer l'élagage du graphe de déparsage.

5.2 Déterminer les états possibles du vecteur d'en-têtes par l'analyse symbolique

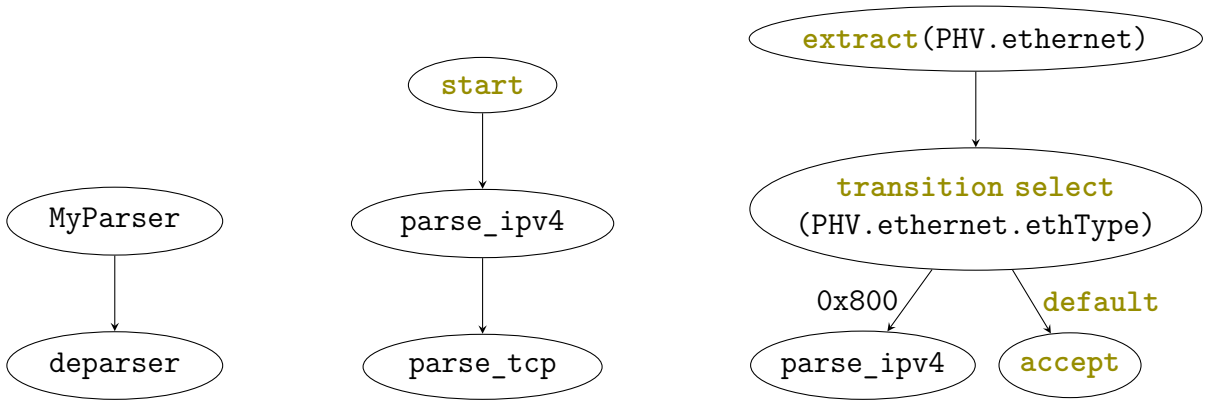
La section précédente présente un exemple d'analyse de programme P4 permettant de réduire la taille du graphe de déparsage. Pour faire cela, il faut déterminer les combinaisons d'en-têtes pouvant être émis. Dans cette section nous présentons une analyse symbolique pour déterminer ces combinaisons d'en-têtes. Nous commençons par présenter les différentes fonctions affectant le bit de validité du vecteur d'en-têtes. Par la suite, nous présentons la manière dont nous interprétons les blocs conditionnels d'un programme P4. Finalement, nous discutons de la complexité de l'analyse symbolique.

5.2.1 Vue d'ensemble de l'analyse symbolique d'un programme P4

L'analyse symbolique permet de générer un ensemble, *HeaderV*, contenant les combinaisons possibles d'en-têtes étant actifs en même temps. Une combinaison d'en-têtes est enregistrée dans le Vecteur de Validité d'En-tête (VVE). Le VVE associe, à chacun des en-têtes du programme P4, la valeur de son champ de validité. Dans les programmes P4, tous les champs de validité des en-têtes sont invalides au début du programme. Au début de l'analyse sym-

bolique, l'ensemble *HeaderV* est donc initialisé avec un seul VVE dans lequel chacun des en-têtes est invalide.

Un programme P4 peut se représenter sous la forme d'un GDA. Chaque structure du programme est représentée par un sommet spécifique dans le GDA, l'analyse d'un programme P4 consiste à visiter les sommets du GDA. Ainsi les blocs `control` et `parser`, sont représentés par des sommets, comme montré dans la figure 5.3a dans le cas du programme, composé d'un parseur et d'un déparseur, utilisé dans la section 5.1. Les sommets du GDA représentant un bloc de code contiennent un sous GDA représentant ce bloc de code comme montré dans les figures 5.3b et 5.3c. L'analyse symbolique commence donc par évaluer le premier sommet du GDA, puis visite récursivement tous les sommets des sous GDA. Le premier sommet est évalué avec l'ensemble *HeaderV* contenant un seul VVE avec les en-têtes invalides, l'évaluation du sommet consiste à visiter les sous GDA. Puis le sommet suivant est évalué en utilisant l'ensemble *HeaderV* obtenu lors l'analyse du sommet précédent.



(a) GDA principal (b) GDA du sommet `MyParser` (c) GDA du sommet `start`

Figure 5.3 Exemples de GDA et sous GDA pour un programme P4 composé des listages 5.1 et 4.1

5.2.2 Les fonctions affectant la validité du vecteur d'en-têtes

La spécification P4 définit trois fonctions pouvant modifier la validité d'un en-tête : `extract`, `setValid` et `setInvalid`. La fonction `extract` prend en paramètre un en-tête et met son champ de validité à vrai. Cette fonction est associée à l'`extern packet_in`. La fonction `setValid` est associée au type `header` et met le champ de validité de l'en-tête à vrai. Finalement, la méthode `setInvalid` est également associée au type `header` et met le champ de validité de l'en-tête à faux.

Pour effectuer l'analyse symbolique des méthodes, l'algorithme 5.1 est appliqué. Pour chaque sommet du GDA représentant une méthode, nous regardons si cette méthode peut affecter le champ de validité de l'en-tête. Si tel est le cas, *HeaderV* est mis à jour en conséquence. Sinon, l'ensemble n'est pas modifié puis le sommet suivant du GDA est évalué.

Algorithme 5.1 : Analyse symbolique d'un appel de fonction

Entrée : *HeaderV* : Ensemble de Vecteurs de Validité d'En-tête (VVE)
Entrée : *func* : fonction appelée
Sortie : *HeaderV* : Esemble de VVE mis à jours
Si *func* n'est pas `setValid` OU `setInvalid` OU `extract` **Alors**
  return *HeaderV*; `// On ne change pas l'ensemble`
FinSi
hdr ← getHeader(Method) ; `// On récupère l'en-tête modifié`
Si *Method est* `setInvalid` **Alors** newVal ← Faux **Sinon** newVal ← Vrai;
Pour Chaque *VVE dans HeaderV* **Faire** VVE[hdr] ← newVal;
return *HeaderV*

5.2.3 L'évaluation symbolique dans les blocs conditionnels

Il existe dans le langage P4 trois types de blocs conditionnels. Il y a les structures basées sur la sélection de transition «=transition select=», présentes uniquement dans le parseur, les tables «=table=», présentes dans les blocs de contrôle, et les conditions «=if=». Ces différents blocs conditionnels sont tous composés de branchements. Les conditions «=if=» peuvent avoir un ou deux branchements. Le nombre de branchements des tables est égal au nombre d'actions qu'elles possèdent. Dans le cas des «=transition select=», le nombre de branchements est égal au nombre de valeurs possibles que la condition de transition doit considérer.

Dans le cas des tables et des choix de transition, l'évaluation de la condition n'est pas faisable. Dans le cas de la sélection de transition dans le parseur, cela dépend, dans la majorité des cas, du paquet reçu en entrée. Pour ce qui est des tables, les actions à sélectionner sont déterminées par la condition ainsi que par les règles insérées par le plan de contrôle. Pour ces raisons, nous n'évaluons pas les conditions des blocs conditionnels et nous considérons que pour n'importe quel *HeaderV* (algorithme 5.2) , tous les branchements doivent être évalués. De cette manière, tous les cas possibles sont intégrés, il est cependant possible que certains VVE ne soient jamais rencontrés lors de l'exécution du code.

Ces différentes considérations amènent à l'utilisation de l'algorithme 5.2 pour faire l'analyse symbolique des blocs conditionnels. L'algorithme présenté est utilisé à chaque fois qu'un sommet rencontré lors de l'analyse du GDA est un des trois blocs conditionnels. Nous visitons

chacune des branches, avec le *HeaderV* passé en paramètre. La visite de la branche retourne un nouvel ensemble de VVE. Cet ensemble est ensuite fusionné dans *retHdrV*. Lors de la fusion, les doublons sont enlevés.

Algorithme 5.2 : Analyse symbolique des blocs conditionnels

Entrée : *HeaderV* : Esemble de Vecteurs de Validité d'En-tête (VVE)
Entrée : *Branches* : liste des branchements
Sortie : *retHdrV* : VVE mis a jours
retHdrV ← ∅ ;
Pour Chaque *b dans Branches* **Faire** `// Prochains sommets possibles`
  newHdrV = visit(b, HeaderV) ; `// Visite la branche avec HeaderV`
  merge(retHdrV, newHdrV) ; `// Fusion de newHdrV dans retHdrV`
FinPourCh
return retHdrV

5.2.4 Exemple d'évaluation symbolique

Les sections précédentes présentent les algorithmes pour l'analyse symbolique de certaines structures P4. Dans cette section, nous présentons un exemple d'évaluation symbolique appliquée au parseur du listage 5.1. L'analyse commence à l'état `start` avec un ensemble *HeaderV* contenant un VVE contenant trois en-têtes «Invalide» : `ethernet`, `ipv4` et `tcp`.

Lors de l'analyse de l'état `start`, le premier sommet rencontré est la fonction `extract`, le GDA de ce sommet est montré dans la figure 5.3c. L'algorithme 5.1 est appliqué. Comme la fonction `extract` modifie le champ de validité de l'en-tête, on récupère l'en-tête `ethernet` sur lequel la fonction va être appliquée. Puis pour chacun des VVE de *HeaderV*, on met l'en-tête `ethernet` à «Valide». Une fois la fonction analysée, on analyse le sommet suivant du GDA du bloc. Le sommet suivant est une condition, `transition select(PHV.ethernet.ethType)`, avec deux branches, une pour aller à l'état `parse_ipv4` et `default : accept` finissant le parsage. L'algorithme 5.2 est donc utilisé pour évaluer ce sommet, nous visitons donc chacune des branches et fusionnons l'ensemble. La visite de la branche `default` retourne le même ensemble *HeaderV* avec seulement l'en-tête `ethernet` valide. La visite de l'autre branche amène à l'analyse de l'état `parse_ipv4`.

Dans l'état `parse_ipv4`, la fonction `extract` est appelée sur l'en-tête `ipv4`. Après l'analyse de cette méthode, le VVE de l'ensemble *HeaderV* a maintenant les en-têtes `ethernet` et `ipv4` valides. Après l'analyse de la fonction, un sommet conditionnel avec deux branches, une vers `parse_tcp` et une vers `accept`, est rencontré. Le résultat de l'analyse de la seconde branche retourne un ensemble de VVE non modifié. L'analyse de la première branche consiste à effectué l'analyse de l'état `parse_tcp`.

Dans l'état `parse_tcp` la fonction `extract` est appelée, et met à jours l'en-tête `tcp`. Lors de l'analyse de cet état, on a donc un ensemble contenant un VVE avec les trois en-têtes, `ethernet`, `ipv4` et `tcp` valident. Une fois l'analyse de `parse_tcp` terminée, on retourne cet ensemble pour faire la fusion dans l'analyse de la condition de `parse_ipv4`, nous obtenons ainsi un ensemble *retHdrV* contenant deux VVE un avec les trois en-têtes valides et un avec `ethernet` et `ipv4` valides. Après cette fusion on retourne cet ensemble pour faire la fusion de l'analyse de la condition de `start`. Le résultat de la fusion génère l'ensemble de trois VVE présenté dans le tableau 5.1.

Tableau 5.1 Ensemble de VVE après l'analyse symbolique du parseur du listage 5.1

	`ethernet`	`ipv4`	`tcp`
1	Valide	Invalide	Invalide
2	Valide	Valide	Invalide
3	Valide	Valide	Valide

5.2.5 La complexité de l'analyse symbolique

Dans les sections précédentes, nous avons montré les algorithmes utilisés pour faire l'analyse symbolique d'un programme P4. Dans cette section, nous expliquons pourquoi dans le cas du langage P4 l'analyse symbolique peut être réalisée. Dans un premier temps, nous présentons certaines spécificités du langage qui rendent l'analyse possible. Par la suite, nous discutons de la taille de l'ensemble de VVE. Puis, nous détaillons la complexité de l'analyse par l'évaluation du nombre total de chemins possibles dans le graphe analysé. Finalement, nous discutons de la minimalité de l'ensemble de VVE généré.

Les spécificités du langage P4

Comme expliqué précédemment, l'analyse symbolique détermine les valeurs que peuvent prendre les variables d'un programme. L'analyse symbolique est généralement limitée à certaines portions d'un programme à cause du problème de l'arrêt [110]. Dans le cas du langage P4, trois de ses propriétés nous permettent de rendre cette analyse possible sur le champ de validité du vecteur d'en-têtes. La première propriété porte sur le fait que le champ de validité du vecteur d'en-tête est une valeur booléenne, ce qui limite l'ensemble des possibilités à prendre en compte. La seconde propriété est que le nombre d'en-têtes est connu lors de la compilation. Finalement, la dernière propriété est que tout programme P4 peut se traduire comme un GDA, l'évaluation de tous les chemins est donc garantie d'être fini [111].

La taille de l'ensemble des combinaisons d'en-tête

L'objectif de l'analyse symbolique est de générer un ensemble de VVE valide, qui devra être stocké en mémoire lors de l'analyse. Par conséquent, il est essentiel de déterminer quelle peut être la taille de cet ensemble. Un VVE indique pour chacun des en-têtes s'il est valide ou non. Cette condition prend donc l'une ou l'autre de deux valeurs possibles. Par conséquent, le nombre maximal de VVE possibles est de 2^N où N est le nombre d'en-têtes. Ce nombre maximal devrait rester petit pour deux raisons principales : le nombre d'en-têtes dans un commutateur est de l'ordre de la dizaine et le nombre de combinaisons est limité. Pour ce qui est du nombre d'en-têtes, Gibb et al. [40] présentent un graphe de parsage pour un commutateur intégrant quatre cas d'utilisations : en entreprise, dans un centre de données, en périphérie du réseau et pour un fournisseur de services. Ces quatre cas combinés requièrent 22 en-têtes différents.

Aussi, dans le cas des réseaux, chaque en-tête correspond à une couche dans la pile de protocoles réseau [112]. Par exemple, dans un réseau TCP/IP, l'en-tête Ethernet correspond à la couche liaison, IP à la couche réseau et TCP à la couche transport. Ainsi, comme chaque en-tête appartient à une couche, certains en-têtes ne peuvent pas être actifs en même temps, donc le nombre de VVE s'en trouve réduit. Par exemple, dans le réseau internet standard, un en-tête de la couche transport est toujours précédé d'un protocole de la couche réseau, donc il ne peut jamais être le seul en-tête valide. Il en va de même pour la couche réseau qui est toujours précédée d'un en-tête de la couche liaison. Ainsi dans une pile de protocoles réseau avec Ethernet, IP et TCP, si l'en-tête TCP est valide alors les en-têtes Ethernet et IP le sont aussi. Le nombre d'éléments dans l'ensemble de VVE sera bien plus petit que le nombre théorique maximal. Dans le cas du graphe de parsage de Gibb et al. [40], l'ensemble de VVE contient seulement 677 éléments en comparaison du nombre maximal de VVE possible de plus de 4 millions.

La complexité de l'analyse symbolique

L'analyse symbolique d'un programme P4 consiste à parcourir tous les sommets du GDA du programme. Lors de ce parcours, chacun des sommets n'est évalué qu'une fois, la complexité du parcours d'un programme P4 est donc $O(M)$ avec M le nombre de sommets. Dans l'ensemble des sommets analysés, seuls les sommets de type conditionnel et les sommets appliquant une méthode sont considérés. La complexité de l'analyse des méthodes (algorithme 5.1) dépend de la taille de l'ensemble de VVE qui dans le pire cas est de 2^N avec N le nombre d'en-têtes. La pire complexité est donc de 2^N si la méthode est parmi les trois méthodes modifiant l'en-tête, dans les autres cas la complexité est $O(1)$. La complexité de l'analyse d'un

sommet conditionnel (algorithme 5.2) est déterminée par l'opération de fusion de deux ensembles de VVE soit $O(n_1 \times n_2)$ avec n_1 et n_2 les tailles respectives des deux ensembles. La taille d'un ensemble peut être jusqu'à 2^N, donc la complexité de la fusion peut être, dans le pire cas, de 2^{2N}. Cette opération est effectuée autant de fois qu'il y a de branches dans le sommet conditionnel. Une telle complexité devient rapidement excessive quand N augmente.

Par conséquent, on constate que la complexité de l'analyse symbolique est principalement dépendante de la taille de l'ensemble VVE ainsi que du nombre de modifications apportées au champ de validité du vecteur d'en-tête. Si aucune méthode n'est appelée pour modifier le vecteur d'en-têtes alors la complexité est de $O(M)$. Cependant, dans un cas où tous les noeuds modifient le vecteur d'en-têtes et toutes les combinaisons d'en-têtes sont possibles, alors la complexité est de $O(M \times 2^N)$. L'implémentation de cette passe d'analyse dans un compilateur devrait donc intégrer une borne supérieure sur la taille totale de l'ensemble de VVE afin de garantir que la compilation se termine toujours dans un temps raisonnable.

La minimalité de l'ensemble de VVE

L'analyse symbolique fournit un ensemble de VVE intégrant les combinaisons d'en-têtes pouvant être valide à différents points du programme P4. Cet ensemble est par la suite utilisé pour effectuer la spécialisation du programme notamment pour réduire le graphe de déparsage. Par conséquent, l'ensemble *HeaderV* doit être minimal et complet pour permettre de réduire au minimum le déparseur tout en garantissant sa validité.

Définition 5.1. Un ensemble *HeaderV* de VVE associé à un point d'exécution du programme P4 est minimal si et seulement si tout élément de *HeaderV* peut être généré, à ce point d'exécution, par au moins un paquet d'entrée.

Définition 5.2. Un ensemble *HeaderV* de VVE associé à un point d'exécution du programme P4 est complet si pour tout paquet d'entrée son VVE, à ce point dans le programme, est présent dans *HeaderV*.

Lors de l'analyse symbolique, tous les sommets du GDA d'un programme P4 sont évalués. Aussi les sommets conditionnels sont évalués avec l'algorithme 5.2 dans lequel toutes les branches sont évaluées. Par conséquent, l'ensemble *HeaderV* est complet. Cependant lors de l'analyse d'un sommet conditionnel, la condition n'est pas évaluée. Il est donc possible que certaines branches soient évaluées avec des VVE invalides. Si tel est le cas, l'ensemble *HeaderV* contiendra certains VVE invalides. Par conséquent, le résultat de l'analyse symbolique ne garantit pas que l'ensemble *HeaderV* soit minimal.

5.3 Élagage du graphe de déparsage

Nous avons présenté dans la section précédente l'analyse symbolique d'un programme P4. Cette analyse génère un ensemble *HeaderV* de Vecteur de Validité d'En-tête (VVE) possible à certains points du programme P4. Dans cette section, nous présentons comment ce résultat peut être utilisé pour élaguer le graphe de déparsage. Dans un premier temps, nous présentons l'élagage du graphe de déparsage en faisant une évaluation partielle du déparseur en avec l'ensemble *HeaderV*. Puis nous démontrons que la solution proposée permet de générer des graphes de déparsage minimaux. Finalement, nous présentons les résultats d'implémentation du déparseur générés à partir d'un GDA de déparsage élagué.

5.3.1 L'évaluation partielle du graphe de déparsage

Algorithme pour l'élagage

L'algorithme 5.3 décrit l'élagage du graphe de déparsage. L'algorithme prend en entrée un ensemble de combinaisons d'en-tête possible *HeaderV* et le graphe de déparsage transitivement fermé G_c. L'algorithme retourne le graphe de déparsage élagué. L'élagage consiste à reconstruire un graphe G_o avec uniquement les chemins pouvant être suivis selon les combinaisons d'en-tête possible. Pour chaque combinaison d'en-tête VVE dans *HeaderV*, on cherche dans le graphe d'origine les arêtes valident. Pour chaque arête dont la condition est valide avec VVE, on insère l'arête dans le nouveau graphe G_o. Une fois le nouveau graphe construit, on retire la condition des arêtes qui sont uniques en sortie de noeud.

Exemple d'application de l'algorithme

Soit le graphe de déparsage transitivement fermé de la figure 4.3. Soit l'ensemble *HeaderV* présenté dans le Tableau 5.1. Dans la figure 5.4, les différents sous-graphes sont présentés. Le premier VVE analysé a uniquement l'en-tête `ethernet` qui est valide, les arêtes valident sont donc celles entre start et `ethernet` et entre `ethernet` et end. Cela résulte dans le graphe de la figure 5.4a.

Une fois le premier VVE utilisé, le second VVE est utilisé. Le second VVE a deux en-têtes valides : `ethernet` et `ipv4`. Les arêtes validant ces conditions dans le graphe de déparsage transitivement fermé sont celles entre start et `ethernet`, `ethernet` et `ipv4` et `ipv4` et end. Comme l'arête entre start et `ethernet` est déjà présente, elle n'est pas ajoutée, seules les deux autres arêtes sont ajoutées au GDA de sortie. Le GDA résultant est celui de la figure 5.4b.

Algorithme 5.3 : Retirer des arêtes du graphe de déparsage

input : G_c : Le graphe de déparsage transitivement fermé
input : *HeaderV* : L'ensemble de Vecteurs de Validité d'En-tête (VVE)
output : G_o : Le graphe élagué.

```
G_o ←new DAG() ;                                    // Création d'un GDA
Pour Chaque VVE dans HeaderV Faire
    Pour Chaque n dans G_c Faire
        // recherche de l'arête de sortie de n valide avec VVE
        eIt ← n.edges.iterator.begin ;
        while not evaluate(eIt.label, VVE) do eIt.next();
        G_o.InsertEdge(eIt) ;                       // Ajout de l'arête de sortie valide
    FinPourCh
FinPourCh
// Retire les étiquettes des arêtes uniques à la sortie d'un noeud
Pour Chaque n in G_o.nodes Faire
    Si n.outedges.size == 1 Alors n.outedges.removeLabels()
FinPourCh
return G_o
```

Le dernier VVE a les trois en-têtes `ethernet`, `ipv4` et `tcp` valides. L'analyse de ce VVE résulte dans l'ajout, au GDA de sortie, d'une arête entre `ipv4` et `tcp` et d'une arête entre `tcp` et end. Nous obtenons ainsi le graphe de la figure 5.4c.

Finalement, une fois l'ensemble des VVE analysé, nous traversons le GDA généré afin de retirer les conditions des arêtes uniques en sortie de sommet. Dans le cas du GDA de la figure 5.4c, cela revient à retirer la condition associée à l'arête entre start et `ethernet`. Cela résulte dans le graphe de la figure 5.2.

5.3.2 Le graphe de déparsage minimal

La section précédente présente l'algorithme 5.3 pour effectuer l'élagage du GDA de déparsage à partir des résultats de l'analyse symbolique. Dans cette section, nous montrons que l'algorithme proposé génère un graphe de déparsage minimal au sens de la définition 5.3.

Définition 5.3. Un Graphe Dirigé Acyclique (GDA) de déparsage minimal d'un programme P4 est un GDA ayant le nombre minimum de sommets et d'arêtes entre les sommets «start» et «end» qui permet de correctement reconstituer n'importe quel paquet traiter par le programme.

Lemme 5.1. *Si l'ensemble de Vecteur de Validité d'En-tête (VVE) en entrée contient seulement un VVE, alors le GDA de déparsage minimal possède seulement un chemin.*

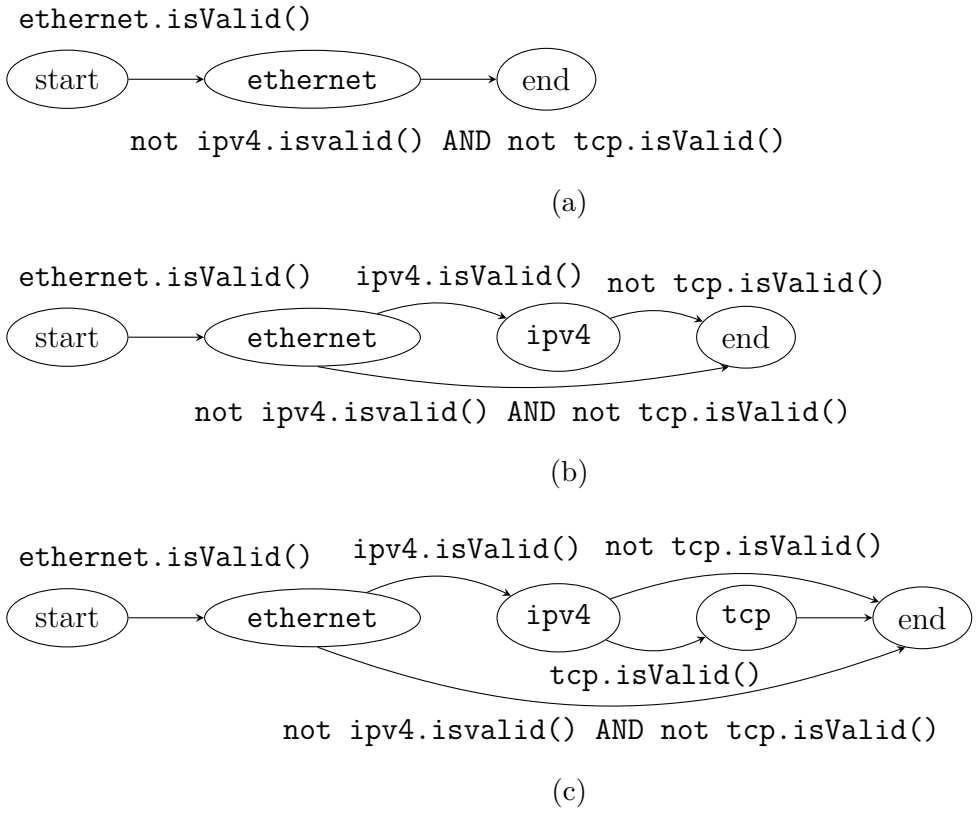


Figure 5.4 Les sous-graphes générés après l'analyse de chacun des VVE du tableau 5.1

Démonstration. Si le résultat de l'analyse symbolique d'un programme P4, est un ensemble avec seulement un VVE, alors une seule combinaison d'en-têtes est valide au moment du déparsage. Par conséquent la recomposition du paquet intégrera toujours les mêmes entêtes, et il y aura donc seulement un chemin valide dans le GDA de déparsage. □

Théorème 5.1. *L'algorithme 5.3 permet de générer un GDA de déparsage minimal d'un programme P4 en fonction des résultats de l'analyse symbolique du programme.*

Démonstration. Prenons le cas initial ou l'ensemble *HeaderV* contenant un seul VVE. Dans ce cas, d'après le lemme 5.1, le GDA est minimal s'il est composé d'un seul chemin. Appliquons l'algorithme 5.3 à cet ensemble en entrée. La première boucle « **Pour Chaque** VVE dans *HeaderV* **Faire** » aura donc une seule itération. Lors du parcours des sommets, seulement une arête valide pour le VVE est sélectionnée et ajoutée au GDA de sortie G_o. Par conséquent, chacun des sommets de G_o aura uniquement une seule arête de sortie. Il y aura donc un unique chemin entre les sommets «start» et «end» dans le GDA G_o, donc le GDA généré est minimal.

Prenons maintenant le cas où l'ensemble *HeaderV* contient n VVE. À la première itération lors du parcours de *HeaderV*, un graphe minimal est généré comme montré précédemment. Lors de l'itération suivante, un autre VVE est sélectionné. Lors de l'analyse, si l'arête *eIt* évaluée est déjà présente alors le GDA est inchangé, le graphe reste donc minimal. Si l'arête n'est pas présente alors elle est insérée dans le GDA. Le GDA G_o généré reste donc minimal, car seulement les arêtes requises pour ce VVE sont ajoutées. Ce processus est répété pour tous les VVE suivants. □

5.3.3 Résultats d'implémentation après élagage du graphe

Pour évaluer l'impact de l'optimisation proposée, les programmes **T0**, **T1**, **T2** et **T3** utilisés dans la section 4.4 ont été réutilisés et sont présentés dans le tableau 5.2. Le tableau présente, pour chaque programme, le nombre de sommets, le nombre de chemins entre les états «start» et «end» dans le graphe de déparsage avec et sans élagage, la largeur du PHV et la liste des en-têtes utilisés. On constate que, pour chacun des graphes de déparsage, l'élagage permet de réduire le nombre de chemins dans le graphe. On constate que ce gain est d'autant plus important que le nombre de chemins d'origine est élevé. Dans le cas de **T0** où il y a 8 chemins dans le graphe non élagué, il en reste 3 après élagage, soit une division par presque 3. Dans le cas de **T1**, on passe de 32 chemins à 7, soit une division par plus de 4. Dans le cas de **T2**, le nombre de chemins est divisé par 14 en passant de 128 à 9. Finalement, pour **T3**, le nombre de chemins dans le graphe de déparsage passe de 2048 à 81, soit une division par 25.

Tableau 5.2 Caractéristiques des cas de test pour l'implémentation du déparseur

Test	Nombre de sommets	Nombre de chemins		Largeur PHV (bits)	Liste des en-têtes
		Non élagué	Élagué		
T0	3	8	3	432	`ethernet`, `ipv4`, `tcp`
T1	5	32	7	816	**T0** + `ipv6`, `udp`
T2	7	128	9	880	**T1** + `icmp`, `icmpv6`
T3	11	2048	81	1008	**T2** + 2× `vlan`, 2× `mpls`

La figure 5.5 présente le graphe de parsage de T2. Les figures 5.6 et 5.7 présentent, respectivement, les graphes de déparsage non élagués et élagués pour le cas de T2. On constate que les graphes de parsage et de déparsage sont similaires. Cela était attendu étant donné que le code P4 utilisé ici ne modifie pas la validité des en-têtes en dehors du parseur. En comparant les graphes des figures 5.6 et 5.7, on constate la réduction du graphe. On constate notamment qu'en sortie du sommet «start», seule l'arête vers le sommet `ethernet` est gardée. Également, on constate qu'il ne reste plus que trois transitions possible en sortie du sommet `ethernet`

qui sont vers `ipv4`, `ipv6` et «end», alors qu'il y en avait sept dans le graphe d'origine. Pour ce qui est des sommets `ipv4` et `ipv6`, ils ont chacun quatre arêtes à la sortie de ces sommets. Finalement, les sommets `tcp`, `udp`, `icmp` et `icmp6` n'ont plus qu'une arrête vers «fin».

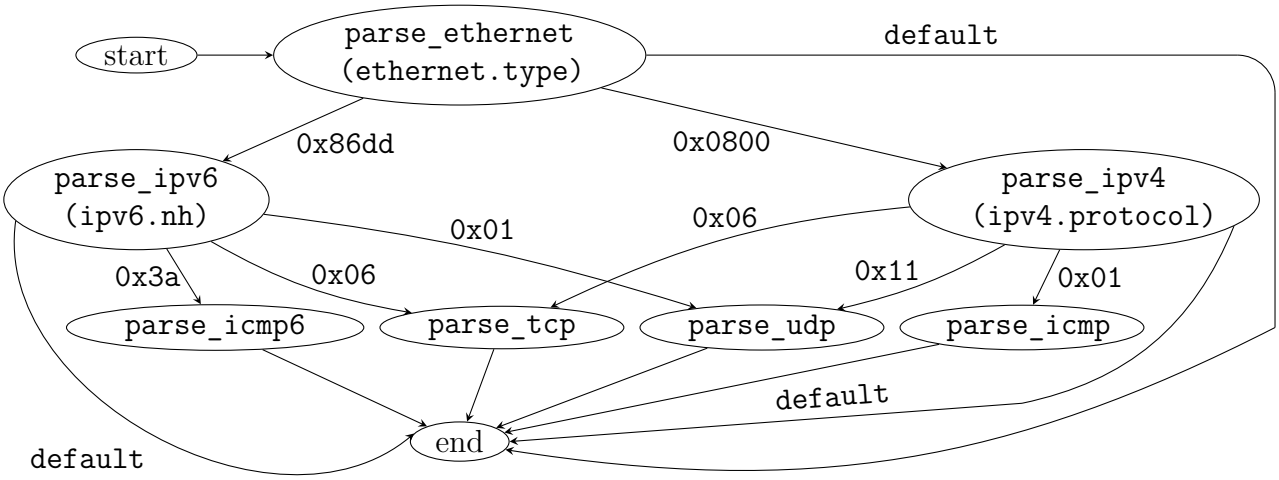


Figure 5.5 Graphe de parsage de **T2**

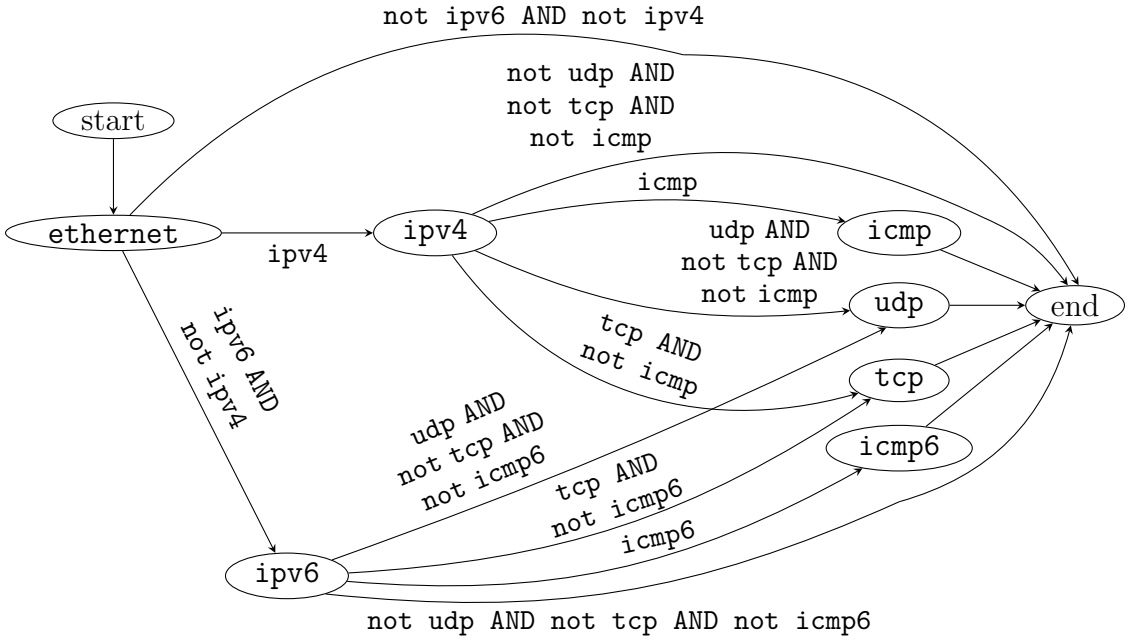


Figure 5.6 Graphe de déparsage élagué de **T2**

Les différents déparseurs ont également été implémentés sur un FPGA Xilinx Virtex Ultrascale+ *xcvu3p* avec des bus de sortie de 256 et 512 bits. Les résultats de l'implémentation sont présentés dans le tableau 5.3. On constate que l'élagage du graphe de déparsage permet

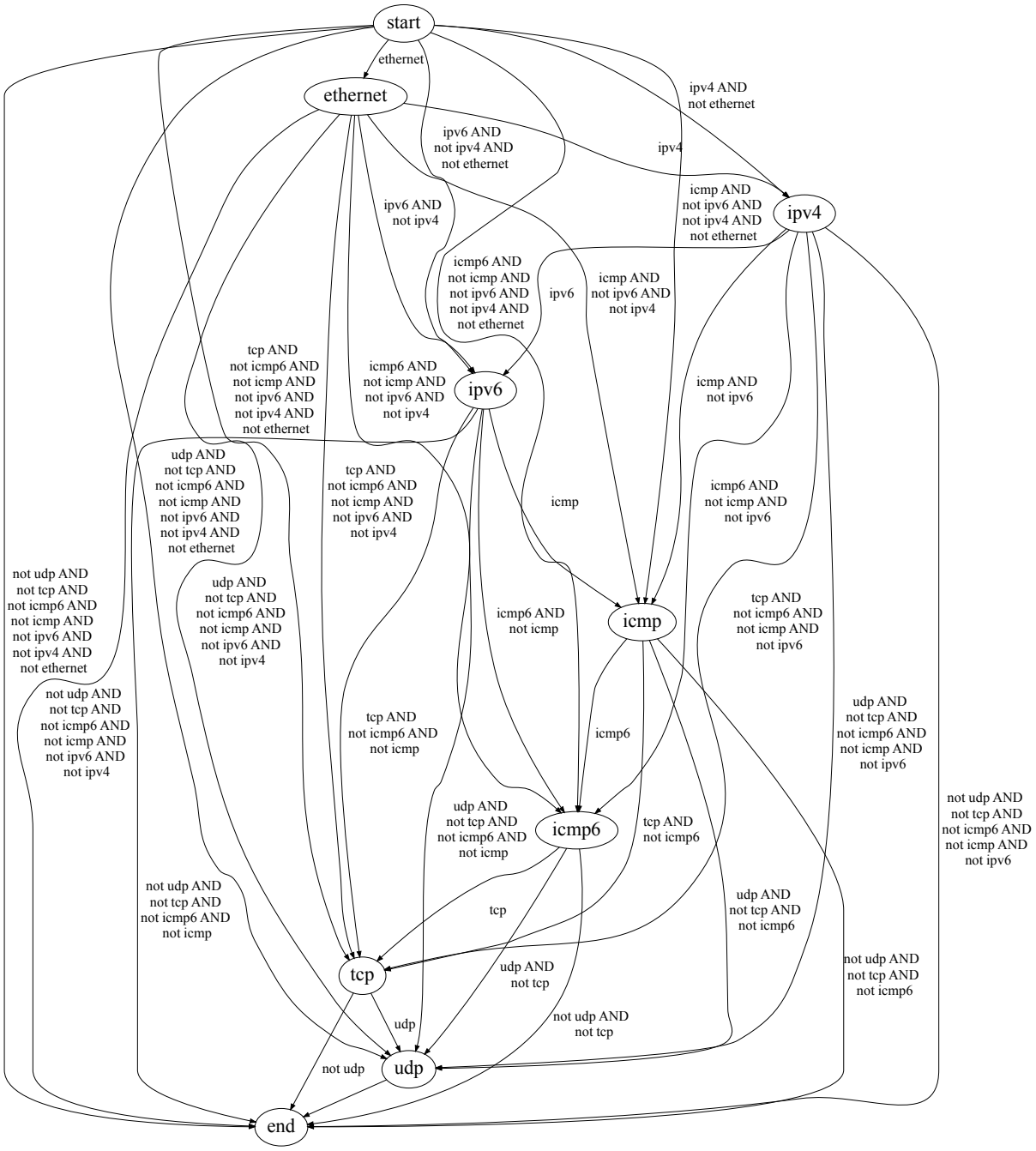


Figure 5.7 Graphe de déparsage non élagué de **T2**

une réduction de la consommation de ressources ainsi qu'une légère augmentation du débit de 4 % en moyenne. L'utilisation de LUT diminue en moyenne de 46 % entre les versions élaguées et non élaguées alors que l'utilisation de FF est réduite de 11 % en moyenne. Cela est cohérent avec les résultats de la section 4.4 où l'on montre que l'utilisation de FF dépend

principalement de la largeur du bus de sortie et l'utilisation de LUT dépend en grande partie du nombre de chemins dans le graphe de déparsage.

Tableau 5.3 Résultats après implémentation du déparseur avec et sans élagage sur un FPGA Xilinx Ultrascale+ *xcvu3p*

Test	Largeur du bus de sortie	Graphe non élagué			Graphe élagué			Gain élagage		
		LUT	FF	Débit	LUT	FF	Débit	LUT	FF	Débit
T0	256 bits	1562	1276	171 Gb/s	1500	1215	177 Gb/s	-4%	-5%	3%
	512 bits	7766	2545	320 Gb/s	3709	2422	343 Gb/s	-52%	-5%	7%
T1	256 bits	2678	1522	168 Gb/s	1557	1373	175 Gb/s	-41%	-10%	4%
	512 bits	8840	3002	318 Gb/s	5489	2690	322 Gb/s	-38%	-10%	1%
T2	256 bits	4660	1602	166 Gb/s	1514	1378	170 Gb/s	-67%	-14%	2%
	512 bits	10892	3197	291 Gb/s	5552	2693	320 Gb/s	-49%	-15%	10%
T3	256 bits	7401	1664	168 Gb/s	2657	1468	173 Gb/s	-64%	-12%	3%
	512 bits	14673	3369	313 Gb/s	7246	2865	321 Gb/s	-51%	-15%	3%
							Moyenne :	-46%	-11%	4%

Dans le cas de **T0**, avec le bus de sortie de 256 bits, on constate une diminution de LUT de 4 % entre la version élaguée et non élaguée en passant de 1 562 à 1 500, alors qu'avec le bus de sortie de 512 bits, on constate une baisse de 50 % du nombre de LUT qui passent de 2678 à 1557. Cette variation s'explique par le faible nombre de chemins et par la taille totale du PHV de 472 bits. En effet, dans le cas d'un bus de 512 bits, le bus de sortie peut intégrer tous les en-têtes. Lors de l'élagage du graphe de déparsage, la portion responsable de sélectionner les bits de PHV ne requiert plus de multiplexeur, ce qui simplifie le nombre total de signaux de contrôle requis, mais réduit aussi le nombre de multiplexeurs à implémenter. Dans le cas du bus de 256 bits, il n'est pas possible d'insérer tous les en-têtes sur le bus de sortie, donc des multiplexeurs sont toujours nécessaires après l'élagage du graphe. Toutefois, comme le graphe élagué contenait seulement huit chemins, les multiplexeurs déjà présents utilisaient déjà peu de LUT, une majorité en consommait déjà une seule. Par conséquent, le nombre de LUT utilisés, après élagage, a peu diminué pour le cas du bus de 256 bits.

Dans le cas de **T1**, le nombre de LUT diminue de 41 % et 38 % pour les bus de sortie de 256 et 512 bits. Dans les cas de **T2** et **T3** avec un bus de 256 bits, la consommation de LUT diminue de plus de 60%, et d'environ 50 % avec un bus de sortie de 512 bits. Ces résultats confirment que le nombre de chemins impacte significativement la consommation de LUT. Avec un bus de sortie de 256 bits, on remarque qu'avec le graphe de déparsage élagué **T0**, **T1**, et **T2** consomment presque la même quantité de ressources avec des débits similaires

de l'ordre de 170 Gb/s. Avec un bus de 512 bits, on constate que les cas **T1** et **T2** avec le graphe élagué consomment une quantité de ressources similaire avec un débit d'environ 320 Gb/s. Ces résultats montrent donc que l'élagage du graphe de déparsage permet une réduction significative de la consommation de ressources sur le FPGA.

CHAPITRE 6 CONCLUSION

Avec les réseaux programmables, les gestionnaires de centres de données peuvent adapter leurs réseaux à leurs besoins. Notamment, avec l'émergence des plans des données programmables, le déploiement de nouveaux protocoles réseau est grandement facilité sans nécessairement compromettre les débits de traitement. Aussi, comme les centres de données sont des environnements dynamiques, ils intègrent des FPGA comme accélérateurs reconfigurables. On constate donc une tendance grandissante pour l'implémentation de plans des données sur FPGA.

Dans cette thèse, nous nous sommes intéressés à l'utilisation des FPGA pour implémenter des plans des données réseau définis avec le langage P4. La première partie de cette thèse est consacrée à l'évaluation des FPGA comme plateformes pour l'implémentation d'applications écrites avec P4, et la seconde partie propose des solutions pour réduire le coût d'implémentation du module de déparsage sur FPGA. Dans ce chapitre, nous résumons l'ensemble de ces travaux. Tout d'abord, nous présentons une synthèse des travaux réalisés, puis nous discutons des limites de ces travaux et, finalement, nous proposons de futures avenues de recherche.

Synthèse des travaux

Dans cette thèse, nous avons exploré l'implémentation de l'architecture PISA sur FPGA de manière analytique et expérimentale. Ces résultats ont fait ressortir deux limites pour l'implémentation de PISA sur FPGA. La première limite est due à la microarchitecture actuelle des FPGA qui restreint l'implémentation de tables de comparaison TCAM et LPM, et qui diminue les débits maximums pouvant être atteints. Afin de pallier à cette limite, nous avons proposé des modifications à la microarchitecture des FPGA : l'ajout de TCAM dédiées et l'insertion de nouvelles structures de routage. La seconde limite était spécifiquement liée à l'implémentation du déparseur sur FPGA qui consommait beaucoup de ressources. En effet, le déparseur étant considéré comme l'inverse du parseur, peu de travaux ont traité de son implémentation. Cependant, on constate dans les travaux précédents que le déparseur utilise, au minimum, de 3× à 4× plus de ressources que le parseur. Pour remédier à cette limite, nous avons proposé des axes de travail, afin d'améliorer l'implémentation du déparseur sur les FPGA, qui ont servi de base dans le reste de cette thèse.

Nous avons proposé une transformation du graphe de déparsage d'un déparseur décrit dans le langage $P4_{16}$ afin de réintroduire le parallélisme qui était présent dans le graphe de déparsage d'un programme $P4_{14}$. En effet, en passant de $P4_{14}$ à $P4_{16}$, le parseur et le déparseur

deviennent deux éléments indépendants dans un programme P4. En $P4_{14}$ le graphe de déparsage dérive du graphe de parsage, et un sommet dans ce graphe peut avoir plusieurs arêtes de sortie, ce qui permet de paralléliser la sélection du sommet suivant. En $P4_{16}$, le déparseur est un processus séquentiel et chaque sommet ne contient qu'une arête de sortie, ce qui brise le parallélisme disponible en $P4_{14}$. Pour retrouver le parallélisme du graphe de déparsage en $P4_{16}$, nous avons proposé un algorithme basé sur le concept de fermeture transitive en ajoutant aux arêtes les conditions de transition. Cela nous permet de générer par la suite des machines à états dans lesquelles la sélection du prochain état se fait de manière parallèle.

Également, nous avons proposé une architecture de déparseur générée à partir du graphe de déparsage transformé. Cette architecture se base sur deux constats :

1. Le déparseur est une entité qui sérialise un vecteur de bits très large ; et
2. Sélectionner des bits de manière fixe sur FPGA est très peu coûteux, mais la sélection dynamique de larges vecteurs de bits est très coûteuse.

Comme le déparseur sérialise un vecteur de bits, il faut que l'architecture générée permette de sélectionner les bits à sortir pour chacun des bits du bus de sortie. Comme la sélection dynamique est coûteuse, il est nécessaire de minimiser le recours à celle-ci. Pour cela, l'architecture proposée connecte pour chaque bit de sortie uniquement les bits d'en-têtes qui peuvent y être insérés. Cette information est obtenue par le parcours du graphe de déparsage. Grâce à cette approche, nous arrivons à réduire de 4× à 10× l'utilisation des ressources du déparseur par rapport aux travaux antérieurs.

Finalement, nous avons proposé de réduire le nombre de chemins dans le graphe de déparsage en spécialisant le programme P4. En effet, nous avons expliqué que le déparseur de $P4_{16}$ ne bénéficie pas de la réduction du graphe disponible en $P4_{14}$. De plus, nous avons constaté que le nombre de chemins dans le graphe de déparsage impacte l'utilisation de ressources lors de l'implémentation de notre architecture de déparseur. Pour spécialiser le programme P4, nous avons d'abord présenté les algorithmes permettant de faire une analyse symbolique du programme pour générer l'ensemble des combinaisons d'en-têtes valides. Puis, nous avons présenté l'algorithme permettant d'élaguer le graphe de déparsage à partir des combinaisons d'en-têtes valides. Nous avons également montré que l'algorithme proposé permet de générer un graphe de déparsage minimal. L'élagage du graphe de déparsage résulte en une diminution moyenne de 46 % du coût d'implémentation de notre déparseur sur FPGA.

Les résultats présentés dans cette thèse montrent que les FPGA peuvent implémenter un large éventail d'applications réseau décrites avec le langage P4. Également, nous avons montré qu'il est possible de générer et d'implémenter un déparseur décrit avec P4 consommant une quantité de ressources limitées. Finalement, nous avons proposé de spécialiser le programme

P4 afin de réduire le nombre de chemins dans le graphe de déparsage et ainsi réduire le coût d'implémentation du déparseur.

Limites des travaux

Dans notre analyse de l'implémentation de PISA sur FPGA, nous étions limités à l'utilisation du seul compilateur P4 vers FPGA qui nous était accessible. Afin de nous extraire de cette contrainte, nous avons aussi considéré une analyse de l'état de l'art qui reste cependant principalement théorique. L'intégration du plan des données avec le plan de contrôle est un élément essentiel dans la construction d'applications réseau. Il s'agit d'un aspect que nous n'avons pas exploré, notamment en ce qui a trait aux tables de comparaisons.

La génération de l'architecture du déparseur requiert le parcours de tous les chemins du graphe de déparsage afin d'effectuer la construction des sous-graphes. Dans le cas de graphes de déparsage avec un nombre trop important de chemins, cela peut rendre la génération de l'architecture impossible. Cependant, grâce à la réduction du graphe de déparsage que nous avons proposée, la taille des graphes de déparsage générés est significativement réduite. Également, notre architecture ne supporte pas l'émission d'en-têtes de tailles variables. Ceci peut toutefois être outrepassé en ajoutant plusieurs en-têtes de taille fixe.

La plus importantes limitation pour la spécialisation des programmes est pour le cas de programmes P4 ayant un nombre important de combinaisons d'en-têtes. Nous avons présenté le fait que le temps de traitement est linéairement dépendant du nombre de combinaisons d'en-têtes, alors que le nombre total de combinaisons est exponentiel. Comme discuté précédemment, les protocoles réseau sont structurés par couches, donc ce cas est peu probable.

Avenues pour des travaux futurs

À la suite de ces travaux, nous avons constaté qu'il est difficile de tester de nouvelles architectures générées à partir de P4. Par exemple, dans le cas du déparseur, l'architecture que nous générons est différente pour chacun des programmes P4. Pour la tester, il faut donc générer un nombre important de cas de test. Ce problème se rencontre également dans les cas de tests d'autres modules comme les tables de comparaisons. Nous proposons deux axes complémentaires pour palier à cette difficulté : un compilateur P4 vers FPGA libre, et le développement d'une solution de simulation.

L'accès à un compilateur P4 pour FPGA libre permettrait de tester l'intégration de nouvelles architectures générées avec P4 plus facilement. Par exemple, lors de la conception du déparseur, notre capacité à tester l'intégration de celui-ci dans un pipeline PISA n'était

pas possible. Avec un compilateur P4 libre ciblant les FPGA, il sera plus facile de proposer de nouvelles architectures pour l'utilisation des FPGA comme plans des données programmables. Également, le développement d'un compilateur P4 pour FPGA pose la question de la représentation intermédiaire adaptée aux FPGA.

Dans le premier cas, le temps de simulation est très long, mais permet de déterminer exactement quelles sont les portions du circuit généré qui ne fonctionnent pas correctement. Dans le second cas, la simulation est très rapide, mais ne permet pas de déterminer la source d'un fonctionnement incorrect. Nous proposons d'ajouter une nouvelle approche de simulation qui intégrerait une simulation précise au cycle d'horloge, et la simulation à plus haut niveau des autres modules du programme. Par exemple, si l'on propose une nouvelle architecture de table de comparaison, il serait possible de la simuler avec une précision au cycle d'horloge et de tester le reste du programme à plus haut niveau.

Dans le cas plus spécifique du déparseur, nous voyons deux axes de recherche futurs. Le premier axe est d'explorer des solutions afin d'optimiser le processus de génération du déparseur que nous avons proposé. Le second axe de recherche porte sur la modification de l'expression du déparseur en $P4_{16}$. En effet, la description du déparseur dans la version actuelle de P4 limite fortement son expressivité. Notamment, la fonction `emit` définie dans $P4_{16}$ effectue un test sur le champ de validité. Pour s'assurer qu'un en-tête sera inséré, le programmeur est obligé de forcer le champ de validité à vrai. De plus, la vérification du champ de validité peut être décrite de manière explicite en $P4_{16}$. Nous proposons donc d'explorer les manières qui pourraient permettre d'extraire cette condition de la fonction externe tout en gardant la simplicité pour exprimer l'opération de déparsage.

Une des contraintes fortes pour un gestionnaire de centre de données est de faire évoluer ses équipements tout en gardant une gestion homogène de l'ensemble de l'infrastructure. Pour cela il faut que tous les composants possèdent des architectures suffisamment similaires et stables pour ne pas avoir à gérer de nombreuses bases de données d'applications. Par exemple, si un gestionnaire possède des serveurs avec des processeurs utilisant l'architecture x86, le fabricant ainsi que la génération du processeur affectent peu l'exécution d'une application sur le serveur. Cependant, dans le cas des FPGA, il n'existe pas d'architecture standard, et on constate même ces dernières années une divergence des architectures entre les FPGA Intel et Xilinx. Il risque donc de devenir de plus en plus difficile de développer des applications qui pourront être portées sur ces différentes plateformes. Un axe de recherche futur prometteur serait le développement d'architectures pour les FPGA qui seraient l'équivalent des jeux d'instructions pour les processeurs à usage général, c'est-à-dire avec une interface standard stable dans le temps et indépendante de la microarchitecture.

RÉFÉRENCES

[1] R. H. Zakon, "Hobbes' Internet Timeline," RFC 2235, nov. 1997. [En ligne]. Disponible : https://www.rfc-editor.org/info/rfc2235

[2] U. Cisco, "Cisco annual internet report (2018–2023) white paper," *Cisco : San Jose, CA, USA*, 2020. [En ligne]. Disponible : https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf

[3] R. Giladi, *Network processors : architecture, programming, and implementation*, ser. Morgan Kaufmann Series in Systems on Silicon. Burlington, MA : Elsevier, 2008.

[4] G. Papastergiou *et al.*, "De-ossifying the internet transport layer : A survey and future perspectives," *IEEE Communications Surveys Tutorials*, vol. 19, n°. 1, p. 619–639, Firstquarter 2017.

[5] O. N. Foundation. (2022) Sdn technical specification. [En ligne]. Disponible : https://www.opennetworking.org/software-defined-standards/specifications/

[6] N. McKeown *et al.*, "Openflow : Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, n°. 2, p. 69–74, mars 2008. [En ligne]. Disponible : http://doi.acm.org/10.1145/1355734.1355746

[7] P. Bosshart *et al.*, "Forwarding metamorphosis : Fast programmable match-action processing in hardware for SDN," dans *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA : Association for Computing Machinery, 2013, p. 99–110. [En ligne]. Disponible : https://doi.org/10.1145/2486001.2486011

[8] ——, "P4 : Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, n°. 3, p. 87–95, juill. 2014. [En ligne]. Disponible : http://doi.acm.org/10.1145/2656877.2656890

[9] A. Agrawal et C. Kim, "Intel tofino2 – a 12.9 Tbps P4-programmable ethernet switch," dans *2020 IEEE Hot Chips 32 Symposium (HCS)*, 2020, p. 1–32.

[10] X. Chen, "Implementing AES encryption on programmable switches via scrambled lookup tables," dans *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, ser. SPIN '20. New York, NY, USA : Association for Computing Machinery, 2020, p. 8–14. [En ligne]. Disponible : https://doi.org/10.1145/3405669.3405819

[11] A. Putnam *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” dans *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, juin 2014, p. 13–24.

[12] X. Wang *et al.*, “When FPGA meets cloud : A first look at performance,” *IEEE Transactions on Cloud Computing*, p. 1–1, 2020.

[13] A. Andreyev *et al.*, “Reinventing facebook’s data center network,” 2019. [En ligne]. Disponible : https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/

[14] D. Firestone *et al.*, “Azure accelerated networking : Smartnics in the public cloud,” dans *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA : USENIX Association, 2018, p. 51–66. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi18/presentation/firestone

[15] C. Kulkarni *et al.*, “Mapping a domain specific language to a platform FPGA,” dans *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC ’04. New York, NY, USA : Association for Computing Machinery, 2004, p. 924–927. [En ligne]. Disponible : https://doi.org/10.1145/996566.996811

[16] H. Wang *et al.*, “P4FPGA : A rapid prototyping framework for P4,” dans *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17. New York, NY, USA : Association for Computing Machinery, 2017, p. 122–135. [En ligne]. Disponible : https://doi.org/10.1145/3050220.3050234

[17] P. Benáček *et al.*, “P4-to-VHDL : Automatic generation of high-speed input and output network blocks,” *Microprocessors and Microsystems*, vol. 56, p. 22 – 33, 2018. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0141933117304787

[18] J. Varia et S. Mathew, “Overview of amazon web services,” 2014. [En ligne]. Disponible : https://media.amazonwebservices.com/AWS_Overview.pdf

[19] N. Tarafdar *et al.*, “Designing for FPGAs in the cloud,” *IEEE Design Test*, vol. 35, n°. 1, p. 23–29, Feb 2018.

[20] B. Stephens *et al.*, “Your programmable NIC should be a programmable switch,” dans *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’18. New York, NY, USA : Association for Computing Machinery, 2018, p. 36–42. [En ligne]. Disponible : https://doi.org/10.1145/3286062.3286068

[21] A. Kaufmann *et al.*, “High performance packet processing with flexnic,” *SIGPLAN Not.*, vol. 51, n°. 4, p. 67–81, mar 2016. [En ligne]. Disponible : https://doi.org/10.1145/2954679.2872367

[22] B. Buhrow *et al.*, “A highly parallel AES-GCM core for authenticated encryption of 400 Gb/s network protocols,” dans *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, p. 1–7.

[23] Y. Tokusashi *et al.*, “The case for in-network computing on demand,” dans *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys ’19. New York, NY, USA : Association for Computing Machinery, 2019. [En ligne]. Disponible : https://doi.org/10.1145/3302424.3303979

[24] R. A. Cooke et S. A. Fahmy, “Quantifying the latency benefits of near-edge and in-network FPGA acceleration,” dans *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys ’20. New York, NY, USA : Association for Computing Machinery, 2020, p. 7–12. [En ligne]. Disponible : https://doi.org/10.1145/3378679.3394534

[25] S. Ibanez *et al.*, “The P4->NetFPGA workflow for line-rate packet processing,” dans *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 1–9. [En ligne]. Disponible : https://doi.org/10.1145/3289602.3293924

[26] Z. Cao *et al.*, “A fast approach for generating efficient parsers on FPGAs,” *Symmetry*, vol. 11, n°. 10, 2019, information sur le cout d’un shifter. [En ligne]. Disponible : https://www.mdpi.com/2073-8994/11/10/1265

[27] J. Cabal *et al.*, “Configurable FPGA packet parser for terabit networks with guaranteed wire-speed throughput,” dans *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’18. New York, NY, USA : Association for Computing Machinery, 2018, p. 249–258. [En ligne]. Disponible : https://doi.org/10.1145/3174243.3174250

[28] J. Santiago da Silva *et al.*, “P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs,” dans *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’18. New York, NY, USA : Association for Computing Machinery, 2018, p. 147–152. [En ligne]. Disponible : https://doi.org/10.1145/3174243.3174270

[29] V. Puš *et al.*, “Low-latency modular packet header parser for FPGA,” dans *2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, oct. 2012, p. 77–78.

[30] M. Attig et G. Brebner, “400 Gb/s programmable packet parsing on a single FPGA,” dans *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, oct. 2011, p. 12–23.

[31] N. Feamster *et al.*, “The road to SDN : An intellectual history of programmable networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, n°. 2, p. 87–98, avr. 2014. [En ligne]. Disponible : https://doi.org/10.1145/2602204.2602219

[32] D. Kreutz *et al.*, “Software-defined networking : A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, n°. 1, p. 14–76, janv. 2015.

[33] M. Casado *et al.*, “Ethane : Taking control of the enterprise,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, n°. 4, p. 1–12, aug 2007. [En ligne]. Disponible : https://doi.org/10.1145/1282427.1282382

[34] T. Koponen *et al.*, “Onix : A distributed control platform for large-scale production networks,” dans *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC : USENIX Association, oct. 2010. [En ligne]. Disponible : https://www.usenix.org/conference/osdi10/onix-distributed-control-platform-large-scale-production-networks

[35] N. Gude *et al.*, “Nox : Towards an operating system for networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, n°. 3, p. 105–110, jul 2008. [En ligne]. Disponible : https://doi.org/10.1145/1384609.1384625

[36] Open Networking Foundation (ONF), “Openflow switch specification,” 2015. [En ligne]. Disponible : https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[37] D. Comer, “Network processors : Programmable technology for building network systems,” *The Internet Protocol Journal*, vol. 7, n°. 4, p. 2–12, 2004. [En ligne]. Disponible : https://ipj.dreamhosters.com/wp-content/uploads/issues/2004/ipj07-4.pdf

[38] R. Giladi, *Network processors : architecture, programming, and implementation*, ser. Morgan Kaufmann Series in Systems on Silicon. Burlington, MA : Elsevier, 2008.

[39] C. Kulkarni *et al.*, “Programming challenges in network processor deployment,” dans *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES ’03. New York, NY, USA : Association for Computing Machinery, 2003, p. 178–187. [En ligne]. Disponible : https://doi.org/10.1145/951710.951735

[40] G. Gibb *et al.*, “Design principles for packet parsers,” dans *Architectures for Networking and Communications Systems*, oct. 2013, p. 13–24.

[41] M. Mernik *et al.*, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, nº. 4, p. 316–344, déc. 2005. [En ligne]. Disponible : https://doi.org/10.1145/1118890.1118892

[42] E. Kohler *et al.*, “The click modular router,” *ACM Trans. Comput. Syst.*, vol. 18, nº. 3, p. 263–297, août 2000. [En ligne]. Disponible : https://doi.org/10.1145/354871.354874

[43] N. Shah *et al.*, “Np-click : a productive software development approach for network processors,” *IEEE Micro*, vol. 24, nº. 5, p. 45–54, sept. 2004.

[44] N. Bonelli *et al.*, “A purely functional approach to packet processing,” dans *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS ’14. New York, NY, USA : Association for Computing Machinery, 2014, p. 219–230. [En ligne]. Disponible : https://doi.org/10.1145/2658260.2658269

[45] R. Duncan et P. Jungck, “packetc language for high performance packet processing,” dans *2009 11th IEEE International Conference on High Performance Computing and Communications*, juin 2009, p. 450–457.

[46] H. Song, “Protocol-oblivious forwarding : Unleash the power of SDN through a future-proof forwarding plane,” dans *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13. New York, NY, USA : Association for Computing Machinery, 2013, p. 127–132. [En ligne]. Disponible : https://doi.org/10.1145/2491185.2491190

[47] G. Brebner et W. Jiang, “High-speed packet processing using reconfigurable computing,” *IEEE Micro*, vol. 34, nº. 1, p. 8–18, janv. 2014.

[48] M. Budiu et C. Dodd, “The $P4_{16}$ programming language,” *SIGOPS Oper. Syst. Rev.*, vol. 51, nº. 1, p. 5–14, sept. 2017. [En ligne]. Disponible : http://doi.acm.org/10.1145/3139645.3139648

[49] T. P. A. W. Group, “$P4_{16}$ portable switch architecture (psa),” Rapport technique, nov. 2018. [En ligne]. Disponible : https://p4.org/p4-spec/docs/PSA-v1.1.0.pdf

[50] G. Brebner, “Extending the range of P4 programmability,” dans *Keynote in the First European P4 workshop (P4EU)*, 2018. [En ligne]. Disponible : https://opennetworking.org/wp-content/uploads/2020/12/Gordon_Brebner.pdf

[51] Intel, “$P4_{16}$ intel® tofino™ native architecture – public version,” mars 2021. [En ligne]. Disponible : https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch-Document.pdf

[52] F. Hauser *et al.*, “A survey on data plane programming with P4 : Fundamentals, advances, and applied research,” 2021, long survey.

[53] Xilinx®, “SDnet Development Environment,” 2018. [En ligne]. Disponible : https://www.xilinx.com/products/design-tools/software-zone/sdnet.html

[54] M. Budiu, “$P4_{16}$ reference compiler implementation architecture,” juin 2021. [En ligne]. Disponible : https://github.com/p4lang/p4c/blob/8fc09b7b2b51c83922fda355b6b61f8972499f7d/docs/compiler-design.pdf

[55] “$P4_{16}$ reference compiler,” 2022. [En ligne]. Disponible : https://github.com/p4lang/p4c

[56] P. L. Consortiumb, “Behavioral-model : The reference P4 software switch,” 2022. [En ligne]. Disponible : https://github.com/p4lang/behavioral-model

[57] “p4c-ebpf,” 2022. [En ligne]. Disponible : https://github.com/p4lang/p4c/tree/main/backends/ebpf

[58] “p4c-ubpf,” 2022. [En ligne]. Disponible : https://github.com/p4lang/p4c/tree/main/backends/ubpf

[59] “p4c-xdp,” 2022. [En ligne]. Disponible : https://github.com/vmware/p4c-xdp

[60] “What is ebpf ? an introduction and deep dive into the EBPF technology,” 2021. [En ligne]. Disponible : https://ebpf.io/what-is-ebpf/

[61] M. Shahbaz *et al.*, “PISCES : A programmable, protocol-independent software switch,” dans *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16. New York, NY, USA : Association for Computing Machinery, 2016, p. 525–538. [En ligne]. Disponible : https://doi.org/10.1145/2934872.2934886

[62] P. Zanna *et al.*, “A method for comparing openflow and p4,” dans *2019 29th International Telecommunication Networks and Applications Conference (ITNAC)*, nov. 2019, p. 1–3.

[63] N. Networks, “Northbound networks.” [En ligne]. Disponible : https://northboundnetworks.com/

[64] P. Vörös *et al.*, “T4p4s : A target-independent compiler for protocol-independent packet processors,” dans *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, juin 2018, p. 1–8.

[65] “Explore the power of intel® intelligent fabric processors,” 2022. [En ligne]. Disponible : https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[66] A. Seibulescu et M. Baldi, *Leveraging P4 Flexibility to Expose Target-Specific Features*. New York, NY, USA : Association for Computing Machinery, 2020, p. 36–42. [En ligne]. Disponible : https://doi.org/10.1145/3426744.3431326

[67] “Pensado platform,” 2022. [En ligne]. Disponible : https://pensando.io/platform/

[68] Netronome Systems, Inc., “Programming netronome agilio® smartnics,” Netronome, Rapport technique, 2017. [En ligne]. Disponible : https://www.netronome.com/media/redactor_files/WP_NFP_Programming_Model.pdf

[69] M. Kekely et J. Korenek, “Mapping of P4 match action tables to FPGA,” dans *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, sept. 2017, p. 1–2.

[70] T. Stimpfling, “Mémoires associatives algorithmiques pou l’opération de recherche du plus long préfixe sur FPGA,” Thèse de doctorat, Polytechnique Montréal, 2020.

[71] S. Pontarelli *et al.*, “Smashing SDN "built-in" actions : Programmable data plane packet manipulation in hardware,” dans *2017 IEEE Conference on Network Softwarization (NetSoft)*, juill. 2017, p. 1–9.

[72] J. Cabal *et al.*, “Scalable P4 deparser for speeds over 100 Gbps,” dans *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, avr. 2019, p. 323–323.

[73] L. Jose *et al.*, “Compiling packet programs to reconfigurable switches,” dans *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA : USENIX Association, mai 2015, p. 103–115. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose

[74] S. Chole *et al.*, “Drmt : Disaggregated programmable switching,” 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, 2017, p. 1–14. [En ligne]. Disponible : https://doi.org/10.1145/3098822.3098823

[75] B. Vass *et al.*, *Compiling Packet Programs to Reconfigurable Switches : Theory and Algorithms*. New York, NY, USA : Association for Computing Machinery, 2020, p. 28–35. [En ligne]. Disponible : https://doi.org/10.1145/3426744.3431332

[76] T. K. Dangeti *et al.*, “P4LLVM : An LLVM Based P4 compiler,” dans *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, sept. 2018, p. 424–429.

[77] Z. Ma *et al.*, “Cachep4 : A behavior-level caching mechanism for p4,” dans *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos ’17. New York, NY, USA : Association for Computing Machinery, 2017, p. 108–110. [En ligne]. Disponible : https://doi.org/10.1145/3123878.3132003

[78] A. Abhashkumar *et al.*, “P5 : Policy-driven optimization of P4 pipeline,” dans *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17. New York, NY,

USA : Association for Computing Machinery, 2017, p. 136–142. [En ligne]. Disponible : https://doi.org/10.1145/3050220.3050235

[79] P. Wintermeyer *et al.*, "P2GO : P4 profile-guided optimizations," dans *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets '20. New York, NY, USA : Association for Computing Machinery, 2020, p. 146–152. [En ligne]. Disponible : https://doi.org/10.1145/3422604.3425941

[80] T. Luinaud *et al.*, "Bridging the gap : FPGAs as programmable switches," dans *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)*, mai 2020, p. 1–7.

[81] Xilinx®, "7 series FPGAs configurable logic block," Rapport technique, 2016. [En ligne]. Disponible : https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[82] H. Wong *et al.*, "Quantifying the gap between FPGA and custom CMOS to aid microarchitectural design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, nº. 10, p. 2067–2080, oct. 2014.

[83] A. M. Abdelhadi et G. G. Lemieux, "Modular SRAM-based binary content-addressable memories," dans *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, mai 2015, p. 207–214.

[84] R. Pagh et F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, nº. 2, p. 122–144, 2004. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0196677403001925

[85] A. Kirsch et M. Mitzenmacher, "The power of one move : Hashing schemes for hardware," *IEEE/ACM Transactions on Networking*, vol. 18, nº. 6, p. 1752–1765, déc. 2010.

[86] R. Dobai *et al.*, "Adaptive development of hash functions in FPGA-based network routers," dans *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, déc. 2016, p. 1–8.

[87] W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs," dans *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, 2013.

[88] "Ug573 : Xilinx ultrascale architecture memory resources user guide," 2019. [En ligne]. Disponible : https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

[89] "Ug574 : Xilinx ultrascale architecture configurable logic block user guide," 2017. [En ligne]. Disponible : https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[90] P. Reviriego *et al.*, “PR-TCAM : Efficient TCAM emulation on xilinx FPGAs using partial reconfiguration,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, n°. 8, p. 1952–1956, août 2019.

[91] “PG191 : Longest prefix match (LPM) search IP for SDNet - SmartCORE IP product guide,” 2017.

[92] H. Le et V. K. Prasanna, “Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning,” *IEEE Transactions on Computers*, vol. 61, n°. 7, p. 1026–1039, juill. 2012.

[93] M. Bando et H. J. Chao, “Flashtrie : Hash-based prefix-compressed trie for IP route lookup beyond 100Gbps,” dans *2010 Proceedings IEEE INFOCOM*, mars 2010, p. 1–9.

[94] G. Rétvári *et al.*, “Compressing IP forwarding tables : Towards entropy bounds and beyond,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, n°. 4, p. 111–122, août 2013. [En ligne]. Disponible : https://doi.org/10.1145/2534169.2486009

[95] S. Yazdanshenas et V. Betz, “Quantifying and mitigating the costs of FPGA virtualization,” dans *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, sept. 2017, p. 1–7.

[96] L. Kekely *et al.*, “Multi buses : Theory and practical considerations of data bus width scaling in FPGAs,” dans *2020 23rd Euromicro Conference on Digital System Design (DSD)*, août 2020, p. 49–56.

[97] P. H. W. Leong, “Recent trends in FPGA architectures and applications,” dans *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*, janv. 2008, p. 137–141.

[98] Lattice, “AN8071 : Content Addressable Memory (CAM) Applications for ispXPLD Devices,” 2002. [En ligne]. Disponible : https://www.latticesemi.com/~/media/LatticeSemi/Documents/ApplicationNotes/AD/ContentAddressableMemoryCAMApplicationsforispXPLDDevices.PDF?document_id=5597

[99] S. Matsunaga *et al.*, “Fully parallel 6T-2MTJ nonvolatile TCAM with single-transistor-based self match-line discharge control,” dans *2011 Symposium on VLSI Circuits - Digest of Technical Papers*, juin 2011, p. 298–299.

[100] Y. Yang *et al.*, “Accelerating RRT motion planning using TCAM,” dans *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI ’20. New York, NY, USA : Association for Computing Machinery, 2020, p. 481–486. [En ligne]. Disponible : https://doi.org/10.1145/3386263.3406948

[101] W. J. Dally et B. Towles, “Route packets, not wires : On-chip inteconnection networks,” dans *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC ’01. New York, NY, USA : Association for Computing Machinery, 2001, p. 684–689. [En ligne]. Disponible : https://doi.org/10.1145/378239.379048

[102] Achronix, “Revolutionary new 2d network-on-chip,” 2022. [En ligne]. Disponible : https://www.achronix.com/revolutionary-new-2d-network-chip

[103] B. Gaide *et al.*, “Xilinx adaptive compute acceleration platform : Versal™ architecture,” dans *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 84–93. [En ligne]. Disponible : https://doi.org/10.1145/3289602.3293906

[104] A. Ye et J. Rose, “Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits,” dans *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’05. New York, NY, USA : Association for Computing Machinery, 2005, p. 3–13. [En ligne]. Disponible : https://doi.org/10.1145/1046192.1046194

[105] Wikipédia, “DDR4 SDRAM — wikipédia, l’encyclopédie libre,” 2021, [En ligne ; Page disponible le 7-janvier-2021]. [En ligne]. Disponible : http://fr.wikipedia.org/w/index.php?title=DDR4_SDRAM&oldid=178536504

[106] T. Luinaud *et al.*, “Design principles for packet deparsers on FPGAs,” dans *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’21. New York, NY, USA : Association for Computing Machinery, 2021, p. 280–286. [En ligne]. Disponible : https://doi.org/10.1145/3431920.3439303

[107] T. P. L. Consortium, *$P4_{16}$ Language Specification*, P4.org, 2021. [En ligne]. Disponible : https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf

[108] P. Purdom, “A transitive closure algorithm,” *BIT Numerical Mathematics*, vol. 10, n°. 1, p. 76–94, 1970.

[109] T. Luinaud *et al.*, “Network data plane program specialization using symbolic analysis,” *ACM Trans. Archit. Code Optim.*, 2022.

[110] L. Burkholder, “The halting problem,” *SIGACT News*, vol. 18, n°. 3, p. 48–60, avr. 1987. [En ligne]. Disponible : https://doi.org/10.1145/24658.24665

[111] S. Kodeswaran *et al.*, “Tracking P4 program execution in the data plane,” dans *Proceedings of the Symposium on SDN Research*, ser. SOSR ’20. New York, NY, USA : Association for Computing Machinery, 2020, p. 117–122. [En ligne]. Disponible : https://doi.org/10.1145/3373360.3380843

[112] R. T. Braden, “Requirements for Internet Hosts - Communication Layers,” RFC 1122, oct. 1989. [En ligne]. Disponible : https://www.rfc-editor.org/info/rfc1122