POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Mémoires associatives algorithmiques pour l'opération de recherche du plus long préfixe sur FPGA

THIBAUT STIMPFLING
Département de génie électrique

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

Avril 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Mémoires associatives algorithmiques pour l'opération de recherche du plus long préfixe sur FPGA

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

Richard GOURDEAU, président
Yvon SAVARIA, membre et directeur de recherche
Pierre LANGLOIS, membre et codirecteur de recherche
Alejandro QUINTERO, membre
Bertrand GRANADO, membre externe

DÉDICACE

À ma famille, à mes ami.e.s, à mon amoureuse et aux victimes du féminicide de Polytechnique.

REMERCIEMENTS

En premier lieu, je tiens à sincèrement remercier mon directeur, Yvon Savaria, et mon codirecteur, Pierre Langlois, sans qui cette thèse n'aurait pu avoir eu lieu. Par ailleurs, je tiens à les remercier pour l'encadrement reçu, leurs conseils bienveillants, et leur soutien durant toute la durée de ce doctorat. Yvon ma notamment à de nombreuses reprises expliqué pourquoi il est nécessaire de persévérer dans le monde de la recherche. Pierre m'a quant à lui appris le sens de la phrase précise et concise.

Merci au groupe de recherche d'Ericsson Montréal, notamment Denis Monette, André Béliveau et Daniel Migault, de m'avoir intégré dans leur équipe. J'aimerais aussi adresser un merci tout particulier à Kaloom, spécifiquement à Laurent Marchand et Ludovic Béliveau, pour nous avoir entièrement fait confiance, mais aussi d'avoir mis à notre disposition un environnement stimulant combinant développement et recherche.

Je veux aussi exprimier toute ma gratitude à Normand Bélanger, avec qui j'ai eu d'innombrables discussions, autant techniques que personnelles, et qui a grandement contribué à la qualité de mon encadrement et à mon épanouissement. Par ailleurs, je souhaite remercier François-Raymond Boyer, le ninja du C++, qui m'a apporté une grande aide technique en ingénierie logicielle, ainsi qu'en algorithmie. Je tiens aussi à remercier Marie-Yannick Laplante, Nathalie Lévesque et Louise Clément pour leur aide dans l'ensemble des démarches administratives.

Je tiens aussi naturellement à remercier mes collègues de laboratoire qui ont rendu la réalisation de ce doctorat beaucoup plus agréable, me soutenant dans les bons moments, mais aussi dans les moments d'échecs et de lassitude. Merci à Ahmed, Bachir, Charle, Imad, Erika, Mickaël, Patrick et Vincent. Un merci tout particulier à Jeferson et Thomas, les deux mousquetaires, avec lesquels nous avons eu tellement de discussions sur tout et rien. Je suis très fier de l'équipe que nous avons réussi à former ensemble. Je vous considère avant tous comme mes amis.

Merci sincèrement à ma famille pour son soutien indéfectible. Bien qu'une grande distance géographique nous sépare, j'ai toujours ressenti de sa part beaucoup d'encouragements et d'amour. Accessoirement, les paquets postaux, contenant divers mets fins, m'ont toujours apporté un grand bonheur !

Merci à mes ami.e.s : Casper, Claudine, Clément, Constant, Élie, Fauve, Guillaume, Laurence, Mathieu, Maxime, Philippe, Florence L., Florence T., Jérôme, Valérie, qui m'ont apporté une

aide — directe ou indirecte —, et qui m'ont permis de me changer les idées durant ce doctorat. Pardon à ceux ou celles que j'aurai oubliés dans cette liste non exhaustive.

Finalement, je conclus cette section de remerciements avec la «cerise sur le sundae», Alice. J'aimerais t'adresser un remerciement tout spécial et te demander pardon pour les sacrifices que je t'ai imposés durant ce doctorat. Ton soutien est non quantifiable ; tu as été tout le temps là, dans tout ce que cela implique. Tu as su m'écouter, m'encourager et avoir les bons mots pour me donner l'énergie permettant de continuer d'avancer.

RÉSUMÉ

Les réseaux prédiffusés programmables — en anglais *Field Programmable Gate Array*s (FPGAs) — sont omniprésents dans les centres de données, pour accélérer des tâches d'indexations et d'apprentissage machine, mais aussi plus récemment, pour accélérer des opérations réseaux. Dans cette thèse, nous nous intéressons à l'opération de recherche du plus long préfixe — en anglais *Longest Prefix Match* (LPM) — sur FPGA. Cette opération est utilisée soit pour router des paquets, soit comme un bloc de base dans un plan de données programmable. Bien que l'opération LPM soit primordiale dans un réseau, celle-ci souffre d'inefficacité sur FPGA. Dans cette thèse, nous démontrons que la performance de l'opération LPM sur FPGA peut être substantiellement améliorée en utilisant une approche algorithmique, où l'opération LPM est implémentée à l'aide d'une structure de données. Par ailleurs, les résultats présentés permettent de réfléchir à une question plus large : *est-ce que l'architecture des FPGA devrait être spécialisée pour les applications réseaux ?*

Premièrement, pour l'application de routage IPv6 dans le réseau Internet, nous présentons SHIP. Cette solution exploite les caractéristiques des préfixes pour construire une structure de données compacte, pouvant être implémentée de manière efficace sur FPGA. SHIP utilise l'approche «diviser pour régner»pour séparer les préfixes en groupes de faible cardinalité et ayant des caractéristiques similaires. Les préfixes contenus dans chaque groupe sont ensuite encodés dans une structure de données hybride, où l'encodage des préfixes est adapté suivant leurs caractéristiques. Sur FPGA, SHIP augmente l'efficacité de l'opération LPM comparativement à l'état de l'art, tout en supportant un débit supérieur à 100 Gb/s.

Deuxièment, nous présentons comment implémenter efficacement l'opération LPM pour un plan de données programmable sur FPGA. Dans ce cas, contrairement au routage de paquets, aucune connaissance à priori des préfixes ne peut être utilisée. Par conséquent, nous présentons un cadre de travail comprenant une structure de données efficace, indépendamment des caractéristiques des préfixes contenus, et des méthodes permettant d'implémenter efficacement la structure de données sur FPGA. Un arbre B, étendu pour l'opération LPM, est utilisé en raison de sa faible complexité algorithmique. Nous présentons une méthode pour allouer à la compilation le minimum de ressources requis par l'abre B pour encoder un ensemble de préfixes, indépendamment de leurs caractéristiques. Plusieurs méthodes sont ensuite présentées pour augmenter l'efficacité mémoire après implémentation de la structure de données sur FPGA. Évaluée sur plusieurs scénarios, cette solution est capable de traiter plus de 100 Gb/s, tout en améliorant la performance par rapport à l'état de l'art.

ABSTRACT

FPGAs are becoming ubiquitous in data centers. First introduced to accelerate indexing services and machine learning tasks, FPGAs are now also used to accelerate networking operations, including the LPM operation. This operation is used for packet routing and as a building block in programmable data planes. However, for the two uses cases considered, the LPM operation is inefficiently implemented in FPGAs. In this thesis, we demonstrate that the performance of LPM operation can be significantly improved using an algorithmic approach, where the LPM operation is implemented using a data structure. In addition, using the results presented in this thesis, we can answer a broader question: *Should the FPGA architecture be specialized for networking?*

First, we present the SHIP data structure that is tailored to routing IPv6 packets in the Internet network. SHIP exploits the prefix characteristics to build a compact data structure that can be efficiently mapped to FPGAs. First, SHIP uses a "divide and conquer" approach to bin prefixes in groups with a small cardinality and sharing similar characteristics. Second, a hybrid-trie-tree data structure is used to encode the prefixes held in each group. The hybrid data structure adapts the prefix encoding method to their characteristics. Then, we demonstrated that SHIP can be efficiently implemented in FPGAs. Implemented on FPGAs, the proposed solution improves the memory efficiency over the state of the art solutions, while supporting a packet throughput greater than 100 Gbps.

While the prefixes and their characteristics are known when routing packets in the Internet network, this is not true for programmable data planes. Hence, the second solution, designed for programmable data planes, does not exploit any prior knowledge of the prefix stored. We present a framework comprising an efficient data structure to encode the prefixes and methods to map the data structure efficiently to FPGAs. First, the framework leverages a B-tree, extended to support the LPM operation, for its low algorithmic complexity. Second, we present a method to allocate at compile time the minimum amount of resources that can be used by the B-tree. Third, our framework selects the B-tree parameters to increase the post-implementation memory efficiency and generates the corresponding hardware architecture. Implemented on FPGAs, this solution supports packet throughput greater than 100 Gbps, while improving the performance over the state of the art.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ABB	Address Block Binning
AS	Autonomous System
ASIC	Application-Specific Integrated Circuit
BGP	Border Gate Protocol
BST	Binary Search Tree
BRAM	Block RAM
CPU	Central Processing Unit
DAT	Density Adaptive Trie
DRAM	Dynamic Random Access Memory
FIB	Forwarding Information Base
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HTT	Hybrid-Trie-Tree
LB	Leaf Bucket
LB_i	Lower Bounds i
LL	Lookup Latency
LPM	Longest Prefix Match
LSB	Least Significant Bit
LUT	Lookup Table
MAT	Match Action Table
MSB	Most Significant Bit
NHI	Next Hop Information
NIC	Network Interface Card
OSPF	Open Shortest Path First
PCR	Prefix Candidate Resolution
PCS	Prefix Candidate Selection
PE	Processing Engine
PISA	Protocol Independent Switch Architecture
PLS	Prefix Length Sorting
RAM	Random Access Memory

RIB	Routing Information Base
RIP	Routing Information Protocol
SDN	Software Defined Networking
SRAM	Static Random Access Memory
TCAM	Ternary Content Addressable Memory
TE	Traversal Engine
TPU	Tensor Processing Unit
UL	Update Latency

CHAPITRE 1 INTRODUCTION

1.1 Contexte

Un réseau informatique est divisé entre les utilisateurs finaux, qui génèrent et reçoivent des paquets, et des routeurs [1] qui aiguillent les paquets vers les utilisateurs finaux. Historiquement, Internet a été conçu afin que la complexité du traitement effectué dans le réseau soit effectuée par les utilisateurs finaux et non par les routeurs [1]. En effet, le requis principal des architectes du réseau Internet était la flexibilité d'interconnexion avec les réseaux existants et futurs. En réponse à ce requis, les routeurs implémentent uniquement la fonctionnalité d'aiguillage de paquets, ce qui correspond à la fonctionnalité minimale requise pour interconnecter des réseaux entre eux. Par ailleurs, considérant la faible vitesse des liens initialement utilisés dans le réseau Internet, de l'ordre de la dizaine de kb/s, les routeurs étaient implémentés jusque dans les années 1990 en logiciel sur des processeurs — en anglais *Central Processing Unit* (CPU).

Presque quarante années plus tard, le paysage de la réseautique a changé drastiquement, principalement en raison de l'émergence de centres de données de très grandes tailles. Premièrement, les routeurs n'effectuent plus uniquement l'aiguillage de paquets, mais aussi des fonctionnalités reliées à la sécurité, à la surveillance du trafic, ainsi qu'à la qualité de service. Deuxièmement, les routeurs s'appuient actuellement sur des circuits intégrés spécifiques à une application – en anglais *Application-Specific Integrated Circuit*s (ASICs) —, afin de traiter des débits allant de 100 Gb/s à plusieurs dizaines de Tb/s. De tels débits ne peuvent en effet être soutenus par des solutions logicielles. Toutefois, l'innovation est devenue limitée par la capacité de programmer les routeurs. Ainsi, de nombreux algorithmes proposés afin de maximiser la performance d'un réseau n'ont pu être déployés, en raison de l'incapacité par les ASIC existants d'identifier les nouveaux protocoles associés à ces algorithmes.

Afin de faciliter le déploiement de nouveaux algorithmes, le paradigme de réseau défini par Logiciel – en anglais *Software Defined Networking* (SDN) – a été introduit [2]. Suivant ce paradigme, un réseau est divisé en deux parties : le plan dit de contrôle, logiquement centralisé, et le plan de données, effectuant le traitement des paquets, commandé par le plan de contrôle [3]. Récemment, un langage, P4 [4], et une architecture, PISA [5], spécifiques à la programmation d'un plan de données ont été proposés. P4 permet de décrire comment les paquets sont traités par un plan de données programmable, ce qui permet le déploiement de

1. Le terme routeur utilisé dans cette thèse réfère à la fois à un commutateur réseau et à un routeur.

nouveaux algorithmes réseau. Par ailleurs, l'architecture PISA, programmée dans le langage P4, a été implémentée en ASIC et permet de soutenir des débits de l'ordre du Tb/s [5].

Malgré l'apparition de plan de données programmables, dans les centres de données ayant adopté le paradigme de SDN, certaines fonctionnalités demeurent exécutées en logiciel, [6] car elles sont soit non supportées, ou excédent les capacités des plans de données programmables [6], [7]. En particulier, une partie des opérations de recherche permettant d'identifier où un paquet doit être aiguillé sont exécutées en logiciel [2] [6], [7]. Néanmoins, l'exécution de traitements réseaux en logiciel pose un problème d'extensibilité. En effet, la vitesse des liens connectant les serveurs d'un centre de données à son réseau a explosé en quelques années par un facteur 100×, passant de 1 Gb/s, à 40 Gb/s et maintenant 100 Gb/s [6]. Afin de faire face à l'augmentation du débit des liens, un nombre croissant de curs CPU sont utilisés pour traiter du trafic en logiciel. Néanmoins, l'augmentation de puissance de calcul des CPU est limitée à environ 3% par année, ce qui est inférieur au taux d'augmentation du débit des liens. Or, les ressources utilisées pour les traitements réseaux sont un coût de renoncement pour un opérateur de centre de données, qui ne peut les louer à des clients.

Pour maximiser les revenus d'exploitation, des FPGA sont utilisés pour exécuter et accélérer les traitements réseaux afin de libérer des coeurs de CPU [3] [6], [11]. En effet, des FPGA ont été intégrés dans les centres de données, initialement pour accélérer des applications d'apprentissage machine ou pour l'indexation de données [11], [12]. L'utilisation de FPGA pour des traitements réseaux a permis non seulement de libérer des coeurs CPU, mais aussi de diminuer la latence de traitement, une propriété importante pour garantir l'expérience utilisateur. Néanmoins, l'utilisation de FPGA pour des traitements réseaux, tel que l'aiguillage de paquets, demeure une tâche difficile.

Premièrement, l'opération de recherche du plus long préfixe — en anglais LPM —, utilisée pour le routage de paquets, un type d'aiguillage de paquets, est implémentée de manière inefficace sur des FPGA. L'opération LPM consiste à comparer une clé de recherche, usuellement l'adresse IP de destination contenue dans l'entête d'un paquet, à une collection de préfixes contenus dans une table de recherche. Deuxièmement, programmer des FPGA pour des traitements réseaux demeure complexe, en raison de la différence entre les abstractions présentées par les langages de description de matériel — en anglais *Hardware Description Language*s (HDLs) —, et les abstractions utilisées pour du traitement de paquets. Par conséquent, des travaux récents se sont intéressés à compiler le langage P4 vers des FPGA [13],

2. Des routeurs demeurent néanmoins utilisés pour effectuer certaines opérations d'aiguillage de paquets dans un centre de données.

3. Des FPGA sont aussi intégrés dans certains routeurs [8]-[10] pour élargir le spectre de fonctionnalités pouvant être exécutées.

[14]. Néanmoins, certaines abstractions présentées dans le langage P4, telles que les tables d'action et de recherche configurées pour l'opération LPM sont implémentées de manière inefficace sur des FPGA. Ainsi l'implémentation efficace de l'opération de recherche du plus long préfixe sur FPGA demeure un problème ouvert, que cela soit pour le routage de paquets, ou pour la mise en oeuvre d'un plan de données programmable.

1.2 Problématique

La problématique de cette thèse est l'implémentation efficace de l'opération LPM sur FPGA, c'est-à-dire soutenant un débit de recherches élevé, une faible latence de recherche et une faible consommation de ressources FPGA. Cette problématique est étudiée pour deux contextes. Premièrement, on s'intéresse, aux plans de données programmables, où l'opération LPM est une opération de base pouvant être combinée à d'autres opérations de base pour implémenter le traitement associé à une application réseaux. Deuxièment, on s'intéresse spécifiquement à l'application de routage de paquets IPv6 dans le réseau Internet. La problématique étudiée découle de l'inadéquation entre les contraintes de performance spécifique pour l'opération de routage de paquets IPv6 dans le réseau Internet et pour les plans de données programmables au regard de la performance offerte par les architectures matérielles existantes sur FPGA. Nous détaillons ci-dessous les contraintes pour ces deux contextes, ainsi que les limites des architectures existantes.

1.2.1 Contraintes des réseaux programmables

Premièrement, la solution proposée doit soutenir un débit égal ou supérieur à 100 Gb/s, correspondant au débit des liens actuels et futurs. Deuxièment, la solution doit être flexible. En effet, le nombre d'entrées contenues dans la table de recherche, ainsi que la largeur d'une entrée sont spécifiés par l'utilisateur. Ainsi, la solution proposée doit pouvoir être configurée lors de l'implémentation pour une configuration spécifique de table de recherche. De plus, les entrées contenues dans la table de recherche sont ajoutées à l'exécution, c'est-à-dire après l'étape d'implémentation. Par conséquent, la solution proposée ne peut faire levier sur aucune caractéristique des entrées contenues dans la table de recherche.

1.2.2 Contraintes pour le routage de paquets IPv6 dans le réseau Internet

Premièrement, le nombre d'entrées contenues dans une table de recherche utilisée dans le réseau Internet est typiquement très élevé environ 760000 préfixes IPv4 et 79000 préfixes IPv6 en janvier 2020 selon la référence [15] et ces nombres augmentent dans le temps avec une

tendance quadratique [15]. Les ressources logiques d'un FPGA étant limitées, une solution efficace en termes de consommation de ressources logiques est requise. Deuxièmement, en raison de la taille d'une adresse IPv6, encodée sur 128 bits, soit quatre fois la taille d'une adresse IPv4, les solutions proposées pour IPv4 ne peuvent être utilisées et une solution spécifique pour IPv6 est requise [16]. Troisièmement, à 100 Gb/s, une opération de recherche doit être effectuée à chaque 5 ns pour un paquet de taille minimale [15]. Pour soutenir la prochaine génération de liens à 400 Gb/s, cette contrainte est réduite à 1,5 ns, ce qui requiert une architecture hautement optimisée. Enfin, la latence de recherche de l'architecture matérielle doit être réduite afin de satisfaire les contraintes de latence bout-en-bout très faible de certaines applications en développement [17].

1.2.3 Limites des architectures matérielles

L'opération LPM est implémentée en matériel en utilisant des mémoires associatives. Celles-ci sont émulées sur FPGA, car elles ne sont pas disponibles comme blocs de base dans les architectures actuelles. Deux approches sont utilisées pour émuler des mémoires associatives spécialisées pour l'opération LPM ; l'approche «mémoire transposée» [18] et l'approche algorithmique [16].

L'approche mémoire transposée ne peut être utilisée dans les deux contextes d'intérêt, en raison du faible débit de traitement et d'une consommation de ressources logiques élevée [18], [19]. L'approche algorithmique permet de réduire la consommation en ressources logiques. Par conséquent, des tables de recherche plus profondes peuvent être utilisées. Néanmoins, les solutions existantes ne permettent pas conjointement de soutenir un débit de recherches élevé, une faible latence de recherche, ainsi que d'avoir une faible consommation de ressources logiques. Finalement, dans le cas spécifique des réseaux programmables, l'ensemble des solutions présentées dans la littérature ne peuvent être utilisées, car elles requièrent une connaissance de la table de recherche avant la phase d'implémentation, ce qui n'est pas possible dans ce contexte.

1.3 Objectifs de recherche

Dans cette thèse, l'opération LPM est effectuée en adoptant l'approche algorithmique, où le contenu d'une table de recherche est encodé dans une structure de données implémentée sur FPGA. Par conséquent, les objectifs de recherche couvrent la conception de structures de données, ainsi que la conception d'architectures matérielles pour traverser les structures de données proposées. Plus spécifiquement, les objectifs suivants sont considérés :

1. Proposer une structure de données exploitant les caractéristiques des tables de recherche utilisée pour du routage IPv6 dans le réseau Internet, dont la complexité algorithmique en espace, temps de recherche et temps de mise à jour est faible.
2. Proposer une structure de données indépendante aux caractéristiques des tables de recherche utilisées, dont la complexité algorithmique en espace, temps de recherche et temps de mise à jour est faible
3. Concevoir des architectures matérielles implémentant les structures de données proposées et améliorant les métriques de performance.

1.4 Contributions principales

Dans cette thèse, trois articles sont présentés :

1. Stimpfling, Thibaut, Normand Bélanger, JM Pierre Langlois, et Yvon Savaria. «SHIP : A Scalable High-performance IPv6 Lookup Algorithm that Exploits Prefix Characteristics ». IEEE/ACM Transactions on Networking 27, numéro. 4 (2019) : 1529-1542.
2. Stimpfling, Thibaut, JM Pierre Langlois, Normand Bélanger, et Yvon Savaria. «A Low-Latency Memory-Efficient IPv6 Lookup Engine Implemented on FPGA Using High-Level Synthesis». 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 402-411. IEEE, 2018.
3. Stimpfling, Thibaut, Jeferson Santiago da Silva, François-Raymond Boyer, Yvon Savaria, and JM Pierre Langlois. «Efficient Longest Prefix Matching for Programmable Data Planes on FPGAs». soumis au journal IEEE/ACM Transactions on Networking.

En résumé, les deux premiers articles de cette thèse visent à répondre à la question «Comment implémenter *efficacement* l'opération LPM sur FPGA pour le routage de paquets IPv6 dans le réseau Internet ». À cet égard, une structure de données compacte, SHIP, est proposée pour encoder efficacement les préfixes contenus dans la table de recherche. Deux architectures sont présentées pour implémenter le parcours de SHIP sur FPGA. La première est conçue pour soutenir un débit élevé et une faible latence de recherche, tandis que la seconde est conçue pour atteindre une faible latence de recherche. De plus, une méthode est présentée pour décrire les deux architectures proposées pour une synthèse de haut niveau efficace. Le troisième article s'intéresse exclusivement au contexte des réseaux programmables sur FPGA. Une structure de données conçue spécifiquement pour les requis des réseaux programmables sur FPGA est présentée. De plus, nous présentons un générateur d'architecture, pour l'implémentation de la structure de données proposée, soutenant un débit de recherche élevé et ayant une faible consommation de ressources logiques.

Nous décrivons maintenant de manière spécifique les contributions effectuées dans chacun des articles.

Le premier article, présente deux contributions : SHIP, une structure de données compacte pour encoder les préfixes et une architecture pour parcourir SHIP sur FPGA permettant d'atteindre un débit de recherches élevé et une faible latence de recherche. Afin de réduire la consommation mémoire par rapport aux solutions existantes, SHIP est construite en exploitant les caractéristiques des préfixes IPv6 utilisés dans le réseau Internet et en adoptant une approche de type «diviser pour régner». Premièrement, la table de recherche est divisée en groupes de préfixes, en utilisant la méthode de regroupement «Two Level Prefix Grouping». Celle-ci regroupe d'abord les préfixes IPv6 selon leur structure d'allocation, puis trie les préfixes par longueur, afin de créer des groupes de préfixes de faible cardinalité. Deuxièmement, les préfixes contenus dans chacun des groupes sont encodés dans une structure de données hybride combinant un arbre préfixe à un arbre binaire, — en anglais un *Hybrid Trie-Tree (HTT)*. Cette structure de données hybride est proposée afin de combiner les forces des arbres préfixes et des arbres binaires, tout en réduisant leurs défauts respectifs. L'arbre préfixe est sélectionné pour identifier rapidement un ensemble de préfixes candidats pouvant générer une comparaison positive. Néanmoins, l'arbre préfixe est coûteux pour sélectionner parmi un ensemble de préfixes celui générant une comparaison positive. Ainsi, l'ensemble de préfixes identifiés est encodé dans un arbre binaire, ayant une faible empreinte mémoire. De plus, des méthodes sont introduites pour les deux structures de données afin de réduire la consommation mémoire. Finalement, une architecture implémentant la traversée de la structure de données est présentée. Celle-ci est conçue pour atteindre un débit de recherches élevé et une faible latence de recherche.

Le second article présente une architecture matérielle, implémentant la traversée de SHIP, conçue pour atteindre une faible latence de recherche. Les contributions de cet article sont l'architecture matérielle, ainsi qu'une méthode de description de l'architecture matérielle pour une synthèse de haut niveau efficace. L'architecture matérielle présentée est entièrement pipelinée et organisée afin de réduire la consommation de ressources logiques. Pour ce faire, l'opération de traversée de la structure de données est modifiée, afin que l'étape de traitement la plus coûteuse en termes de consommation logique soit exécutée uniquement dans le dernier étage de l'architecture. Par ailleurs, quatre techniques de restructuration de code sont présentées pour la description d'une architecture matérielle conçue pour une synthèse de haut niveau.

Le troisième article porte sur l'opération LPM pour des réseaux programmables implémentés sur FPGA. La contribution principale de cet article est un cadre de référence permettant de

générer automatiquement une architecture matérielle implémentant l'opération LPM, à partir d'une description d'une table de recherche et d'une description d'architecture de FPGA. L'opération LPM, implémentée en exploitant un arbre B, est rendue possible grâce à deux méthodes. Premièrement, les préfixes sont transformés en intervalles disjoints deux par deux. Deuxièmement, afin de respecter les contraintes des réseaux programmables sur FPGA, les ressources utilisées par la structure de données proposée sont entièrement allouées à la compilation. La méthode utilisée permet d'allouer le minimum de ressources à la compilation. De plus, nous démontrons théoriquement que les ressources allouées permettent d'effectuer les opérations de mise à jour. Par ailleurs, une méthode est présentée afin de réduire la consommation de ressources mémoires du FPGA lors de l'implémentation de la structure de données. Cette méthode permet d'identifier pour chacun des niveaux de la structure de données le type de mémoire et la configuration de celle-ci afin d'augmenter l'efficacité mémoire post implémentation. Finalement, cet article propose un outil pour la génération de l'architecture matérielle.

1.5 Contributions complémentaires non incluses dans cette thèse

Cette section présente les travaux complémentaires effectués durant cette thèse, mais qui ne constituent pas le coeur de celle-ci. Ces travaux portent sur la place des FPGA comme plan de données programmable : *Est-ce que les FPGA peuvent être utilisés au-delà des cartes réseaux ? Est-ce que l'architecture actuelle des FPGA est adaptée aux plans de données programmables ?*. Les questions soulevées et les éléments de réponse présentés sont l'objets de plusieurs publications [8], [20], [21], présentées succintement ci-dessous.

***Architecture hétérogène d'un plan de données programmable* — [8].** Nous nous sommes intéressés à l'architecture de plan de données programmables hétérogènes, comprenant des ASIC et des FPGA. La motivation associée à l'utilisation d'une architecture hétérogène est d'augmenter les capacités de traitement d'un ASIC par l'ajout de FPGA. Ce travail a été publié à *P4 Workshop in Europe* [8].

***Mécanisme de cache pour une architecture hétérogène* — [20].** Un des défis majeurs d'une architecture hétérogène est la différence de débit soutenu par chacune de ces deux plateformes. Là où un ASIC supporte une dizaine de Tb/s, un FPGA est limité une centaine de Gb/s. Une telle architecture hétérogène risque donc d'être limitée par le débit soutenu par un FPGA. Pour réduire l'impact de ce problème, une solution de cache est proposée, visant à maximiser le trafic traité sur l'ASIC, tout en profitant des bénéfices

d'une architecture hétérogène. Ce travail a été soumis à *ACM Computer Communication Review* [20]

***Limites et avenues d'améliorations pour un plan de données programable implémenté sur FPGA* — [21].** Nous avons étudié les limites de performances d'un plan de données programmable implémenté sur FPGA. Nous montrons que la performance de plusieurs blocs de l'architecture *Protocol Independent Switch Architecture* (PISA) [5] est limitée par l'architecture actuelle des FPGA. Pour pallier à ces limitations, deux avenues sont explorées. Premièrement, nous identifions un sous-ensemble d'applications réseaux n'utilisant pas ces blocs souffrant d'une faible performance, pouvant faire levier sur l'architecture actuelle des FPGA. Deuxièmement, nous proposons de spécialiser l'architecture des FPGA afin de supporter un plus grand ensemble d'applications réseaux. Ce travail a été accepté à *IEEE International Conference on High Performance Switching and Routing* [21].

1.6 Plan de la thèse

Cette thèse est organisée en sept chapitres. Tout d'abord, une revue de la littérature est présentée dans le chapitre 2, ainsi qu'un survol des réseaux programmables et de l'opération de routage de paquets. Les chapitres suivants présentent les contributions de cette thèse. Le chapitre 3 présente une solution algorithmique pour implémenter l'opération LPM sur FPGA pour le routage de paquets IPv6 dans le réseau Internet. Cette solution inclut la structure de données SHIP, ainsi qu'une architecture matérielle pour parcourir SHIP, conçue pour atteindre un débit de recherche élevé et une faible latence de recherche. Puis, le chapitre 4, présente une architecture matérielle alternative, conçue pour une faible latence de recherche, ainsi que des techniques de description d'architecturale pour une synthèse de haut niveau. Dans le chapitre 5, une solution matérielle implémentant l'opération LPM pour des réseaux programmables sur FPGA est introduite. Le chapitre 6 contient une discussion générale sur les contributions de cette thèse. Finalement, le chapitre 7 présente la conclusion de cette thèse et introduit différents axes de travaux futurs.

CHAPITRE 2 CONTEXTE GÉNÉRAL, RAPPELS ET REVUE DE LITTÉRATURE

Ce chapitre a pour but de présenter le contexte général de cette thèse, afin de bien situer l'importance et l'impact de l'opération LPM dans les réseaux. Nous présentons dans un premier temps le paradigme de réseaux programmables, incluant un survol des plateformes matérielles utilisées. Un réseau programmable permet d'exécuter de nombreuses nouvelles applications, mais aussi des applications traditionnelles, telles que le routage de paquets, que nous introduisons dans un second temps. Le routage de paquets utilise une opération LPM afin de déterminer le chemin vers lequel un paquet doit être aiguillé. Nous présentons ensuite comment l'opération LPM est implémentée sur différentes architectures. Cette revue de littérature complémente les revues de littérature spécifique présentées dans les chapitres 3, 4 et 5.

2.1 Réseaux programmables

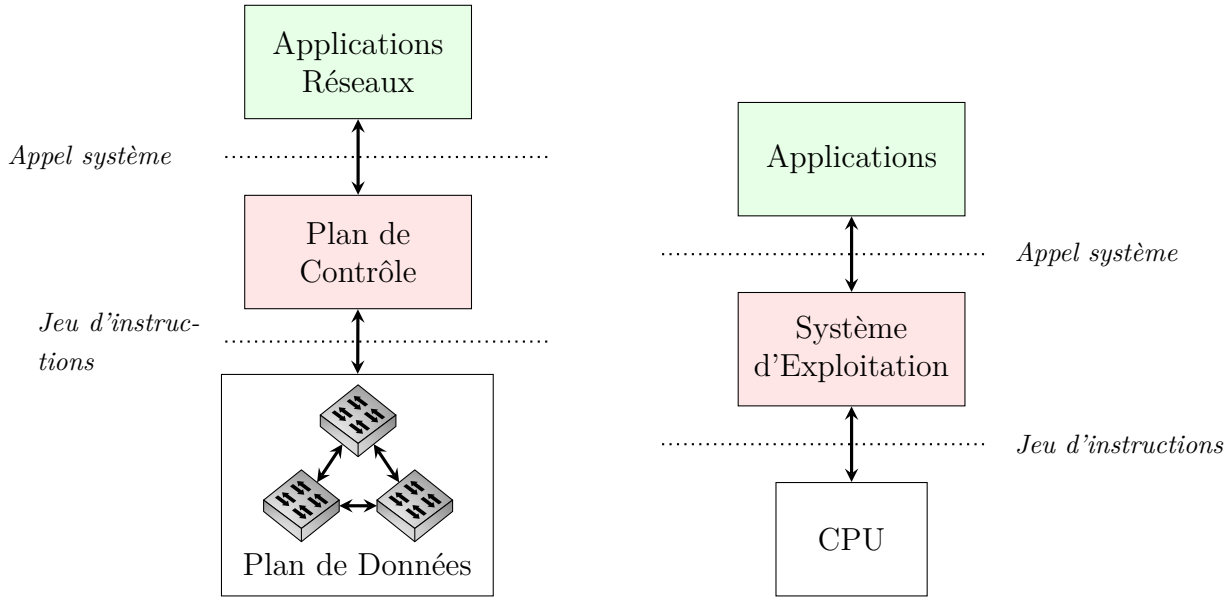
2.1.1 Motivation

Durant les trente dernières années, les chercheurs se sont intéressés à rendre les réseaux programmables [22]. En effet, en l'absence de cette capacité de programmation, de nombreux algorithmes proposés ne pouvaient être déployés dans l'ensemble des réseaux [2], ce qui restreignait l'émergence de nouvelles idées.

L'émergence du paradigme de réseau définie par logiciel — en anglais SDN —, a marqué un tournant majeur vers la programmation des réseaux [3]. Cela a été rendu possible grâce à 1) l'introduction des abstractions de plan de contrôle et de plan de données et 2) à une architecture de réseau où le plan de données est séparé du plan de contrôle. Dans le paradigme SDN, le plan de données est une entité pouvant uniquement traiter des paquets, tandis que le plan de contrôle est une entité logiquement centralisée qui configure et commande le plan de données. De plus, le plan de contrôle présente une vue abstraite du réseau aux applications.

Ces deux abstractions permettent une séparation des préoccupations [23], [24] — en anglais *separation of concerns* — et donc de simplifier le processus de développement et de maintenance d'une application. Par exemple, en présentant une vue abstraite du réseau à une application, les problématiques de système distribué sont abstraites lors de la conception d'une application réseau [23], [24]. Par ailleurs, Shenker [23], [24] explique ainsi que l'informatique a pu se construire en tant que champs d'études grâce à l'introduction d'abs-

tractions, présentées dans la figure 2.1b. Au contraire, selon lui, jusqu'à présent, le domaine du réseau était une pratique et non une discipline, en raison de l'absence d'abstractions. Toutefois, comme présenté dans la figure 2.1a, un parallèle peut maintenant être établi entre les abstractions utilisées dans un réseau défini par logiciel et les abstractions utilisées dans un système informatique.



(a) Couches et interfaces dans le paradigme SDN.

(b) Couches et interfaces d'un ordinateur.

Figure 2.1 Similitudes entre les abstractions proposées pour le SDN et les abstractions utilisées en informatique. Cette figure est inspirée du travail de Song [25]

Historiquement, trois avancées successives ont eu lieu dans le domaine des réseaux programmables [1]. La première avancée est OpenFlow [2], un protocole de communication permettant de configurer le traitement effectué par le plan de données. Bien que le nombre de protocoles compatibles, ainsi que les traitements disponibles soient limités [26], OpenFlow a permis pour la première de fois de configurer un plan de données. La seconde avancée porte sur le plan de contrôle et s'est traduite par le développememet de systèmes d'exploitation pour le réseau, ainsi que des langages de haut niveau pour décrire des applications réseaux [27]-[31]. La troisième avancée porte sur la programmation du plan de données [4], [5], avec l'introduction de P4 [4], un langage spécifique pour configurer un plan de données et de la PISA [5], une architecture de plan de données programmable. [2]. Pour mettre les choses en perspective, P4 [4] et PISA [5] sont pour le réseau ce que Tensorflow [32] et *Tensor Processing Unit* (TPU) [33]

1. Nous parlons uniquement ici des avancées *majeures* depuis OpenFlow. Une présentation complète des différentes évolutions est présentée dans [22].

2. Les différences entre Openflow et P4/PISA en termes de capacité de programmations sont explicitées dans [5].

sont pour l'apprentissage machine. Nous présentons ci-dessous de manière plus détaillée le langage P4 et l'architecture PISA.

2.1.2 Langage P4 et architecture PISA

Le langage P4 [4] introduit trois abstractions pouvant être combinées pour décrire une application réseau : l'extraction d'entêtes d'un paquet — en anglais *packet parser* —, la table de comparaison et d'action — en anglais *match action table* — et le réassemblage des entêtes d'un paquet — en anglais *packet deparser*. Une quatrième abstraction, l'ordonnancement de paquets — en anglais *packet scheduler* — est présentée ici, bien qu'elle ne soit pas encore intégrée à la dernière spécification du langage P4 [34].

De manière réciproque, l'architecture PISA, présentée à la figure 2.2, est composée de modules configurables implémentant les abstractions du langage P4.

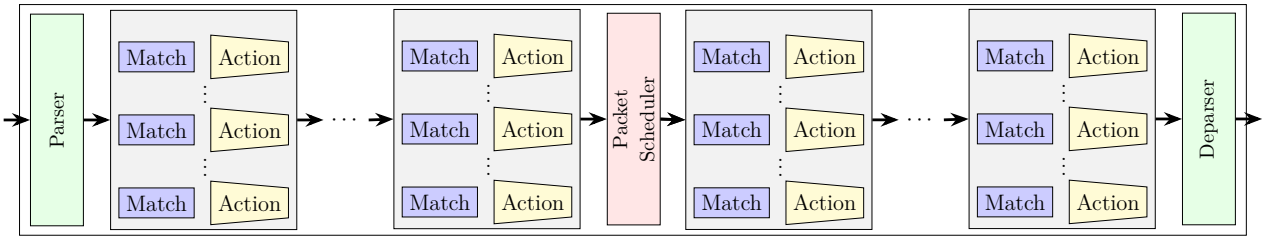


Figure 2.2 Architecture PISA

Extraction de paquets. Ce module extrait les champs contenus dans les entêtes d'un paquet, selon la description des entêtes et séquences d'entêtes, effectués dans un programme P4.

Table de comparaisons et d'actions. Ce module compare une clé de recherche à une collection de paires < clé : action > et exécute une action lors d'une comparaison positive. Une *Match Action Table* (MAT) est définie par quatre propriétés : une taille, des actions, une clé et un type de comparaison [34]. La taille de la MAT correspond à la profondeur de la table. Une action est une fonction avec des paramètres optionnels. Lors d'une comparaison, l'action associée à la règle sélectionnée est exécutée. Une clé de recherche est généralement constituée d'un ou plusieurs champs de l'entête du paquet. Trois types de comparaisons sont spécifiées dans le langage P4 [34] : exact, LPM et ternaire. L'opération de comparaison de type exact consiste à comparer l'intégralité de la clé de recherche aux clés dans la collection de paires, là où seule une portion de la clé de recherche est comparée pour le type LPM ou

ternaire. Le type ternaire est une généralisation du type LPM et permet de sélectionner pour la comparaison n'importe quels bits de la clé de recherche, là où seule une séquence de bits contigus de poids forts peut être utilisée pour le type LPM.

Ordonnancement de paquets. Ce module réordonne les paquets, c'est-à-dire permet de spécifier l'ordre entre les paquets et les moments auxquels les paquets doivent être émis. Bien que cette abstraction soit requise pour exprimer une grande variété d'applications réseaux, la programmation d'un tel module ainsi que son architecture demeure une question de recherche ouverte [35]-[38].

Réassemblage des paquets. Ce module ajoute les nouvelles entêtes du paquet, selon le traitement effectué par les blocs précédents, avant que le paquet ne soit transmis.

Initialement, l'architecture PISA a été proposée pour des ASIC. Une description complète de la micro architecture ainsi qu'une évaluation du surcoût de la programmabilité est présentée dans [5]. Récemment, l'architecture PISA a été portée vers d'autres plateformes, incluant les *Network Interface Card* (NIC) [39], les CPU [40] et les FPGA [13], [14]. Dans le cas spécifique des FPGA, les enjeux actuels consistent au développement de micro architectures optimisées pour FPGA, principalement pour les modules d'extraction d'entêtes [41], [42], de réordonnancement de paquets [35] et de comparaisons et d'actions [19], [43].

Les plans de données programmables permettent d'exécuter un très grand nombre d'applications réseaux [44]-[50], incluant le routage de paquets [5], présenté dans la prochaine section.

2.2 Routage de paquets

L'opération de routage, effectué à un point d'interconnexion entre plusieurs réseaux, consiste à identifier le réseau vers lequel un paquet, transmis par un expéditeur, doit être *routé* pour atteindre un ou plusieurs destinataires. Dans cette section et dans le reste de cette thèse, on s'intéresse uniquement au routage d'un émetteur vers un unique destinataire — en anglais *unicast*. L'opération de routage de paquets est répartie entre le plan de données, qui identifie vers quel réseau un paquet doit être transmis, et le plan de contrôle, qui construit une «carte»des différents réseaux joignables, ainsi que les chemins menant à chacun des réseaux. Nous présentons, ci-dessous, le rôle du plan de contrôle et du plan de données pour l'opération de routage de paquets.

2.2.1 Plan de contrôle

Un *Routing Information Base* (RIB) contient une description de la topologie d'un réseau, inférée selon les informations de topologie locale transmises par les routeurs voisins. Ces derniers transmettent les réseaux auxquels ils sont connectés, le(s) chemin(s) pour les atteindre et dans certains cas une métrique de performance caractérisant le(s) chemin(s) [3].

Un protocole de routage est utilisé pour échanger une information de topologie entre des routeurs. Nous présentons ci-dessous les protocoles de routage traditionnellement utilisés. Ces derniers sont classifiés en deux groupes, selon qu'ils sont utilisés à l'intérieur ou à l'extérieur d'un système autonome — en anglais *Autonomous System* (AS). Un système autonome est défini comme un groupe de réseaux connectés [4] ayant une politique de routage unique et cohérente [51]. Un exemple d'AS est un fournisseur d'accès à Internet.

Protocoles de routage interne. Ceux-ci sont sous-divisés en protocoles à vecteur de distance [52], protocoles à états de liens [5] [53].

Routing Information Protocol (RIP) [52] est un protocole à vecteur de distance, mesurant le nombre de routeurs à traverser pour acheminer un paquet vers une destination. Les meilleurs chemins sont ceux ayant la métrique de distance la plus petite.

Le défaut du protocole RIP est qu'il considère uniquement la distance comme métrique de performance et non la vitesse ou qualité du lien. Le protocole *Open Shortest Path First* (OSPF) [53] a ainsi été proposé afin de pallier à ce défaut. OSPF est un protocole à état de lien, où chacun des routeurs transmet de proche en proche la liste des réseaux auxquels il est connecté, incluant le coût de la connexion [6]. Les meilleures routes sont ensuite sélectionnées en appliquant l'algorithme du plus court chemin proposé par Dijkstra [55].

Protocole de routage externe. *Border Gate Protocol* (BGP) [56] est l'unique protocole de routage externe utilisé. Néanmoins le protocole BGP relève d'une importance primordiale, considérant qu'il est utilisé pour le routage des paquets dans le réseau Internet. BGP adopte une approche de type vecteur de chemin. Chaque AS transmet à ses voisins des messages spécifiant les réseaux auxquels il est connecté ainsi que le chemin pour les atteindre. Ces deux informations sont encodées dans un vecteur de chemin. Lorsqu'un AS reçoit un message

3. La métrique utilisée pour évaluer le coût d'un chemin dépend du protocole de routage exécuté.
4. Ici un réseau est défini par un unique préfixe.
5. Un protocole hybride existe, combinant les deux approches présentées, mais est un protocole propriétaire de Cisco.
6. La métrique de performance associée n'est pas définie dans la spécification du protocole OSPF, mais en pratique comprend la latence, la vitesse du lien et la charge du lien [54].

permettant l'identification d'un nouveau meilleur chemin, alors l'AS est ajouté au chemin reçu, et le nouveau chemin est transmis aux voisins de l'AS. Par opposition, un message n'est pas retransmis s'il ne contient pas un nouveau meilleur chemin non connu. Une procédure similaire est utilisée pour supprimer des chemins. De nombreuses métriques sont utilisées par BGP pour sélectionner un chemin par rapport à un autre et dépendent de la configuration choisie par l'opérateur du réseau. De plus, BGP étant utilisé pour connecter des réseaux indépendants, les accords commerciaux liant les propriétaires de ces réseaux sont aussi utilisés lors de la sélection des routes. Dans le cas où un routeur est en bordure d'un réseau, le RIB peut contenir des chemins reportés par plusieurs protocoles de routage.

2.2.2 Plan de données

Au niveau du plan de données, l'opération de routage consiste à comparer l'adresse IP de destination, contenue dans l'entête d'un paquet, à une liste de réseaux, contenue dans une table de routage — en anglais *Forwarding Information Base* (FIB). Le FIB est construit à partir d'un RIB, mais contient uniquement un sous-ensemble des réseaux du RIB.

Un entrée dans un FIB contient un préfixe, une information du prochain saut, ainsi qu'une d'interface de sortie [57]. Un préfixe est associé à un réseau et décrit l'ensemble des adresses IP contenues dans ce réseau. L'information de prochain saut — en anglais *Next Hop Information* (NHI) — représente l'adresse IP du routeur vers lequel le paquet est transmis. L'interface de sortie est le port physique du routeur sur lequel le paquet est transmis.

Lors de l'opération de routage, l'adresse IP de destination, dénommée clé de recherche, est comparée aux préfixes du FIB. Un préfixe est consituté d'une adresse IP et d'une longueur de préfixe, spécifiant les bits de poids fort de l'adresse IP du préfixe et de la clé de recherche utilisés pour la comparaison. Une comparaison de type plus long préfixe est utilisée, c'est-à-dire que lorsque plusieurs préfixes génèrent une comparaison positive, seul le plus long préfixe est sélectionné.

Tableau 2.1 Exemple d'un FIB pour des adresses IPv4 encodées sur 32 bits. Les préfixes sont représentés en utilisant la notation adresse IP / longueur de préfixe.

Préfixes	**NHI**	**Interface**
P_0= 241.51.32.129/32	241.51.32.129	2
P_1 =194.123.65.12/16	194.123.1.1	0
P_2=194.123.0.1/24	194.123.0.1	1
P_3 =194.123.1.2/24	10.2.4.5	1
P_4 =152.68.200.1/21	127.0.0.0.1	3

Un exemple de FIB est présenté dans le tableau 2.1. Afin d'illustrer l'opération de routage, on considère une adresse IP de destination $IP_{Dst} = 194.123.1.8$. Celle-ci génère une comparaison positive par rapport aux préfixes P_1 et P_3. Par contre, le préfixe P_3 est sélectionné, car sa longueur est supérieure à celle du préfixe P_1. Dans la section suivante, nous présentons les différentes méthodes pour implémenter efficacement l'opération de comparaison LPM dans un plan de données.

2.3 Implémentation de l'opération LPM

Afin de traiter des bandes passantes élevées, dépassant les 100 Gb/s, l'opération de comparaison LPM est traditionnellement implémentée en utilisant du matériel dédié, tel que les mémoires adressables par contenu ternaire — en anglais *Ternary Content Addressable Memory* (TCAM) —, présentés dans la section §2.3.1. Cependant, la TCAM souffre d'un coût élevé en portes logiques et d'une consommation d'énergie importante. Par conséquent, de nombreuses méthodes permettant d'exécuter l'opération LPM à très haut débit sans utiliser de TCAM ont été proposées et sont présentées dans les sections §2.3.2 et §2.3.4.

2.3.1 TCAM

Fonctionnalité à haut-niveau Une TCAM est utilisée pour identifier le plus long préfixe générant une comparaison positive avec une adresse IP, tandis que l'information associée aux préfixes, les NHI et les interfaces de sorties sont contenus dans une mémoire externe. Une TCAM est une mémoire adressable par contenu, qui permet de comparer en parallèle une adresse IP à chacun des préfixes contenus dans un FIB. La TCAM comprend de la mémoire, contenant les préfixes, de la circuiterie pour comparer l'adresse IP à chaque préfixe et un encodeur de priorité permettant de sélectionner uniquement le plus long préfixe parmi ceux générant une comparaison positive.

Un mot mémoire de la TCAM contient un préfixe, pour lequel chacun des bits de l'adresse IP est encodé sous un format ternaire. En utilisant ce format, un bit peut avoir les valeurs 0, 1 ou "$*$". Une valeur "$*$" est associée à un bit ignoré lors de la comparaison, c'est-à-dire un bit pour lequel le résultat d'une comparaison est toujours positif, indépendamment de la valeur comparée. Le résultat de la comparaison d'une adresse IP à un préfixe est encodé dans un bit de contrôle, transmis à une entrée de l'encodeur de priorité. L'encodeur de priorité retourne l'indice de l'entrée à laquelle est stocké le plus long préfixe. Afin de garantir que l'indice de l'entrée associée au plus long préfixe soit sélectionné, les préfixes sont insérés dans la TCAM par ordre décroissant de longueur de préfixe. L'indice retourné par l'encodeur de priorité est

utilisé comme adresse pour retourner les informations associées au préfixe sélectionné dans la mémoire externe.

Organisation On s'intéresse maintenant à l'organisation interne d'une TCAM de taille NxW, c'est-à-dire contenant N préfixes, ayant chacun une adresse IP encodée sur W bits. Chaque entrée de W bits de la TCAM comprend W cellules ternaires de 1 bit, reliées à une ligne de comparaison, qui est à son tour connectée à un amplificateur de détection [58]. Nous présentons d'abord l'organisation d'une cellule ternaire, puis nous couvrons l'opération de recherche.

Une cellule ternaire contient de la mémoire, pour stocker un bit de l'adresse IP du préfixe et de la logique, pour comparer un bit d'une adresse IP au bit contenu dans la mémoire. Traditionnellement, deux types de cellules TCAM sont utilisées, les types NAND et types NOR. Ces deux types de cellules diffèrent uniquement par le circuit utilisé pour la comparaison. La cellule ternaire NAND comprend quatre transistors pour la comparaison, tandis que la cellule NAND requiert trois transistors. Pour stocker un bit de l'adresse IP du préfixe, deux cellules de 1 bit de *Static Random Access Memory* (SRAM) sont requises en raison de l'encodage ternaire utilisé. Considérant qu'une cellule de SRAM utilise traditionnellement six transistors, le surcoût d'une cellule ternaire par rapport à une cellule de SRAM est d'environ 2.6×.

Nous présentons maintenant l'opération de recherche. Dans le cas d'un mot ternaire composé de cellules de type NAND, la ligne de comparaison est préchargée à V_{dd}. Ensuite, l'opération de comparaison entre l'adresse IP du préfixe et la clé de recherche est effectuée sur chacun des bits. En cas de comparaison négative d'un bit, la sortie de cellule ternaire est connectée à la masse. Considérant que les sorties des cellules ternaires sont connectées à une ligne de comparaison commune, il suffit d'une comparaison négative dans une cellule ternaire pour décharger la ligne de comparaison. Finalement, l'amplificateur de détection infère le résultat de la comparaison en détectant l'état logique associé à la ligne de comparaison.

Pour un mot ternaire utilisant des cellules de type NAND, les cellules sont connectées en série sur la ligne de comparaison. La tension en sortie du circuit de comparaison de chaque cellule permet l'activation d'un transistor qui propage la tension présente sur la ligne de comparaison vers la prochaine cellule ternaire. Initialement, la ligne de comparaison est préchargée à V_{dd} au niveau d'une unique cellule. Puis, lors d'une comparaison, une cellule est connectée à la masse. Par conséquent, lors d'une comparaison positive sur l'ensemble des cellules, la ligne de comparaison est déchargée. Finalement, de manière similaire à la cellule NOR, l'amplificateur de détection identifie le résultat de la comparaison.

Bien que la cellule ternaire NAND utilise un transistor de moins que la cellule NOR, la cellule ternaire de type NOR est préférée en raison de son délai de propagation plus faible [58]. En effet, le chemin critique de la cellule NAND est constitué de W transistors, là où la cellule NOR a un chemin critique de deux transistors.

Limitations Une TCAM souffre d'une consommation d'énergie élevée, proportionnelle à la profondeur N, en plus d'un coût en porte logique supérieur à 2.6× par rapport à une mémoire de type SRAM. Ainsi les travaux actuels consistent à réduire la consommation d'énergie [58], [59]. Toutefois, la réduction de consommation d'énergie est effectuée au prix d'une augmentation de la consommation de ressources logiques [58], [59]. Ainsi, en pratique, le coût en porte logique d'une TCAM est 6× à 7 × plus élevé que de la mémoire SRAM [5].

La TCAM est intégrée uniquement dans des ASIC spécialisés pour le traitement réseau à très haut débit [5], [60], et non dans des FPGA [7], ni les CPU et les *Graphic Processing Unit* (GPU). Par conséquent, pour les plateformes hors ASIC, la fonctionnalité d'une TCAMăest émulée. Deux méthodes d'émulation peuvent être utilisées : l'approche mémoire transposée et l'approche algorithmique. Nous présentons dans les sections suivantes ces deux approches.

2.3.2 Mémoire transposée

Cette approche est utilisée exclusivement sur des FPGA et consiste à utiliser un bloc mémoire de FPGA, tel un bloc RAM [8] où une mémoire distribuée — en anglais *distributed-RAM* —, comme une mémoire associative en *transposant* les bus de données et d'adresse. Abdelhadi *et al.* [18] utilisent l'approche mémoire transposée — en anglais *transposed memory* — pour effectuer l'opération LPM sur FPGA [9].

L'idée générale de l'approche mémoire transposée consiste à stocker pour chaque clé de recherche possible l'indice du préfixe associé, en utilisant un encodage 1 parmi n — en anglais *one-hot*. Ainsi, un bloc RAM de taille $W \times N$, où W est la largeur en bits et N la profondeur, permet d'encoder un FIB de taille $W' \times N'$ où $W' = log_2(N)$ et $N' = W$. À titre d'illustration, un bloc RAM de 36 kbits [62], pour lequel $W = 36$ et $N = 1024$, permet de contenir seulement 36 préfixes encodés sur 10 bits. Considérant le coût très élevé en bloc mémoire, les solutions adoptant cette approche visent à réduire le coût en ressource logique, à encoder des

7. Toutefois, certaines anciennes générations de FPGA du fabricant Lattice Semi intégraient des blocs mémoire configurables permettant une opération de recherche de type ternaire [61], mais sont depuis absents des dernières générations de FPGA.

8. Nous adoptons ici la terminologie de Xilinx.

9. Bien que le travail présenté par Abdelhadi *et al.* [18] soit le seul adoptant l'approche mémoire transposée pour effectuer une opération LPM, cette approche est utilisée par de nombreux travaux pour implémenter l'opération de recherche ternaire sur FPGA [19].

FIB plus larges et profonds. Ainsi Abdelhadi *et al.* [18] présente une méthode de recherche hiérarchique pour encoder des préfixes plus larges, mais aussi pour contenir un plus grand nombre de préfixes. L'idée proposée consiste à voir les vecteurs de bits utilisant l'encodage 1 parmi n contenus dans un bloc mémoire comme une matrice creuse et d'enregistrer uniquement l'indice associé au bit non nul à l'aide d'un encodage à deux niveaux. Néanmoins, malgré les différentes techniques présentées, la consommation de ressources logiques demeurre très élevée et le débit de recherches faible.

2.3.3 Rappel sur les tableaux associatifs

L'idée générale de l'approche algorithmique est d'émuler la fonctionnalité d'une TCAM en utilisant un tableau associatif, c'est-à-dire une structure de données. Nous présentons donc dans cette section un rappel sur les tables de hachage, les arbres binaires et les arbres préfixes, les trois principales structures de données utilisées dans l'approche algorithmique.

Table de hachage. À haut niveau, une table de hachage comprend un tableau contenant une collection de paires < clé : valeur >, une fonction de hachage, permettant de convertir une clé de recherche en un indice du tableau et une méthode de résolution de collisions lorsque des clés différentes génèrent le même indice. Lors d'une opération de recherche, après avoir calculé l'indice associé à une clé de recherche, la clé stockée à cet indice est comparée à la clé de recherche. Dans le cas d'une comparaison positive, la valeur associée contenue dans la paire est retournée.

Nous présentons maintenant les différents types de fonctions de hachage utilisées ainsi que les méthodes de résolution de collisions. Les fonctions de hachage sont classifiées en quatre groupes : multiplication et décalage, multiplication addition et décalage, table de correspondance et Murmur [63]. La performance d'une fonction de hachage est caractérisée par le débit de calculs par seconde, ainsi que par la qualité du hachage [63], représentant la distribution des clés hachées.

Les méthodes de résolution de collisions sont divisées en deux groupes : chainage et adressage ouvert [64], [65]. Dans le cas d'une résolution par chainage, chaque entrée du tableau contient une liste de paires en lieu et place d'une unique paire. Lors d'une collision, les clés générant une collision sont ajoutées dans la liste chainée associée à l'entrée du tableau sélectionnée. Par conséquent, dans le pire des cas, une recherche peut nécessiter de traverser une liste chainée contenant la collection complète de paires. Ainsi, la complexité de recherche de cette méthode est $O(N)$ où N est la cardinalité de la collection de paires.

Afin de réduire la complexité de recherche, les méthodes à adressage ouvertes ont été propo-

sées. L'idée générale consiste à insérer une clé générant une collision dans une entrée vide du tableau. Ainsi le tableau est parcouru à partir de l'indice $indice_{Collision}$ où la collision a lieu et le prochain indice sélectionné est $indice_{Collision} + K$, où K peut être une valeur constante, ou générée par un polynôme [63], [64]. Le processus est répété jusqu'à ce qu'un indice ne contenant pas de paire soit identifié. Une méthode récente d'adressage ouvert, le hachage de type coucou [66], consiste à diviser une table de hachage de capacité M, en k tables de hachage, ayant chacune une capacité de M/k. L'idée générale derrière cette technique consiste à évincer la clé présente dans la table de hachage générant la collision par la nouvelle clé à insérer [10]. La clé évincée est ensuite insérée dans une autre table de hachage et le processus est réitéré, jusqu'à ce qu'une entrée vide soit identifiée dans une des k tables de hachage [11]. En utilisant une méthode de résolution de type Coucou, la complexité de recherche devient $O(1)$.

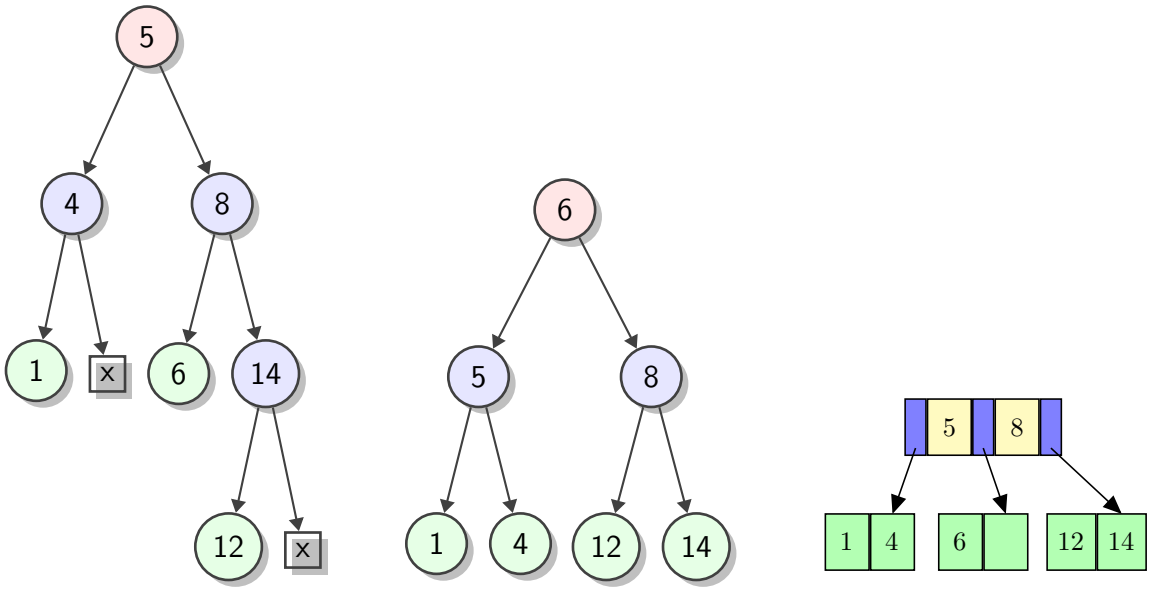
Une table de hachage ne peut être utilisée telle quelle pour effectuer une recherche sur les préfixes d'un FIB. En effet, une fonction de hachage ne peut être appliquée sur les bits non définit de l'adresse IP d'un préfixe, les bits de type " $*$ ". Par conséquent, deux techniques peuvent être employées. Premièrement, une table de hachage peut être utilisée par longueur de préfixe. Néanmoins cette approche requiert d'effectuer une recherche dans W table de hachage. Deuxièmement, afin de réduire le nombre de tables de hachage à traverser, les préfixes peuvent être convertis vers un ensemble réduit de longueurs de préfixes. Cependant, cette approche augmente de manière exponentielle le nombre de préfixes à stocker [69].

Arbre binaire. Un arbre binaire est un graphe orienté acyclique, où chaque noeud contient une paire $<$ clé : valeur $>$ et est connecté au plus à deux noeuds enfants et à un noeud parent. Le noeud initial est appelé racine, tandis que les noeuds sans enfants sont appelés feuilles [64], [65]. Un arbre binaire permet d'identifier si une clé de recherche est contenue dans une collection de paires $<$ clé : valeur $>$. Un exemple d'arbre binaire est illustré dans la figure 2.3a, pour la collection de clés $\{1, 4, 5, 6, 8, 12, 14\}$. Afin de simplifier la figure, aucune valeur n'est associée à chacune des clés.

Une opération de recherche consiste en une traversée itérative de l'arbre du noeud racine vers les feuilles. À chaque noeud sélectionné, une clé de recherche est comparée à la clé contenue dans le noeud. Si la clé de recherche est égale à la clé de recherche, alors la valeur associée est retournée et la recherche est complétée. Sinon, si la clé est de recherche est strictement

10. Cette méthode est inspirée des oiseaux coucous, qui font couver leurs oeufs par d'autres oiseaux, tout en éjectant les oeufs déjà présents dans un nid.

11. En pratique le nombre maximum d'évictions est borné [67], [68] et la table de hachage est alors considérée remplie.



(a) Arbre binaire. (b) Arbre binaire balancé. (c) Arbre B.

Figure 2.3 Exemples d'arbres binaires contenant la collection de clés $\{1, 4, 5, 6, 8, 12, 14\}$. Un noeud vide est représenté par un $\boxtimes$.

inférieure à la clé contenue dans le noeud, alors le noeud enfant gauche est sélectionné. Dans le cas contraire, le noeud droit est sélectionné. Le processus de traversée est répété en utilisant le noeud sélectionné, jusqu'à atteindre une feuille, ou un noeud vide. Pour insérer une nouvelle paire dans un arbre binaire, l'arbre est d'abord parcouru, en utilisant la clé à insérer comme clé de recherche, jusqu'à sélectionner un noeud enfant vide. Ensuite, la clé et sa valeur associée sont insérées dans le noeud enfant sélectionné.

Pour une collection de N paires, un arbre binaire a une complexité d'espace de $O(N)$ et dans le pire des cas, une complexité de recherche, d'insertion et de suppression de $O(N)$ [64], [65]. En effet, si les N clés sont insérées par ordre croissant, alors chaque noeud a un unique enfant, à droite, créant N niveaux de noeuds à traverser lors d'une opération de recherche.

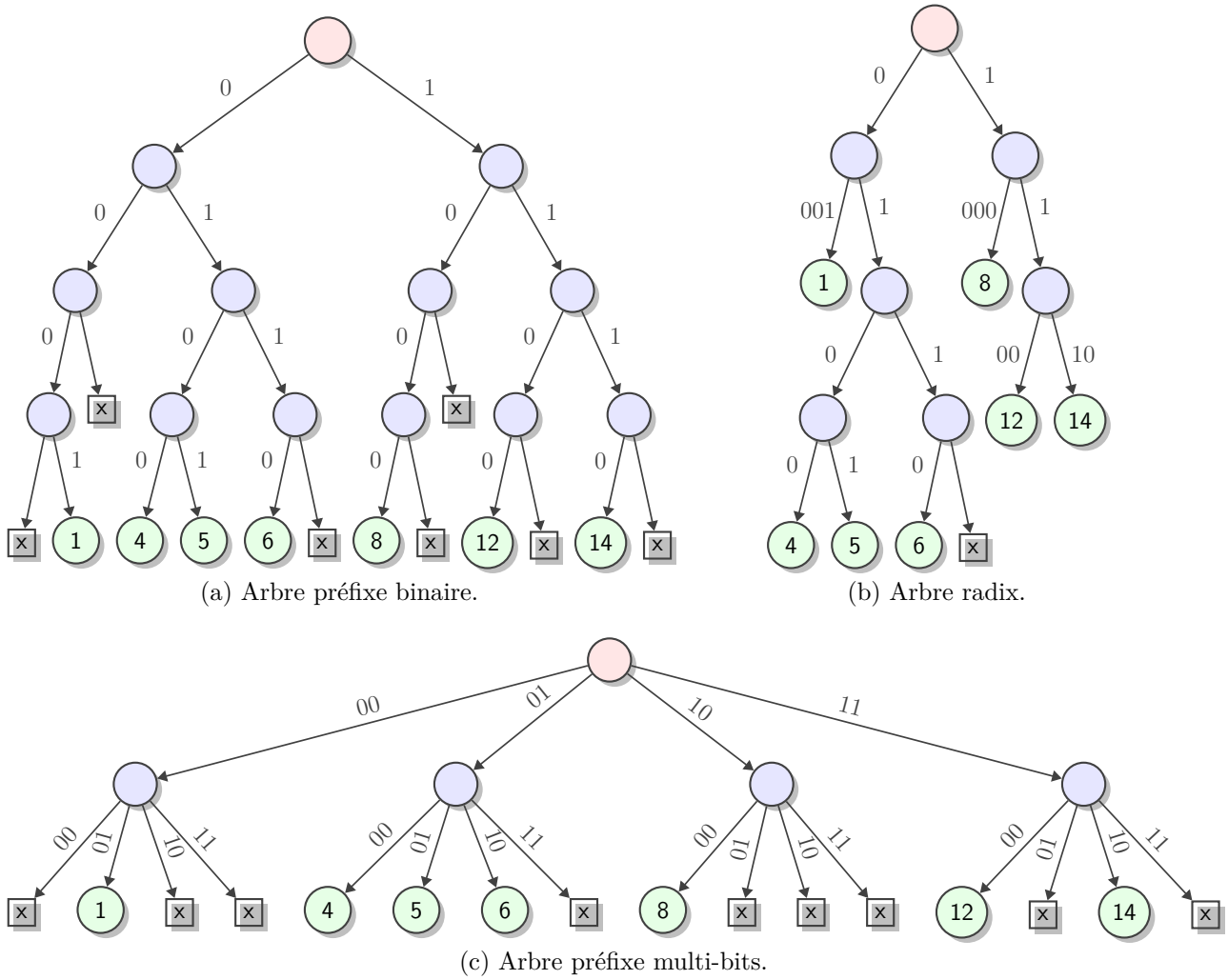
Afin de réduire la complexité de recherche à $O(log(N))$, les arbres binaires équilibrés — en anglais *balanced binary tree* — ayant des branches de profondeur équilibrées, ont été proposés [64], [65]. À titre d'illustration, un arbre binaire parfaitement équilibré est représenté dans la figure 2.3b. Toutefois, dans le cas où toutes les branches sont parfaitement équilibrées, la complexité d'insertion devient $O(N)$, car l'arbre doit être reconstruit pour respecter l'invariant sur la profondeur. Une alternative consiste à utiliser les arbres B [70], qui réduisent la complexité algorithmique de mise à jour à $O(log(N))$ [64], [70]. Un arbre B est un arbre binaire équilibré, possédant plusieurs clés par noeud. Un exemple d'arbre B est illustré dans la figure 2.3c.

Pour effectuer une opération de recherche avec un arbre binaire, l'opérateur de comparaison $<$ doit être étendu aux préfixes. Pour garantir que le plus long préfixe soit retourné lors d'une comparaison, les préfixes doivent être ordonnés par valeur d'adresse IP et par longueur de préfixes, considérant que la longueur de préfixe est inconnue lors d'une opération de recherche. Pourtant, en triant les préfixes par valeur d'adresse IP et par longueur de préfixes, il n'est alors pas possible de construire un arbre binaire équilibré lorsque des préfixes sont chevauchés. Deux techniques sont généralement utilisées pour encoder des préfixes dans des arbres binaires équilibrés. La première technique consiste à séparer les préfixes en groupes de préfixes non chevauchés [71], [72]. La seconde technique consiste à convertir les préfixes en intervalles non chevauchés d'adresses IP [73], [74]. Néanmoins, cette conversion peut dans le pire des cas transformer N préfixes en $2 \cdot N - 1$ intervalles, ce qui augmente le nombre de noeuds, et donc le coût mémoire.

Arbre préfixe. Un arbre préfixe [75], [76] — en anglais *trie* — est un graphe orienté acyclique et encodant une collection de paires $<$ clé : valeur $>$. Contrairement à un arbre binaire, un noeud ne contient pas de clé ; celle-ci est encodée par la position du noeud dans l'arbre préfixe. Un exemple d'arbre préfixe est présenté dans la figure 2.4a. Le noeud initial est appelé racine, tandis que les noeuds sans enfants sont appelés feuilles [64]. Dans un arbre préfixe *binaire* une clé de recherche est comparée à la collection de paires un bit à la fois, jusqu'à ce que l'ensemble des W bits de la clé de recherche soient évalués. La comparaison est effectuée des bits de poids fort vers les bits de poids faible et à chaque niveau de l'arbre un unique bit de la clé de recherche est évalué. Ainsi, la complexité de recherche d'un arbre préfixe binaire est de $O(W)$ pour des clés encodées sur W bits. Dans le pire des cas, chaque clé insérée dans l'arbre pourrait créer une branche unique, c'est-à-dire W noeuds. Ainsi, pour N clés, la complexité d'espace est de $O(N \cdot W)$. Lorsqu'une clé est insérée (ou supprimée), l'arbre est traversé pour identifier où insérer (ou supprimer) la valeur associée à la clé. Par conséquent la complexité d'insertion et suppression est de $O(W)$.

Un arbre préfixe peut être optimisé pour la complexité d'espace [76], [77] et pour la complexité de recherche [78], [79]. Dans le premier cas, l'arbre Radix [77] est un arbre préfixe pour lequel chaque noeud ayant un unique enfant est fusionné avec le noeud parent. L'idée générale est de comparer une séquence contigüe de bits, en lieu et place d'un seul bit, lorsqu'il existe un unique chemin dans une branche de l'arbre. Un arbre Radix permet de réduire la complexité de recherche uniquement en moyenne.

L'idée de comparer une séquence de bit est aussi utilisée par un arbre préfixe multi bits [78], [79], où k bits sont comparés à chacun des niveaux de l'arbre. Par conséquent, la complexité de recherche est réduite à $O(W/k)$, mais la complexité d'espace devient $O(N \cdot W/k \cdot 2^s)$.



(a) Arbre préfixe binaire.

(b) Arbre radix.

(c) Arbre préfixe multi-bits.

Figure 2.4 Exemples d'arbres préfixes contenant la collection de clés $\{1, 4, 5, 6, 8, 12, 14\}$. Un noeud vide est représenté par un $\boxtimes$.

Un arbre préfixe ne requiert aucune modification pour identifier le plus long préfixe associé à une adresse IP. En effet, la comparaison est effectuée sur des segments contigus de bits, en partant des bits de poids fort, vers les bits de poids faible, ce qui garantit que le plus long préfixe est sélectionné. Cependant, une problématique spécifique aux arbres préfixes multi bits est la complexité d'espace élevée, comparativement aux autres structures de données. En effet un noeud d'un arbre préfixe k bits a 2^k noeud enfants, bien que de nombreux enfants soient vides, ou associées à un même préfixe. Ainsi, les solutions adoptant un arbre préfixe proposent différentes méthodes d'encodage des noeuds, afin de minimiser le nombre de noeuds stockant la même information[80], [81].

2.3.4 Approche algorithmique

Nous présentons les solutions adoptant l'approche algorithmique et proposées pour les plateformes CPU, GPU et FPGA. Les solutions présentées ci-dessous considèrent un FIB simplifié, contenant uniquement une collection de paires < préfixe : NHI >, en lieu et place des paramètres présentés dans la section §2.2.2.

L'approche algorithmique peut être utilisée autant pour concevoir une solution pour le routage de paquets, que pour l'opération LPM dans un plan de données programmable. Néanmoins, la majorité des solutions présentées ici sont conçues pour le routage de paquets dans le réseau Internet et exploitent la connaissance d'un FIB pour augmenter la performance.

CPU

La performance des solutions logicielles est principalement limitée par la latence d'accès à la mémoire et non par le temps de calcul [82]. Par conséquent, les solutions proposées visent à minimiser le nombre d'accès mémoire, à concevoir une structure de données compacte pouvant être stockée dans la mémoire cache et à cacher la latence d'accès à la mémoire. L'ensemble des solutions présentées ci-dessous ont été conçues pour le routage de paquets dans le réseau Internet, et exploitent les caractéristiques d'un FIB pour augmenter le débit de recherches.

La solution présentée par [83] utilise une recherche dichotomique sur la longueur des préfixes. Pour chaque longueur de préfixes, les préfixes sont encodés dans une table de hachage. La méthode proposée a une complexité de recherche de $O(log(W))$ et une complexité d'espace de $O(N \cdot log(W))$. Afin de réduire en pratique le nombre d'accès mémoire, les auteurs proposent de diviser les préfixes en groupes basés sur la valeur bits de poids fort et de commencer l'opération de recherche avec les longueurs de préfixes ayant la cardinalité la plus élevée.

Plusieurs solutions alternatives proposées utilisent des arbres préfixes [81], [82], [84]-[86]. Les contributions présentées consistent principalement à réduire la consommation mémoire de l'arbre en encodant efficacement les noeuds vides, ou les noeuds associés à un même préfixe. Une solution présentée par [86] consiste à utiliser un arbre radix, pour lequel chaque noeud de l'arbre ayant un unique enfant est fusionné avec le noeud parent. Une solution alternative, le *Level Compressed Trie* [84] utilisé un noeud d'arbre préfixe multi bit uniquement si plus de la moitié des noeuds enfants sont non-vides, sinon un noeud d'arbre radix est utilisé. Kalia *et al.* [82] présentent des techniques génériques exploitant l'architecture d'un processeur pour cacher la latence d'accès mémoire et augmenter le débit de recherches. Asai et Ohara présentent Poptrie [81], qui utilise un encodage basé sur un tableau de bits — en anglais *bitmap* — permettant de spécifier les feuilles vides ou associées à un même préfixe. De plus, pour réduire la latence de l'opération de recherche, l'implémentation de Poptrie utilise une instruction de comptage de population — en anglais *Population couting (PopCount)*. Le concept de tableau de bit est aussi utilisé par l'algorithme SAIL [87] pour encoder la présence des noeuds à chacun des niveaux d'un arbre préfixe multi bit.

DXR [74], [88] est une solution basée sur un arbre binaire. Zec *et al.* [74], [88] proposent de convertir les préfixes en intervalles non chevauchés et d'effectuer une recherche par dichotomie – en anglais *binary search* — sur les intervalles. Pour réduire le nombre d'intervalles sur lesquels la recherche dichotomique est effectuée, les préfixes sont partitionnés en groupes selon la valeur des bits de poids fort et les intervalles contigus associés à un même NHI sont fusionnés.

Récemment, Rétvári *et al.* [85] ont présenté des structures de données succinctes pour l'opération LPM. Une structure de données est dite succincte lorsque sa taille est proche du minimum donné par la théorie de l'information, tout en effectuant l'opération de recherche rapidement [89]. Rétvári *et al.* démontrent que leurs solutions permettent de réduire la taille des structures de données par un ordre de grandeur par rapport à l'état de l'art.

Finalement, indépendamment de la performance des solutions proposées, le facteur limitant pour les solutions logicielles est le débit maximal pouvant être transmis au niveau des entrées et sorties, malgré l'utilisation d' environnement de développement spécialisé pour le traitement de paquets [90]-[92]. En effet, la majorité des solutions présentées ci-dessus, évaluent la performance en utilisant des traces paquets. Dans un système réel où les paquets sont transmis et reçus par une carte réseau, le débit maximum de paquets est de l'ordre de la dizaine de millions de paquets par seconde [82], [86], [90], [92].

GPU

Considérant le très grand nombre d'unité de calcul et de la taille de mémoire disponible dans un GPU, plusieurs solutions de comparaison de type LPM ont été optimisées pour des GPU [93]-[96]. De manière similaire aux solutions développées pour des CPU, les solutions visant un GPU sont conçues principalement pour le routage de paquets dans le réseau Internet, et exploitent les caractéristiques d'un FIB pour augmenter le débit de recherches.

D'un point de vue algorithmique, les solutions NBA [96] et Packet Shader [93] utilisent la solution proposée par Waldvogel *et al.* [83]. La contribution de ces deux solutions consiste à maximiser le nombre de paquets pouvant être reçus et transmis par les entrées sorties, ainsi qu'un environnement de développement pour applications réseaux sur GPU. Zhou et Prasanna [94] proposent d'utiliser un arbre multi bit à trois niveaux et les noeuds contenus à chaque niveau sont encodés dans une table de hachage. Leur approche fait levier sur la très grande quantité de mémoire disponible sur un GPU, tout en parallélisant la traversée de leur structure de données afin de maximiser le débit de recherches. La solution Gamt [95] exploite un arbre préfixe multi bits, dont l'encodage des noeuds est optimisé pour l'architecture d'un GPU. Toutefois, Kalia *et al.* [82] remettent en perspective l'utilisation de GPU pour effectuer des opérations LPM, en démontrant que des solutions faisant levier sur l'architecture d'un CPU peuvent atteindre une performance similaire à celle d'un GPU dans un environnement réel, où les paquets sont transmis au travers d'une carte réseau.

Ainsi, de manière similaire aux solutions implémentées sur CPU, la performance des solutions implémentées sur GPU est limitée par le débit de paquets pouvant être reçus au niveau des entrées sorties. Par ailleurs, pour atteindre un débit de recherches très élevé, ces solutions utilisent un traitement par lot — en anglais *batch processing* — qui contribue à augmenter la latence de traitement.

FPGA

La contrainte principale d'un FPGA est la quantité limitée de ressources logiques disponibles et plus particulièrement de mémoire intégrées sur puce — en anglais *on-chip memory*. Par conséquent, les solutions ciblant un FPGA cherchent à minimiser la taille de la structure de donnée.

L'ensemble des solutions présentées ici, à l'exception de [18], [97], ont été développées exclusivement pour le routage de paquets IPv6 dans le réseau Internet et nécessitent de connaitre le FIB à encoder pour minimiser la consommation de ressources logiques d'un FPGA. À l'inverse, les solutions de Xilinx [97] et de Abdelhadi *et al.* [18] n'ont pas besoin de connaitre

par avance le FIB à encoder et peuvent donc être utilisée comme bloc de base dans un plan de données programmable.

Bando *et al.* [69] présentent une solution basée uniquement sur des tables de hachage, où les préfixes sont convertis vers un ensemble restreint de longueurs de préfixes, ce qui augmente la consommation de ressources logiques.

Des méthodes basées sur des arbres préfixes ont été aussi proposées [80], [98]. La méthode proposée par Tree bitmap [80] consiste à encoder dans un tableau de bits la validité des noeuds contenus dans k niveaux d'un arbre préfixe. Une alternative proposée par Bando *et al.* [98] consiste à réduire la taille du tableau de bit en adoptant un encodage compact. De plus, afin de réduire la profondeur de la structure de données, une table de hachage est utilisée pour effectuer une recherche sur les bits de poids fort, tandis que les bits de poids faible sont comparés à l'aide d'un arbre préfixe utilisant un encodage compact des noeuds. Afin de minimiser l'empreinte mémoire d'un arbre préfixe multi bit, Rétvári *et al.* [85] propose une méthode de «repliement»— en anglais *folding* — des branches associé à un même NHI, permettant de réduire l'empreinte mémoire d'un ordre de grandeur par rapport aux solutions existantes. Plusieurs travaux ont proposé d'utiliser des arbres binaires [71], [72], [97]. La solution de Xilinx utilise un arbre binaire équilibré [97], nécessitant de transformer les préfixes en intervalles non chevauchés. Afin de réduire la profondeur de la structure de données à traverser lors d'une opération de recherche, des arbres B sont utilisés dans Layered Trees [72] et la solution présentée par Le et Prasanna ă[71]. Les contributions consistent à minimiser le nombre de préfixes après conversion d'un FIB en groupes de préfixes disjoints. Dans Layered Trees [72] les préfixes sont *pelés* en couches de préfixes disjoints. Le nombre de groupes n'est pas contraint, afin d'éviter d'augmenter le nombre total de préfixes après le partitionnement. Le et Prasanna [71] proposent une méthode de partitionnement des préfixes lorsque le nombre total de groupes est contraint. Leur méthode utilise un algorithme de programmation dynamique pour minimiser le nombre total de préfixes après partitionnement.

2.4 Conclusion

Dans un centre de données, l'opération LPM ne peut être implémentée en logiciel sur des CPU ou des GPU pour deux raisons. Premièrement, le trafic pouvant être acheminé vers le processeur à partir des entrées/sorties est actuellement limité à la dizaine de Gb/s. Deuxièmement, les opérateurs de centre de données cherchent à maximiser la puissance de calcul CPU pouvant être louée à des clients. Par conséquent, les FPGA, de par leur flexibilité, apparaissent comme une plateforme idéale pour accélérer des traitements réseau, incluant l'opération LPM. Sur FPGA, l'opération LPM peut être implémentée en utilisant l'approche

mémoire transposée, ou l'approche algorithmique. Comme nous le montrerons dans le chapitre 5, les solutions utilisant approche mémoire transposée offrent une efficacité très inférieure aux solutions adoptant l'approche algorithmique. Néanmoins, les solutions adoptant l'approche algorithmique souffrent d'un débit limité ou d'une consommation de ressource logique élevée, comme nous le présenterons dans les chapitres 3 et 5. Finalement, la majorité des solutions proposées ont été conçues exclusivement pour le routage de paquets IPv6 dans le réseau Internet, en exploitant les caractéristiques des préfixes contenus dans un FIB. Par conséquent, ces solutions ne peuvent être utilisées dans un plan de données programmable. Ainsi, il est nécessaire de concevoir des solutions efficaces pour effectuer l'opération LPM sur FPGA, que cela soit dans le contexte spécifique de routage, ou pour la mise en oeuvre d'un plan de données programmable.

CHAPITRE 3 ARTICLE 1 - SHIP : A SCALABLE HIGH-PERFORMANCE IPv6 LOOKUP ALGORITHM THAT EXPLOITS PREFIX CHARACTERISTICS

Autheurs : Thibaut Stimpfling, Normand Belanger, J.M. Pierre Langlois et Yvon Savaria.

Publié dans : IEEE/ACM Transactions on Networking 27, numéro 4 (2019).

Abstract Due to the emergence of new network applications, current IP lookup engines must support high bandwidth, low lookup latency and the ongoing growth of IPv6 networks. However, existing solutions are not designed to address jointly these three requirements. This paper introduces SHIP, an IPv6 lookup algorithm that exploits prefix characteristics to build a data structure designed to meet future application requirements. Based on prefix length distribution and prefix density, prefixes are first clustered into groups sharing similar characteristics, and then encoded in hybrid trie-trees. The resulting memory-efficient and scalable data structure can be stored in low-latency memories and allows the traversal process to be parallelized and pipelined in order to support high packet bandwidth in hardware. In addition, SHIP supports incremental updates. Evaluated on real and synthetic IPv6 prefix tables, SHIP has a logarithmic scaling factor in terms of the number of memory accesses, and a linear memory consumption scaling. Compared to other well-known approaches, SHIP reduces the required amount of memory per prefix by 87%. When implemented on a state-of-the-art FPGA, the proposed architecture can support processing 588 million packets per second.

3.1 Introduction

Global IP traffic carried by networks is continuously growing [99]. To handle this increasing Internet traffic, network working groups have ratified the 100-gigabit Ethernet standard, and are studying the 400-gigabit Ethernet standard. As a result, network nodes have to process packets at those line rates, which requires IP address lookup engines to process an IPv6 packet in less than 6 ns [71].

The IP lookup task consists of identifying the next hop information NHI to which a packet should be forwarded. The lookup process consists of matching the destination IP address, extracted from the packet header, against a list of entries stored in a lookup table, called the Forwarding Information Base FIB. Each entry in the FIB, a prefix, represents a network

defined by its prefix address and its prefix length, which represents the number of valid prefix bits. While a destination IP address may match multiple entries in the FIB, only the NHI associated with the longest prefix matched is returned [100].

IP lookup algorithms and architectures tailored for IPv4 technology are not performing well with IPv6 [71], [101], due to the fourfold increase in the number of bits in IPv6 addresses over IPv4. Thus, dedicated IPv6 lookup methods are needed to support upcoming IPv6 traffic.

IP lookup engines must be optimized for high bandwidth, low latency, and scalability for two reasons. First, due to the convergence of wired and mobile networks, many future applications require high bandwidth and low latency at the same time, such as virtual reality, remote object manipulation, eHealth, autonomous driving [17]. Second, IPv6 FIBs are expected to grow as IPv6 technology is still being deployed [102], [103]. However, current solutions presented in the literature are not jointly addressing these performance requirements.

In this paper, we introduce SHIP : a Scalable and High Performance IPv6 lookup algorithm designed to meet current and future performance requirements. SHIP is built around the analysis of prefix characteristics. Three main contributions are presented :

— Two-level prefix grouping that clusters IPv6 prefixes in groups sharing common characteristics. Prefixes are divided into bins based on their 23 most significant bits, while the bins are stored in a hash table. Within each bin, prefixes are sorted in groups, based on the FIB prefix length distribution.

— An hybrid trie-tree (HTT), a shallow and memory efficient data structure. Each group of prefixes is encoded in an HTT. The HTT revisits the concept of multi-bit trie, but adapts the number of nodes built at each level on the prefix density. At the bottom of the trie, multiple leaves are encoded into a leaf bucket.

— A high-throughput implementation of SHIP targeting Field Programmable Gate Arrays (FPGAs). The implementation is based on an that performs a pipelined traversal of the SHIP data structure, and leverages on-chip memory to support a high lookup rate, while balancing the lookup latency.

In reported characterization results, SHIP stores 580 k prefixes and the associated NHI using less than 5.2 MB of memory. Evaluated on multiple benchmarks, SHIP achieves a linear memory consumption scaling and a logarithmic latency scaling. SHIP also supports incremental updates. In addition, when implemented on FPGA, the architecture can support a high-throughput of 588 million packets per second.

The remainder of this paper is organized as follows. Section 3.2 introduces common approaches used for IP lookup. Section 3.3 gives an overview of SHIP. Then, two-level prefix

grouping is presented in Section 3.4. The proposed hybrid trie-tree is covered in Section 3.5. Section 3.6 presents a method to update the SHIP data structure and its cost. Section 3.7 introduces the method and metrics used for performance evaluation and Section 3.8 presents simulation results, while Section 3.9 presents FPGA implementation results. Section 3.10 compares SHIP performance with other methods. Lastly, we conclude the work by summarizing our main findings and results in Section 3.11.

3.2 Related Work

Traditionally, IP lookup is implemented using TCAMs, a specialized hardware with $O(1)$ lookup time complexity. A TCAM is an associative memory that matches a key simultaneously against all prefixes. However, TCAMs combine a high power consumption and a high cost, making them unattractive for routers holding a large number of prefixes [98], [104]. TCAM implementations on FPGAs also yield a relatively poor performance [18].

As a result, algorithmic solutions have been explored as an alternative to TCAMs. Algorithmic solutions emulate the TCAM functionality using data structures. Four main types of data structures are used by algorithmic solutions : hash tables, Bloom filters, tries and trees. The challenge is to encode efficiently prefixes, which are loosely structured and with a highly nonuniform prefix length distribution. In addition, for any given prefix length, prefix density ranges from sparse to very dense, where the prefix density represents the sparseness of non-empty nodes within a level of a trie.

Interest for using a hash table is twofold. First, a hash function aims at distributing uniformly a large number of keys over bins independently of the key structure. Second, a hash table provides $O(1)$ lookup time on average and $O(N)$ space complexity, where N is the number of prefixes. However, a pure hash-based LPM solution has a lookup time complexity of $O(W)$, where W is the number of distinct prefix lengths, as one hash table is probed per prefix length. Bando *et al.* [69] proposed to expand prefixes to few prefix lengths, at the cost of an increased memory consumption. Waldvogel *et al.* [83] proposed a binary search on prefix lengths to reduce the lookup time complexity to $O(log(W))$ with a space complexity of $O(N \cdot log(W))$. Still, a hash function can generate collisions that degrades performance. To reduce the number of collisions, a method that exploits multiple hash tables [69], [105] was proposed. This method divides the prefix table into groups of prefixes, and selects a hash function to minimize the number of collisions within each prefix group [69], [105]. The solution presented by Zhou et Prasanna [94] leverages perfect hash functions, which are guaranteed to be collision-free. Although their solution achieves a very high throughput when implemented on GPU, a large memory space is required and no incremental update support is provided.

Bloom filters, space-efficient probabilistic data structures, have also been used in the literature to select a set of prefixes that may match an IP address [106]-[109]. Recent studies have shown that Bloom filters can significantly reduce the average lookup time [106], [108], [109]. However, by design, this data structure generates false positives independent of the configuration parameters used. Thus, a Bloom filter can lead to poor performance in the worst case, when many prefix sets are selected.

B-trees, generalized self-balancing binary search trees, have also been explored [71], [72], [74], [88], [101]. Such data structures are tailored to store loosely structured data such as prefixes, as their time complexity of $log(N)$ and storage complexity of $O(N)$ are independent from the prefix distribution characteristics. However, B-trees can only be used with non-overlapped prefixes, i.e. disjoints two by two. Because converting prefixes into non-overlapped prefixes increases the number of prefixes, previous works focused on minimizing the prefix growth.

Le et Prasanna divide a FIB into disjoint prefix groups using dynamic programming [71], and build a B-tree per group. The FPGA implementation requires several external memories due to a low memory efficiency, which increases lookup latency. Chang *et al.* [72] proposed LayeredTrees, where the FIB is peeled iteratively into layers of disjoint prefixes. However, the throughput supported by their architecture is similar to previous works [71]. A variation of a balanced binary search tree was introduced by Zec et Mikuc [74], [88]. Their solution is tailored for IPv4, and lead to performance degradation when adapted to IPv6 [81].

k-bit trie data structures are attractive because k bits of the IP address are compared at a time, and thus, they have a $O(W/k)$ time complexity, where W is the IP address size. However, the low time complexity of k-bit tries comes at a large storage complexity of $O(2^k N \cdot W/k)$, which lead to very low memory efficiency when the trie is built with unevenly distributed prefixes [79], [110]. As a result, significant efforts were dedicated to reducing the memory footprint.

The level compression trie (LC-trie) [84] technique was proposed to combine multi-bit trie for dense regions with a level compressed binary trie for sparser regions [77]. Bando *et al.* proposed the tree bitmap structure, where k levels of a 1-bit trie are encoded into a bitmap [80]. The PC-trie, an improved tree bitmap, was proposed in the FlashTrie architecture [98]. However, the FlashTrie architecture requires multiple external memories, leading to a high lookup latency.

Bitmap solutions were also explored by Yang *et al.*, who proposed the splitting approach to IP lookup (SAIL) [87]. SAIL decomposes the lookup into the identification of the prefix length, and then, the identification of the NHI. However the SAIL performance was only evaluated with the data structure used for the prefix length identification. Asia and Ohara

presented Poptrie [81], a CPU-optimized implementation of a k-bit trie. Poptrie encodes each node of a k-bit trie using a bitmap, and merges leaves covering the same prefix. Because PopTrie is tailored for IPv4 prefixes, the performance is reduced when using IPv6 prefixes.

Another solution proposed by Luo *et al.* [111] combines TCAM and k-bit trie to reduce the number of entries stored in the TCAM. Still, this solution is unattractive as most of the prefixes are stored in the TCAM.

GAMT [95] leverages the large amount of memory available on GPUs to implement a k-bit trie. This GPU-based solution supports a very high lookup rate, but suffers from a very high latency. Unoptimized k-bits tries were also evaluated on CPU, but were shown to provide a limited lookup rate [86], [87] even when low-level CPU optimizations were used [82].

Recently, a new approach exploiting information-theoretic and compressed data structures were proposed to compress a trie, yielding very compact data structures [85], but only achieving a low lookup rate.

Multiple LPM solutions were implemented in software, as the high-operating frequencies of CPUs or GPUs can be leveraged to support very high lookup rates. However, from a system point-of-view, the lookup rate of software LPM solutions is limited by the rate supported by the packet I/O framework such as netmap [90], or DPDK [91]. Indeed, these I/O frameworks cannot forward at very high-rates packets received from the network interface card (NIC) to the main memory [82], [86], [90], [92]. In addition, most solutions were evaluated with packets already stored in the memory [74], [81], [87], [88], [94], [95]. Lastly, software LPM solutions typically uses batch processing techniques to increase the bandwidth at the expense of latency [82], [90], [92], [94], [95].

In summary, the solutions optimized for software implementation were not shown to support high lookup rates in complete systems. In addition, the solutions targeting hardware implementation generally suffer from a low throughput, or from low memory efficiency.

3.3 SHIP Overview

SHIP comprises a procedure to build an efficient data structure, and a procedure to traverse it, namely the lookup algorithm.

SHIP combines prefix clustering methods with memory efficient data structures. A clustering method, two-level prefix grouping, divides prefixes into address block bins (ABB) based on their 23 most significant bits (MSBs), and further divides prefix into prefix length sorted (PLS) groups based on the FIB prefix length distribution. The address block bins are recorded in a hash table, while the prefixes in each prefix length sorted group are encoded in an HTT.

The SHIP data structure is illustrated in Fig 3.1. An N-entry hash-table holds M valid address block bins (ABBs) obtained after dividing the prefixes on their MSBs. Each valid ABB holds a pointer to a set of K prefix length sorted (PLS) groups, encoded in HTTs.

The lookup algorithm identifies the NHI associated to the longest prefix matched. First, the MSBs of the destination IP address are hashed to select an ABB pointer. In this example, the m-th ABB is selected in the hash table. The selected ABB points to a set of K HTTs represented with the dashed rectangle. Second, using the least significant bits of the destination IP address, the selected HTTs are traversed in parallel. Because each of the HTT can hold a prefix matching the IP address, a priority resolution module is used to select the NHI associated with the longest prefix.

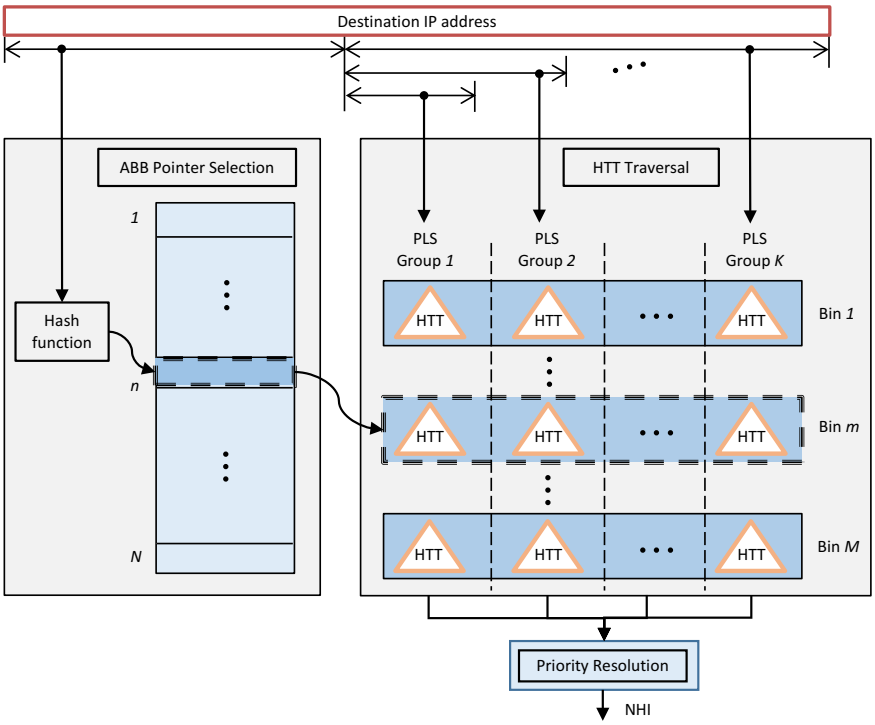


Figure 3.1 SHIP two-level data structure organization with M address block bins and K prefix length sorting groups.

3.4 Two-level Prefix Grouping

Two-level prefix grouping is a clustering method that divides prefixes in ABBs, and further sorts prefixes into prefix length groups. This clustering method is proposed to divide a prefix table in groups, each holding a fraction of the prefixes. The prefix distribution within groups is then leveraged by the HTT, as it will be shown in section 3.8.

3.4.1 Address Block Binning

Prefixes are binned in ABBs based on their 23 most significant bits (MSBs), and ABBs are recorded in a hash table.

The motivation to bin prefixes on 23 MSBs relates to the known IPv6 address space allocation. IPv6 prefixes are allocated from a pool of prefix blocks managed by the Internet Assigned Numbers Authority (IANA) ranging from /12 to /23 [112]. Because few prefix blocks are assigned [112], when prefixes are binned on their 23 MSBs, a small number of bins are created. As a consequence, the number of prefixes in bins is reduced by up to two orders of magnitude over the original prefix table size (see section 3.8.1).

The ABB method leverages a perfect hash table [113] to store the bin values for two reasons. First, the bin values are almost static because they represent address spaces allocated to regional internet registries that are unlikely to be updated on a short time scale. Second, perfect hash functions guarantee an $O(1)$ time complexity as no collisions are generated.

While the idea of using a hash table or direct index table to do a lookup on the MSB is not new for IPv4 [74], [81], [83], [110], the ABB method is optimized for IPv6 and differs from previous works by leveraging the known allocation of the IPv6 address space.

Because prefixes associated to an ABB can overlap, the prefix length sorting method is introduced to reduce the number of overlapping prefixes.

3.4.2 Prefix Length Sorting

PLS divides the prefixes associated with an ABB into groups based on the IPv6 prefix length distribution. This method sorts prefixes with length /24 to /64 into K groups. Each group cover a contiguous range of prefix lengths.

The prefix length range covered by each group is selected based on two principles. First, when a prefix length accounts for a large percentage of the total number of prefixes, the prefix length is used as an upper bound of the considered group. Second, prefix length ranges must be chosen such that the K groups are as balanced as possible in terms of the number of prefixes.

To illustrate those two principles, an analysis of prefix length distribution using a real prefix table [103] is presented in Fig. 3.2. The first 23 prefix lengths are omitted in Fig. 3.2, as the ABB method already bins prefixes based on their 23 MSBs. It can be observed in Fig. 3.2 that the prefix lengths with the largest cardinality are /32 and /48 for this example. Applying the two principles of prefix length sorting to this example, the first group covers prefix lengths

from /24 to /32, and the second group covers the second peak, from /33 to /48. Finally, all remaining prefix lengths, from /49 to /64 are left in the third prefix length sorting group.

The first principle aims at minimizing the number of prefix overlaps inside a prefix group, by isolating a large number of prefixes from longer prefixes that can overlap. The second principle aims at balancing as much as possible the number of prefix overlaps between PLS groups, in order to obtain HTTs with relatively similar characteristics.

The prefix distribution within groups after appliyng the two-level prefix grouping method is presented in section 3.8.1.

3.5 Hybrid Trie-Tree data structure

The proposed HTT encodes the prefixes held in each non-empty PLS group. The HTT data structure is tailored to adapt its *shape* to the characteristics of the prefixes used. An HTT combines a density-adaptive trie (DAT), a memory efficient multi-bit trie, and a leaf bucket (LB), which derives from a tree leaf.

3.5.1 Density-Adaptive Trie

The proposed density-adaptive trie revisits the concept of the multi-bit trie. It does so by *adapting* the number of nodes created based on the prefix density, which refers to the sparseness of non-empty nodes within a trie level, and to the prefix replication factor on contiguous nodes within a trie level.

Similarly to a k-bit trie, a DAT is applied iteratively on sub-tries, extracted from a binary trie holding a prefix set. A DAT encodes the leaves of a sub-trie after pushing prefixes to the leaves, as shown in Fig. 3.3 with the dashed arrows. The method to extract sub-tries from a

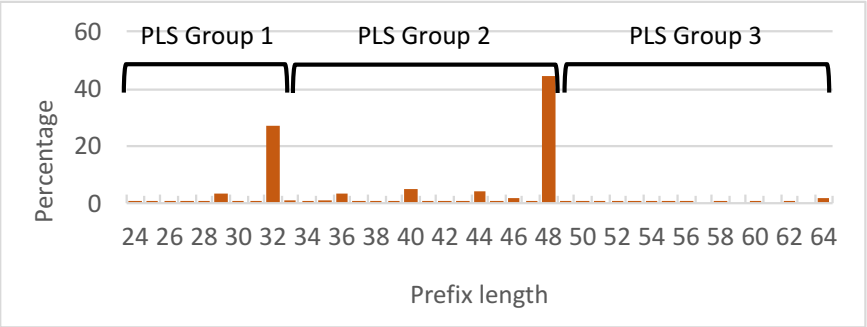


Figure 3.2 The uneven prefix length distribution of a real prefix table used by the PLS method to create 3 PLS groups.

binary trie is presented in section 3.5.3.

The main idea behind a DAT is to merge contiguous empty leaves or leaves recording the same prefix, as shown in Fig. 3.3a, and encode them in a single merged node, as shown in (Fig. 3.3b). By contrast, a k-bit trie encodes each sub-trie leaf into a node, as illustrated in Fig. 3.3a.

To determine if two contiguous leaves can be merged, the number of prefixes *covered* by each leaf is evaluated, which includes the prefix held in the leaf and the number of prefixes held in the branch down the leaf. Two leaves are merged when the number of prefixes covered by the merged node is either (1) smaller than a fixed threshold, namely the size of a leaf bucket (see 3.5.2), or (2) is not higher than the largest number of prefixes covered by each of the two leaves.

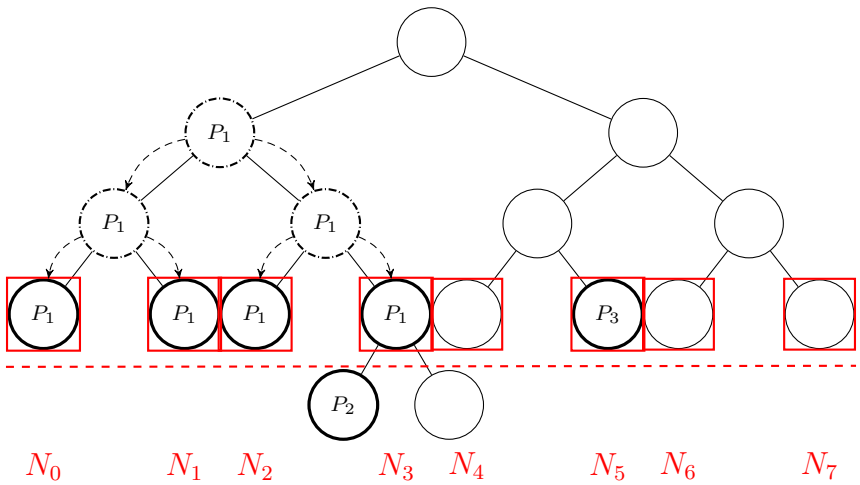
The merging method is applied from the leaves at both edges of a sub-trie, toward the center. For instance, in Fig. 3.3a, the merging method starts with the leaves associated to nodes N_0 and N_7. For both directions, this method evaluates whether a leaf can be merged with the next contiguous leaf. The merging process is repeated until a leaf can no longer be merged with its next contiguous leaf. Otherwise, the method is repeated from the last leaf that was left unmerged.

The merging method has two constraints with respect to the number of merged leaves. First, a merged node encodes only a number of contiguous leaves that is a power of two, as the space covered by the merged nodes is represented using the prefix notation. Second, the total number of merged nodes is bounded by the adaptive trie node size.

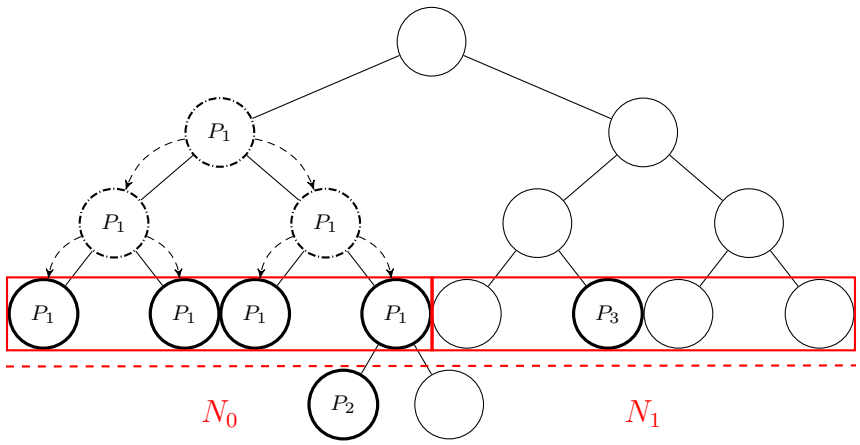
The leaf indices evaluated (merged or not) by the merging method are encoded in the $LtoH$ and $HtoL$ arrays. The $LtoH$ and $HtoL$ arrays hold the index of leaves traversed from low to high indices, and high to low indices, respectively. All leaves not evaluated by the merging method are left unmerged, and are said to be encoded in a unmerged zone.

The benefits of the merging method used in a DAT is illustrated in Fig. 3.3, where the number of nodes stored in memory after encoding a sub-trie is reduced from 8 using a 3-bit trie in Fig. 3.3a down to 2 using a DAT, as illustrated in Fig. 3.3b. With a DAT, both contiguous leaves storing prefix P_1 are merged, but also contiguous empty leaves.

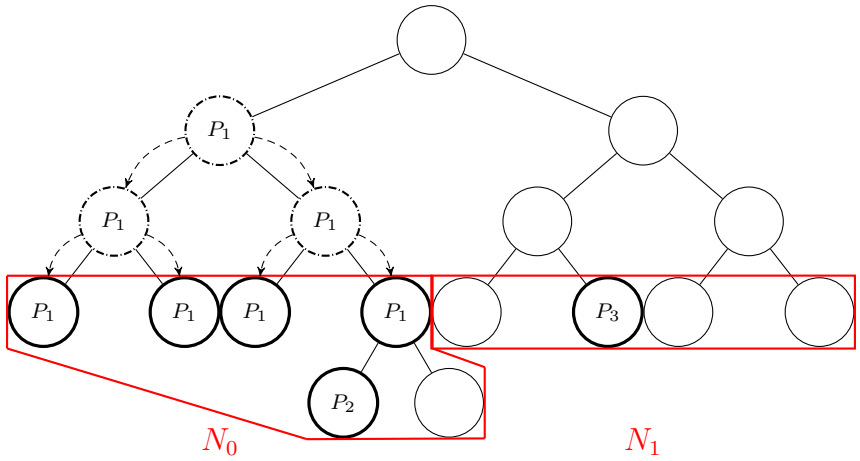
A DAT is tailored for memory efficiency by merging contiguous nodes recording the same information. However, in Fig. 3.3b, a DAT requires another level to separate the prefix P_2 from prefix P_1. Hence, to reduce the depth of a DAT, when the number of prefixes covered by a node is below a threshold value b, prefixes are encoded in a leaf bucket.



(a) 3-bit trie : child nodes N_0 to N_7 are created to encode the sub-trie above the red line. The child nodes to encode the remaining sub-tries are not shown.



(b) DAT : N_0 and N_1 are created to encode the sub-trie above the red line. The child nodes to encode the remaining sub-tries are not shown.



(c) HTT (DAT with LBs) : two child nodes N_0 and N_1 are created to encode the *complete* trie, assuming that each LB can store up to $b = 2$ prefixes.

Figure 3.3 Reducing the number of child nodes N_i stored in memory for a 3-bit trie (a) using a Density-Adaptive Trie (DAT) (b) combined with Leaf Buckets (LB) (c). Root nodes are not shown. The prefix set used is given in Fig. 3.1.

Tableau 3.1 An example of prefix table.

Prefix	**Value**	**NHI**
P_1	0/1	NHI_1
P_2	0110/4	NHI_2
P_3	101/3	NHI_3

3.5.2 Leaf Bucket

A leaf bucket (LB) stores a set of up to b distinct prefixes and their NHI, *covered* by a DAT node and in the branch down the DAT node. For instance, in Fig. 3.3b, the DAT node N_0 covers the prefixes P_1 and P_2.

The proposed LB derives from a tree leaf, but only the prefix bits left unmatched are stored. The number of unmatched bits for each prefix is encoded in the LB, as well as the number of prefixes stored. By reducing the amount of information stored, the proposed LB improves the memory efficiency over a tree leaf and requires fewer memory accesses to be read. The interest of using an LB to encode prefixes is two-fold. When a sub-trie holds up to b sparsely distributed prefixes at the leaves, an LB requires fewer nodes to store the prefixes compared to a DAT. In addition, because most PLS groups hold very few prefixes (see section 3.8.1), a LB can encode in a single node a PLS group.

In Fig. 3.3c, DAT nodes *covering* two or fewer prefixes are encoded in LBs. Hence, using an HTT, which combines a DAT with LBs, the prefix set shown in Table 3.1 is encoded in a DAT root node with two child nodes LB_0 and LB_1, as shown in Fig 3.4. The content of each LB is also presented in Fig. 3.4.

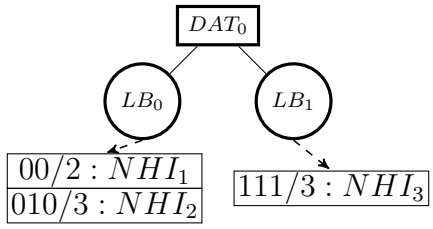


Figure 3.4 Complete HTT for the prefix table presented in Fig. 3.1 : a root node DAT_0 and two child nodes LB_0 and LB_1. The prefixes held in each leaf bucket LB are also shown.

3.5.3 HTT Build Procedure

The HTT build procedure is split into two parts : the main procedure and the selection of a sub-trie.

Main Procedure

The main procedure is initiated at the root node of a binary trie. If the number of prefixes stored in the binary trie is below a fixed threshold b, prefixes are encoded in a LB. Otherwise, an algorithm (see 3.5.3) iteratively selects from the binary trie sub-tries, i.e. binary tries carved out from the main binary trie, that are encoded into HTT nodes.

For a selected sub-trie, the DAT merging method is applied on the leaves. Then, when a DAT node *covers* up to b prefixes, the prefixes are encoded in a LB. Otherwise, a DAT node is created and the process is repeated on each non-empty branches of the sub-trie.

An illustration is given in Fig. 3.3c and 3.4 for the prefix set shown in Table 3.1. Prefixes are inserted in a binary trie, shown in Fig. 3.3a. Then, an algorithm, presented in Section 3.5.3), selects from the binary trie a sub-trie to be encoded in HTT nodes. In Fig. 3.3a and 3.3c, the selected sub-trie is above the dashed red line. The sub-trie leaves are encoded in HTT nodes. The process is then repeated for each sub-trie below the red dashed line.

Selection of a Sub-Trie

A greedy algorithm [114], presented as algorithm 1, is used to extract at each iteration a sub-trie from which a memory-efficient and shallow HTT is built.

The greedy algorithm extracts from a binary trie the deepest sub-trie respecting a memory consumption constraint. Because a deep sub-trie is selected at each iteration, a data structure with few levels is built, which in turns requires few memory access to traverse.

Starting from a binary trie node used as the root of the selected sub-trie, the algorithm increases iteratively the sub-trie depth until a memory consumption constraint is violated.

The space measurement factor ($Smpf$), used as a memory consumption constraint, is evaluated as the number of prefixes *covered* in the selected sub-trie and its branches multiplied by a constant, set here to 8 (line 1). Using a higher constant value favors the selection of a deeper sub-trie, at the cost of higher memory consumption. The constant used here was selected experimentally to provide a good trade-off between the sub-trie depth and the memory consumption.

At each iteration, a space measurement (Sm) function estimates the memory consumption

of the selected sub-trie by counting the number of prefixes held at the leaves and held in the branches ($Num_{Prefixes}(leaf_j)$), to which is added a penalty term discussed below. The number of prefixes held at the leaves is evaluated after pushing the prefixes to the leaves of the sub-trie.

The heuristic selects a sub-trie enclosing the average prefix length of the branch. If the average prefix length in the selected branch is higher than the prefix lengths covered by the sub-trie, Sm will grow at a slow rate, and a deeper sub-trie will be selected. To avoid the selection of a very deep sub-trie when Sm remains unchanged for many iterations, a penalty term is added to Sm, and this penalty is defined as the sum of $Sm(i-1)$ and the number of leaves of the selected sub-tries.

Algorithm 1 : Heuristic to select the depth of a sub-trie to be encoded into an HTT node.

Input : Branch of a binary trie
Output : Depth of the selected sub-trie (D)

1 $Smpf = Num_{Prefixes} \cdot 8; Sm_0 = 0; i = 1;;$
2 **do**
3 $\quad D_i = i;;$
4 $\quad Sm_i = \sum_{j=0}^{D_i} Num_{Prefixes}(leaf_j) + 2^{D_i} + Sm_{i-1};;$
5 $\quad i = i + 1;;$
6 **while** $Sm_i \leq Smpf$;
7 **return** D_i

3.5.4 HTT Lookup Procedure

The HTT lookup algorithm starts with a traversal of the density-adaptive trie until a leaf bucket is reached. Then, the prefixes held in a leaf bucket are matched against a destination IP address, and the NHI of the longest matching prefix is returned.

Density-Adaptive Trie Traversal

When a DAT node is read from memory, the sub-trie encoded in the DAT node must be decoded to compute the address of the child node to read next. Using the *LtoH* and *HtoL* arrays stored in a DAT node, the sub-trie merged leaves can be identified.

In a DAT, the one-to-one mapping between a sub-trie leaf and a node illustrated in Fig. 3.3a for a k-bits trie does not hold because of the merging method, as shown in Fig. 3.3b and 3.3c. As a result, the memory location of a child node is a function of the number of merged leaves

preceding the selected child node. The procedure to computed the child node address is presented in algorithm 2.

Algorithm 2 : Identification of the matched child node address

Input : Child base address, IP address segment IP_{seg}, $LtoH$ and $HtoL$ arrays
Output : Child node address

```
1  L = |LtoH| ;
2  if IP_seg ≤ LtoH[L − 1] then                    /* Matched node in LtoH zone? */
3  |  p_LtoH = (i ∈ [0; L − 2] | LtoH[i] ≤ IP_seg ≤ LtoH[i + 1]) ;
4  |  offset = p_LtoH ;
5  else
6  |  if IP_seg ≥ HtoL[0] then                    /* Matched node in HtoL zone? */
7  |  |  p_HtoL = (i ∈ [1; L − 1] | HtoL[i − 1] ≤ IP_seg ≤ HtoL[i]) ;
8  |  |  offset = p_HtoL + HtoL[0] − LtoH[L − 1] + L − 1 ;
9  |  else                                  /* Matched node in a non-merged zone? */
10 |  |  offset = IP_seg − LtoH[L − 1] + L − 1 ;
11 |  end
12 end
13 return Child node address = base address + offset
```

The address of the child node to visit next is computed in two steps. First, the zone holding the child node ($LtoH$ array, $HtoL$ array, or the unmerged zone) is identified as well as the child node index in the selected zone. Second, the number of merged leaves prior to the matched node index is evaluated to derive its address.

First, a segment of the IP address is extracted to select a sub-trie leaf which index is equal to the IP address segment. Both $LtoH$ and $HtoL$ arrays are used to identify whether the selected leaf index is covered by a merged node in the $LtoH$ zone, the $HtoL$ zone or the unmerged zone (lines 2, 6 and 9). In addition, if the $LtoH$ zone or $HtoL$ zone is selected, the array index of the merged node that contains the selected leaf index is identified (lines 3 and 7). The array index, in the $LtoH$ or $HtoL$ array is noted p_{LtoH} and p_{HtoL}, respectively. If the matched node is held in the unmerged zone, the array index is the IP address segment.

Second, using the number of merged leaves preceding the child node index, the child node offset is derived. If the child node is in the $LtoH$ zone, the offset is given directly by p_{LtoH} (line 4). In a unmerged zone, the number of merged leaves, computed using the $LtoH$ array, is subtracted from the IP address segment to obtain the child node offset (line 10). In the $HtoL$ zone, the number of merged leaves, evaluated using both the $LtoH$ and $HtoL$ arrays, is subtracted from the child node index p_{HtoL} (line 8). Lastly, the offset is added to the children base address (line 13).

Algorithm 2 is illustrated in Fig. 3.5 for the case $L = 3$ and $IP_{seg} = 10$. The IP address segment matches a node held in the $HtoL$ zone, as $IP_{seg} \geq HtoL[0]$ (line 6). The node matched within the $HtoL$ zone is stored at the array index $p_{HtoL} = 1$ (line 7) as $9 \leq IP_{seg} \leq 10$ (line 7). The number of merged leaves up to the matched node is computed using both the $HtoL$ and $LtoH$ arrays (line 8). Based on the $LtoH$ array, the number of merged leaves is $LtoH[L-1] - (L-1) = 1$ (line 9). As $p_{HtoL} + HtoL[0] = IP_{seg}$, no leaves are merged in the $HtoL$ zone preceding the matched node. Thus, the offset of the matched child node is $IP_{seg} - 1 = 9$.

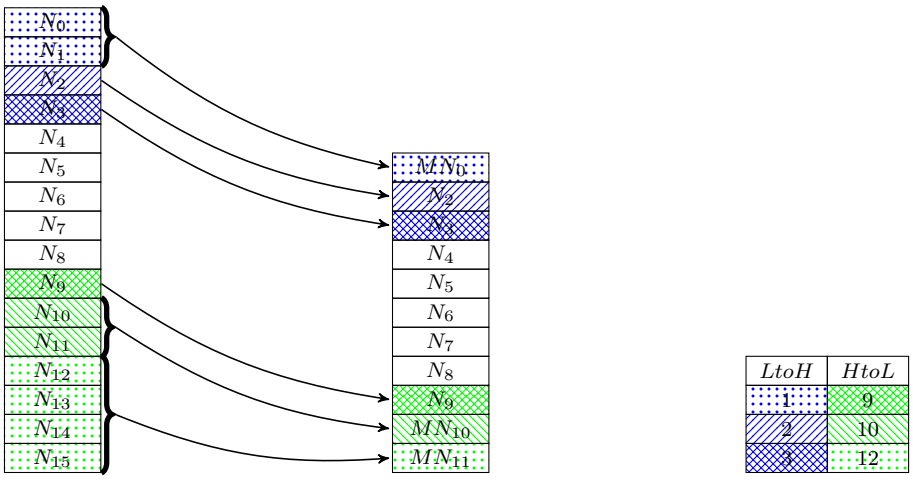


Figure 3.5 Node indices before merging (a), and after merging(b), stored in the $LtoH$ and $HtoL$ arrays (c). MN stands for Merged Node

Leaf Bucket Matching

The DAT is traversed until a leaf bucket is reached. The leaf is first parsed, and then prefixes are read. Next, all prefixes are matched against the destination IP address, and their prefix length is recorded if matches are positive. When all the prefixes are matched, only the longest prefix match is returned with its NHI.

3.6 Update Support

First, an analysis of the updates characteristics is presented, then the cost of updates using the SHIP algorithm is evaluated.

Similar to other works, an offline procedure identifies the nodes of the SHIP data structure to modify [81], [87], [98], [115] upon receiving a prefix update.

3.6.1 Analysis of the Prefix Updates

Three types of prefix updates can be applied on a FIB ; NHI modification of an existing prefix, prefix insertion, prefix deletion. The updates received by the RIS remote route collector *rrc00* [103], from 2017/12/1 to 2017/12/2 included, are shown in Fig. 3.6. Based on Fig. 3.6, peaks of 2000 NHI modifications per second can be observed, whereas the peak prefix insertion and deletion rates are relatively similar at around 550 updates per second.

However, the effective deletion and insertion rates can be significantly reduced. Indeed, as a router can exchange network state information with multiple routers, the same update information can be received by a single router multiple times. Moreover, a prefix can be withdrawn and inserted many times within minutes following a network link failure. Indeed, previous works have shown that the mean time observed to recover after a network link failure is in the order of minutes [115]. As the information is received multiple times from neighbouring routers, a prefix can be withdrawn and inserted many times within minutes following a network link failure. In summary, by delaying deletions and pushing a single update per prefix per timestamp, the effective prefix addition and deletion rates drop to less than 20 updates per second.

3.6.2 Update Cost

We now evaluate the cost of updates by counting the number of memory accesses to modify the SHIP data structure. Let C_{HTT} and $C_{hash\,table}$ be the update cost of the HTTs and the hash table, respectively. The insertion or a deletion of a prefix can trigger a hash table update and an update of a HTT, independently of the prefix length. If the 23 MSBs of the updated prefix are not already associated to an ABB held in the hash table, both the hash table and the HTT must be updated. Otherwise, only a HTT is updated. $C_{hash\,table}$ is first evaluated. When a prefix update requires deletion of an ABB in the hash table, a single memory access is required to invalidate the associated entry. However, the entire hash table must be rebuilt to add an ABB.

To reduce the update complexity, we propose to build a hash table holding all the ABBs associated to the current IPv6 unicast address space [112]. Thus, when a prefix is added, the associated ABB entry is enabled with a single memory access. In addition, less than a single update per second was observed to apply to the hash table. Hence, $C_{hash\,table} = insertion_{rate} \times \#memoryAccess_{perinsertion} = 1$. The hash table holding all the ABBs uses 54 kB, which is negligible compared to the SHIP data structure size. In addition, the IPv6 unicast address space was not modified since 2006, and its usage is still extremely low, making

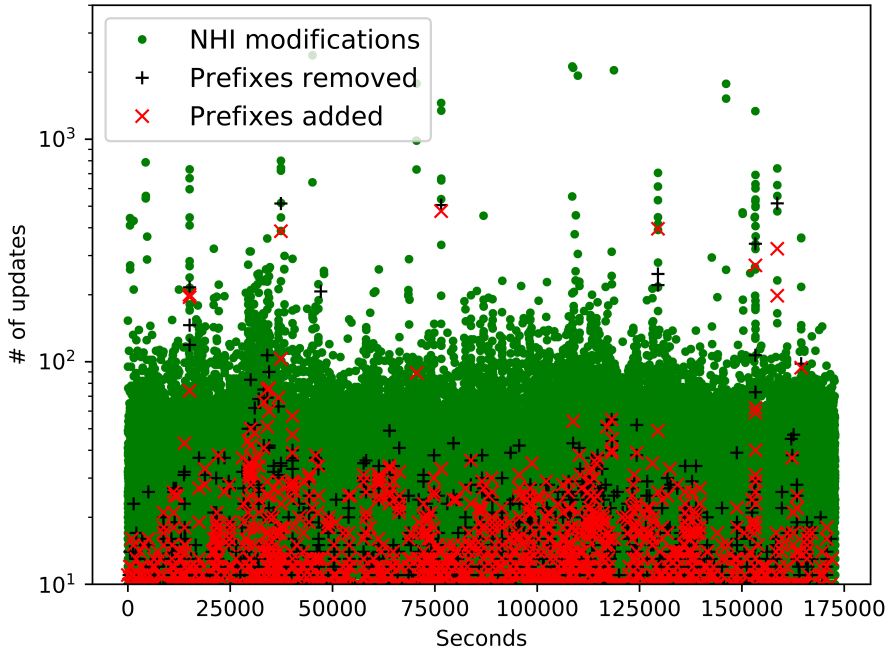


Figure 3.6 Update rates over two days from the rrc00 collector.

the proposed method valid for future IPv6 network growth.

The cost of updates is now evaluated for the HTTs. When a prefix NHI is updated, a single memory access is required to update an HTT leaf. However, the prefix insertions or deletions require to rebuild a portion of an HTT, or a complete HTT. On all the benchmarks used, the number of nodes in an HTT is smaller than the number of prefixes held. As a result, the number of nodes to modify after a prefix update is at most equal to n, the number of prefixes held in an HTT. Because most of the prefix updates observed are applied to the HTTs, $C_{HTT} = insertion_{rate} \times n + deletion_{rate} \times n + NHI_{update\,rate}$.

The cost of updates on the SHIP data structure is $C_{HTT}+C_{hash\,table} = 20\times n+20\times n+2000+1$. Hence, the update complexity of the SHIP architecture is $O(n) \approx O(N)$, where N is the number of prefixes in the FIB. Using our largest prefix table, the largest number of prefixes encoded in an HTT is $n = 7,000$. Thus, in the worst case, up to 282,001 memory accesses are required each second to update the SHIP data structure.

Handling up to 282,001 memory accesses per second in the worst case has a very limited impact on the lookup rate that is on the order of hundred million of lookups per second.

To update the SHIP data structure in the proposed pipelined hardware architecture, *write bubbles* are used. Write bubbles insertion [115] is a technique widely adopted to push updates to a pipelined hardware architecture. A write bubble holds a triplet (pipeline stage, memory

address, value) that is inserted in the pipeline. When a write bubble reaches the pipeline stage specified in its triplet, a control circuitry updates the memory address with the new value held in the triplet.

3.7 Performance Measurement Methodology

SHIP's performance is evaluated using eleven real prefix tables, holding approximately 25 k prefixes, extracted from the RIS remote route collectors [103]. Each scenario, noted *rrc* followed by a two-digit number, characterizes the location in the network of the remote route collector used. In addition, synthetic prefix tables were also used. One synthetic 580 k prefix table was generated using a non-random method [116]. Four smaller prefix tables were created from the 580 k prefix table, with a similar prefix length distribution, holding respectively 290 k, 116 k, 58 k and 29 k prefixes.

Because SHIP uses two-level prefix grouping to cluster prefixes, the prefix distribution after clustering is evaluated in Section 3.8.1. The prefix distribution is evaluated using a small real prefix table, *rrc*00, and the largest prefix table with 580 k prefixes.

SHIP performance is characterized by the number of memory accesses to traverse the data structure and its memory footprint. Two cases were considered ; one without clustering, i.e where a single HTT encodes all prefixes, and one where prefixes are clustered using two level prefix grouping and encoded in multiple HTTs. For the second case, the number K ranges from 1 to 6. The results are reported in sections 3.8.2 and 3.8.3.

The memory bus of the presented architectures allows to read one HTT node per clock cycle. In addition, the selected K HTTs within an ABB are traversed in parallel. The reported number of memory accesses is the largest number of memory accesses between all the HTTs amongst all ABBs.

The HTT memory consumption is given in bytes per byte of prefixes to capture the data structure overhead. This metric is evaluated as the size of the data structure divided by the sum of the size of each prefix held in the prefix table. To characterize the HTT efficiency per level, the HTT node distribution, the HTTs depth distribution, and the prefix distribution are evaluated. Only the maximal depth is considered for the last two metrics. Combined together, these metrics allow to evaluate the average number of prefixes encoded per HTT node, which directly reflects the HTT efficiency. These metrics are evaluated using $K = 2$ PLS groups, similarly to the parameters used for the clustering analysis and for the FPGA implementation, presented in section 3.9.

3.8 Results

3.8.1 Prefix Distribution Within Clusters

As presented in section 3.4, the motivation to bin prefixes upon their MSBs lies in the IPv6 addresses structure [112]. In this section, we demonstrate experimentally that prefixes can be binned using an ABB width set to 23, i.e. the 23 MSBs.

In Fig. 3.7, the impact of the ABB width is evaluated on the prefix distribution, number of bins and number of prefixes. In this figure, the two clustering methods ABB and PLS are applied to prefixes. The number of ABBs bins and the number of prefixes are normalized for an ABB width set to 23 bits because the proposed clustering method bins prefixes on their 23 MSBs.

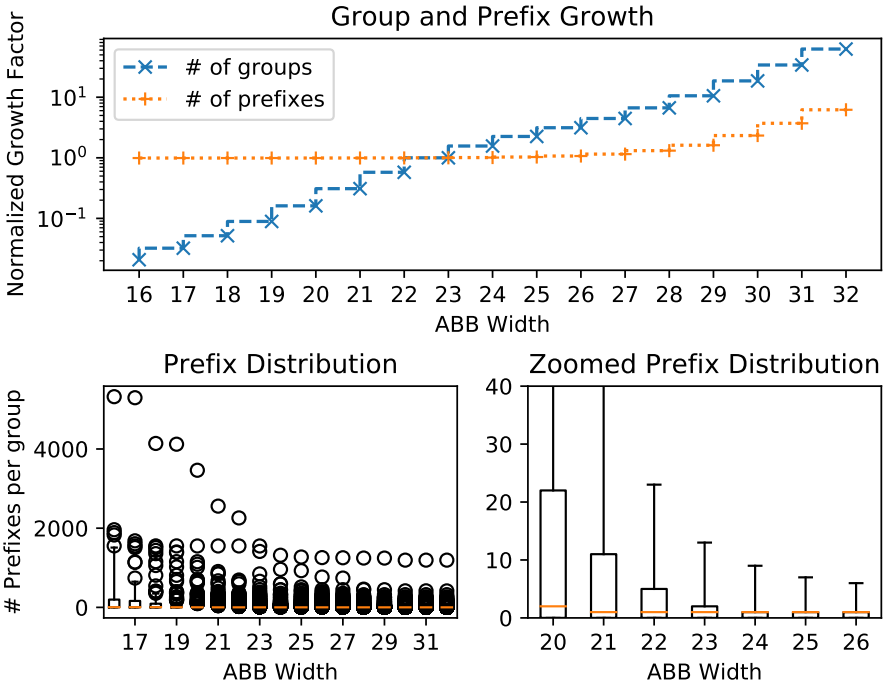
Using real prefixes with an ABB width equal to or greater than 23 bits distributes more evenly the prefixes along PLS groups shown in Fig. 3.7a. Using ABB widths greater than 23 does not help to reduce the maximum number of prefixes held within a PLS group, as illustrated in the box-plot with outliers. In addition, using ABB widths greater than 23 bits has little to no impact on the total number of prefixes (up to 26 bits) but the number of bins has a superlinear growth.

Using synthetic prefixes, a more even distribution is observed for ABB widths greater than or equal to 24 bits. Although using an ABB width greater than 23 bits has very little impact on the total number of prefixes, the number of bins has a superlinear growth with the ABB width.

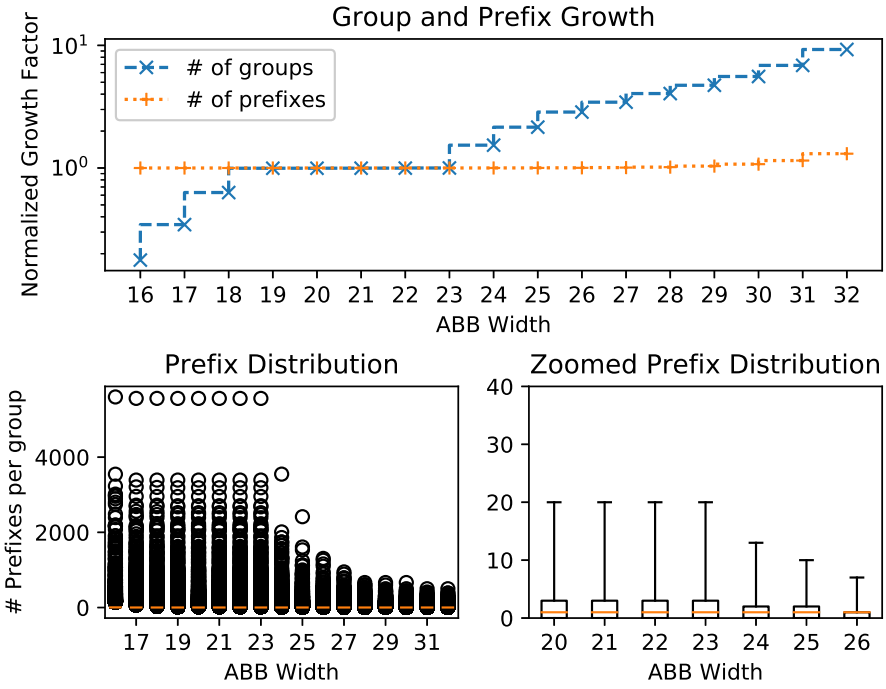
In conclusion, our experiments with real and synthetic prefixes have shown that using 23 bits is a good tradeof.

3.8.2 ABB Hash Table

The performance of the hash table recording the ABB pointers is reported in Table 3.2. For real prefixes, the ABB method uses between 19 kB and 24 kB. The memory consumption is similar across all the scenarios tested because prefixes share most of the 23 MSBs. Using synthetic prefixes, on average, 2.7 bytes of memory per prefix byte are used for the 5 scenarios evaluated. The hash table shows a linear memory consumption scaling. The number of memory accesses is constant to 2 by construction for all scenarios, because a perfect hash function is used.



(a) Real Prefixes.



(b) Synthetic Prefixes.

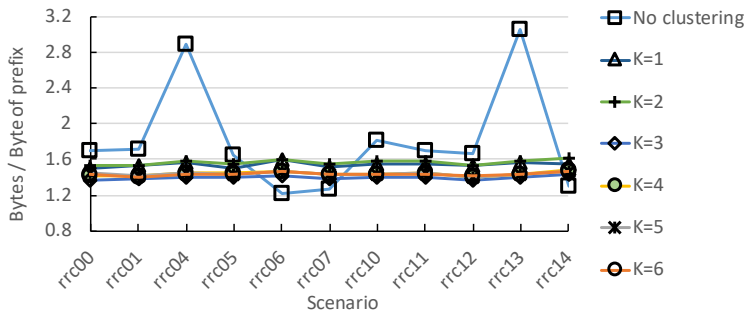
Figure 3.7 Prefix distribution within PLS groups after clustering as a function of the ABB width on (a) real prefixes and (b) synthetic prefixes. The whiskers represent the 5th percentile and 95th percentile.

Tableau 3.2 ABB memory consumption and number of memory accesses.

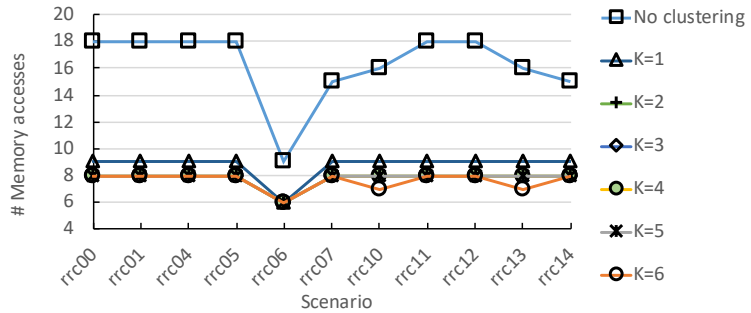
Scenario	Real Prefixes (rrc)											Synthetic Prefixes				
	00	01	04	05	06	07	10	11	12	13	14	29 k	58 k	110 k	290 k	580 k
Hash table size (kB)	20	19	24	19	19	20	21	20	20	22	21	82	162	322	642	1282
Memory accesses	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

3.8.3 HTTs

Real Prefixes. The performance of HTTs is presented in Fig. 3.8a and 3.8b. Two-level prefix grouping reduces the memory consumption and smooths its variability. The memory consumption of the HTTs ranges from 1.36 to 1.60 bytes per prefix byte for all scenarios with two-level prefix grouping. By constrast, without clustering, the memory consumption ranges between 1.22 up to 3.15 bytes per byte of prefix for a single HTT encoding the whole prefix set.



(a) Memory Consumption.



(b) Memory Accesses.

Figure 3.8 Real prefix tables : impact of K (number of PLS groups) on the memory consumption (a) and the number of memory accesses (b) of the HTTs.

Fig. 3.8a shows that increasing K up to 3 reduces the memory consumption, although using more groups worsens the memory consumption. Indeed when K is increased, most groups

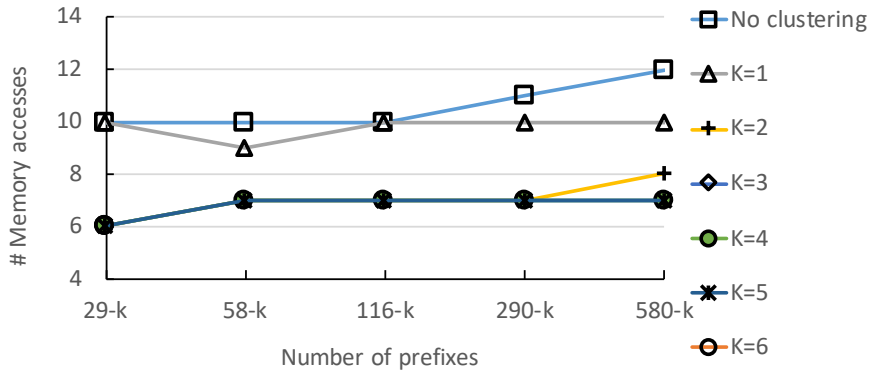
hold very few prefixes, which leads to a large of portion of the memory allocated to the HTT being unused.

The number of memory accesses reduces on average by 2× using the clustering method. While the number of memory access ranges between 9 and 18 without clustering for a single HTT, using two-level prefix grouping, it ranges between 6 and 9, as observed in Fig. 3.8b. However, increasing K from 1 to 6 yields little gain on the number of memory accesses. Indeed, we observed that the number of memory accesses is limited by a few PLS groups holding only \48 prefixes. Hence, increasing K cannot reduce the number of prefixes held in these PLS groups, which results in no improvement to the number of memory access.

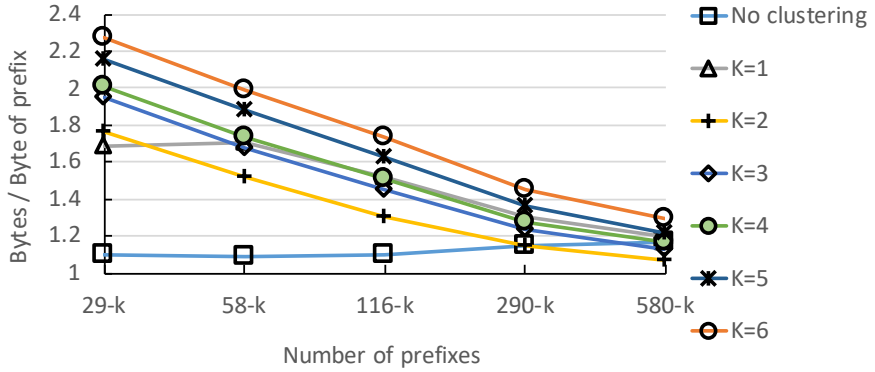
Table 3.3 shows that the average HTT depth is 1.4 and 2.1 for PLS group 1 and 2, respectively. Around 75% of HTTs have a depth equal to one because 75% of the PLS groups hold fewer than 2 prefixes, as presented in Fig. 3.7. Indeed, in an HTT, two prefixes can be stored in a single node. Experimentally, an average of 1.3 prefix and 1.4 prefix are encoded per HTT node at the level one for PLS group 1 and 2, respectively. For levels greater than one, the average number of prefixes encoded per HTT node is 1.5 and 1.0 for PLS groups 1 and 2, respectively. Based on Fig. 3.7, HTT with a depth greater than one encode more than 75% of the prefixes.

Synthetic Prefixes. The performance of the HTTs with synthetic prefixes is presented in Fig. 3.9a, 3.9b, and 3.9c.

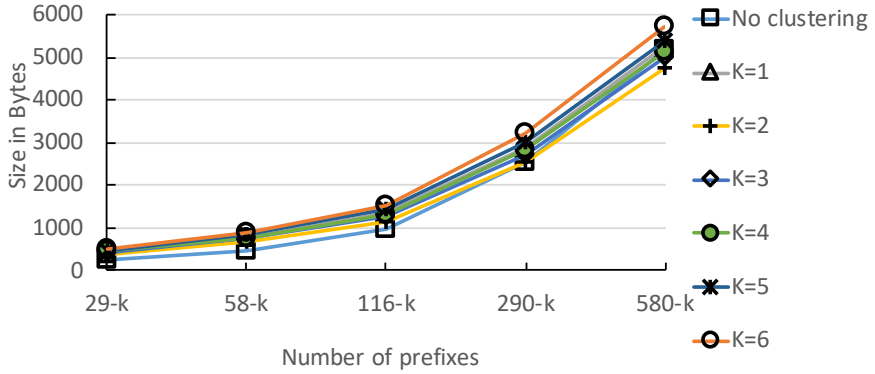
Two behaviors can be observed for the memory consumption in Fig. 3.9b. For prefix tables holding 290 k prefixes or more, using two-level prefix grouping with $K = 2$ groups slightly decreases the memory consumption over the case of a single HTT (i.e. without clustering). In addition, using $K > 2$ does not improve memory efficiency. For smaller prefix tables with up to 116 k prefixes, a lower memory consumption is achieved using only a single HTT (i.e without clustering). Indeed, using synthetic prefix tables holding up to 116 k prefixes, most of the PLS groups hold a single prefix. Thus, for each PLS group, a large of portion of the memory allocated to the HTT is unused, which reduces the memory efficiency.



(a) Memory accesses.



(b) Memory consumption.



(c) Memory consumption scaling.

Figure 3.9 Synthetic prefix tables : impact of K (number of PLS groups) on the number of memory accesses (a), the memory consumption (b) and the HTTs memory consumption scaling (c).

Tableau 3.3 HTTs analysis per level : distribution of HTT nodes, prefix distribution and HTT depth distribution. The number of prefixes and number of HTTs at level i is the sum of the number of prefixes encoded in HTTs with a depth of i, and the total number of HTTs with a depth of i. The reported results are for a synthetic (580K) and a real prefix table (rrc00) using two PLS groups. Depth and level are used here as synonyms.

Depth	1		2		3		4		5		6		7		8	
PLS Group	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
rrc00																
# Prefixes	2390	65	937	181	1223	14	3573	29	7026	237	9669	445	189	0	174	0
# HTTs	1881	46	171	28	109	1	73	2	67	2	49	8	4	4	2	0
# Nodes	2356	93	8720	222	2628	187	2887	92	1238	253	133	63	12	0	3	0
580 k																
# Prefixes	54821	545	11494	675	23935	145	61138	74	150351	1122	176147	1203	95890	968	2229	0
# HTTs	36691	423	1560	174	1217	10	1137	5	2161	13	655	12	306	5	1	0
# Nodes	43728	642	80921	727	68149	561	56604	435	34817	309	16619	175	522	8	2	0

The memory consumption scaling of the HTTs is presented in Fig. 3.9c. This memory consumption, with and without two-level prefix grouping, grows linearly with the number of prefixes. Note that the abscissa uses a logarithmic scale. Thus, the memory consumption scaling of the HTT is linear with and without two-level prefix grouping.

Based on Fig. 3.9a, using two-level prefix grouping reduces the number of memory accesses over a single HTT. On average, the number of memory accesses is reduced by 40% over a single HTT using 2 groups or more. However, using $K > 3$ does not further reduce the number of memory accesses. Indeed, the PLS groups causing the largest number of memory accesses cannot be reduced in size by increasing the number of groups. Finally, Fig. 3.9a shows that the increase in the number of memory accesses for a search is at most logarithmic with the number of prefixes, since each curve is approximately linear and the x-axis is logarithmic.

In terms of HTT efficiency, the same conclusions can be drawn with synthetic prefixes. Based on Table. 3.3 the average HTT depth is 1.5 for both PLS groups 1 and 2. 80% of HTTs have a depth equal to one, which is coherent with the prefix distribution presented in Fig. 3.2. Experimentally, on average, 1.5 and 1.3 prefixes are respectively encoded per HTT node at the level one for PLS groups 1 and 2. For levels greater than one, the average number of prefixes encode per HTT node is 2.0 and 2.1 for PLS groups 1 and 2, respectively.

Lastly, the HTTs used with two-level prefix grouping have a linear memory consumption scaling, and a logarithmic scaling for the number of memory accesses. The hash table used in the ABB method has shown to offer a linear memory consumption scaling and a fixed number of memory accesses. Thus, SHIP has a linear memory consumption scaling and a logarithmic scaling for the number of memory accesses using all the considered benchmarks.

3.9 FPGA Implementation

Two FPGA hardware implementations of the SHIP architecture were developed and characterized. One was optimized to reduce latency while the other was optimized to offer high throughput

3.9.1 Architectures Overview

The two proposed architectures implement a pipelined traversal of the SHIP data structure. The HTT traversal module of both architectures is divided in $K = 2$ parallel pipelines. Each pipeline $i \in [1, ..., K]$ performs the traversal of the HTT selected by the root pointer address held in the i-th PLS group.

For the low-latency architecture, the HTT traversal module pipeline is divided into d stages, such that each pipeline stage is dedicated to the traversal of a single HTT level. A pipeline stage $j \in [1, ..., d]$ is composed of a Traversal Engine (TE) and a memory, holding the HTT nodes of level j of all the ABBs (Bin 1 to Bin M). The traversal engines of the first $d-1$ levels are used to process only DAT nodes, whereas LB nodes processing is only done at the level d. The low-latency lookup architecture was presented in a previous paper [117]. To increase the lookup rate over the low-latency architecture, the high-throughput architecture divides each pipeline stage of the low-latency architecture over multiple stages. By reducing the number of logic levels in each stage, shorter clock periods can be obtained, which also increased the lookup rate. The two architectures were described in C++, synthesized with Vivado High Level Synthesis (HLS) 2018.1, and implemented using Vivado 2018.1 with the Virtex7 VX690TFFG1761 and the UltraScale+ XCVU9P-FLGA2577. The method used to describe accurately the SHIP architecture using C++ is presented in [117].

3.9.2 Results

For each design presented in Table 3.4, four performance metrics are of interest ; the lookup rate, the latency, the on-chip memory usage (BRAM) and the logic usage. The lookup latency, or wall-clock time, is evaluated as the clock period by the number of pipeline stages.

The results of the low-latency architecture are discussed in a previous paper [117]. In this section, the performance of the high-throughput architecture is first discussed. Then, the overhead of the high-throughput architecture is compared to the low-latency architecture.

High-Throughput Architecture

The lookup rate of the high-throughput architecture implemented on a Virtex7 degrades with the prefix table size. Using 25 k prefixes, up to 344 Mpps is supported, while the lookup rate is reduced to 263 Mpps when 290 k prefixes are used, and is further reduced to 166 Mpps with the largest prefix table.

The lookup rate degradation with the prefix table size relates to the distribution of the on-chip memories in the FPGA. On-chip memories, block RAMs, are divided in columns. When a level of the SHIP data structure is mapped to more than one block RAM column, the clock period increases due to the routing delay to access multiple columns. As a direct consequence of an increased routing delay, the lookup rate is reduced.

The latency also increases with the prefix table size as a consequence of the clock period degradation for larger scenarios. For prefix table holding up to 290 k prefixes, the lookup

Tableau 3.4 SHIP architecture performance and cost for different table sizes.

Prefix table	Latency (ns)	Period (ns)	Lookup rate (Mpps)	BRAM 36-Kb	Slices (in K)
Low-latency Architecture					
rrc00	105.6	9.6	104	96.5	7.3
29 k	104.5	9.5	105	114.5	7.3
58 k	105.6	9.6	104	214.5	7.6
116 k	114.4	10.4	96	311.5	7.9
290 k	119.9	10.9	91	734.0	8.6
580 k	125.4	11.4	87	1416.0	9.2
High-throughput Architecture					
rrc00	189.5	2.9	344	96.5	9.8
29 k	225.7	3.4	294	114.5	11.0
58 k	216.0	3.3	333	214.5	10.8
116 k	189.0	2.9	344	311.5	10.6
290 k	246.4	3.8	263	734.0	15.2
580 k	396.3	6	166	1416.0	15.8
580 k [i]	219.2 [i]	3.4[i]	294 [i]	1416.0 [i]	9.5 [i]

[i] Implemented on a state-of-the art UltraScale+

latency ranges between 189.5 and 246.4 ns, while the lookup latency increases to 396.3 for the largest prefix table.

The block RAM usage is almost linear with the prefix table size, which is a consequence of the linear memory consumption observed in Fig 3.9c.

As a consequence of the higher consumption of block RAMs for larger scenario, the FPGA logic usage increases with the prefix table size. Between the smallest scenario and the largest scenario, the logic usage increases by up to 61%. Indeed, the circuitry needed to combine individual block RAMs into a large memory consumes FPGA logic.

As the high-throughput architecture performance on the Virtex7 is limited by routing delays, we evaluated this architecture on a state-of-the-art FPGA, an UltraScale+, which features deeper block RAMs columns and reduced routing delays. On this FPGA, the clock period is reduced by 1.7× over a Virtex7 for the largest scenario. Consequently, the lookup rate is increased to 294 Mpps, and the lookup latency is reduced to 219.2 ns. In addition, the logic consumption is reduced by 1.6× over the Virtex 7 due to modified internal organization of the UltraScale+ that allows to build large memories in the block RAM column without additional use of logic resources.

To further increase the lookup rate, the high-throughput architecture can be modified to

double the lookup rate. The proposed technique consists in using two parallel lookup engines sharing dual-port block RAMs available on both the Virtex7 and the UltraScale+. A single dual-port memory block can serve simultaneously the two lookup engines. As a result, the high-throughput architecture with dual-port memoriy blocks can support 332 Mpps and 588 Mpps on the Virtex 7 and on the UltraScale+, respectively. However, because two parallel lookup engines are implemented, the logic consumption is almost doubled.

Overhead of the High-Throughput Architecture

The high-throughput architecture almost triple the lookup rate over the low-latency architecture when using a Virtex7, and by more than 3.3× using an UltraScale+, as shown in Table 3.4. However, the high-throughput architecture uses up to 1.8× more FPGA logic than the low latency architecture because of its deeper pipeline, which requires FPGA logic to synchronize intermediate results in each pipeline stage. Still, the increased lookup rate of the high-throughput architecture outweighs the logic consumption overhead compared to the low-latency architecture. In addition, the on-chip memory usage is not impacted by the pipeline depth, and thus, the consumption of block RAMs remains similar for the two architectures.

The down side of the high-throughput architecture lies in the increased lookup latency compared to the low-latency architecture. As the number of pipeline stages is increased by 5.4×, while the clock period is reduced by less than 3×, the lookup latency increases by 3× up to 396 ns for the largest prefix table compared to the low-latency architecture on a Virtex7. Using an UltraScale+, the lookup latency of the high-throughput architecture is only increased by 1.7×, to 219 ns, over the low-latency architecture.

3.10 Comparison with Previously Reported Results

3.10.1 Performance Comparison

To normalize the performance between the various works summarized in Table 3.5, only the performance of a single lookup engine is reported without using dual-port memories.

Table 3.5 compares the performance of SHIP and previous work in terms of lookup latency, lookup rate, memory consumption, logic consumption, and update complexity. The memory consumption is expressed in bytes per prefix, as the size of the data structure divided by the number of prefixes used. The hypothesis used to evaluate the lookup latency of solutions using external memory such as SRAM or DRAM were presented in a previous paper [117].

Tableau 3.5 Performance comparison.

Method	# of prefixes	Latency (ns)	Lookup Rate (Mpps)	BRAM 36-Kb	External Memory (Mbits)	Memory Efficiency (B/prefix)	Memory Efficiency Relative to this work	Slices	Update complexity	Platform
TCAM-based solutions										
[18][i]	32 768	N/A	84	778	0	109	+808%	N/A	O(1)	FPGA Intel StratixV
Tree-based solutions										
[97]	65 535	126.4	217	320.5	0	22.5	+100%	1327	O(N)	FPGA Xilinx Virtex7
[71]	332 118	290	188	580	32.5	21	+87%	15 358	O(log(N))	FPGA Xilinx Virtex6
[72][ii]	7049	91.2	206	N/A	0	29.4	+145%	2919	O(log(N))	FPGA Xilinx Virtex6
Trie-based solutions										
[84]	410 513	142	7	40	0	0.4	N/A	N/A	N/A	N/A
[98]	318 043	295	88	108	307.2	124.2	+1008%	4583	O(N)	FPGA Xilinx Virtex4
[81]	20 440	N/A	211	0	1.4	72.0	+500%	N/A	N/A	CPU Intel i7 4770K
Misc solutions										
[85][iii]	410 513	142	7	40 [iv]	0	0.4	-97%	N/A	N/A	FPGA Xilinx VirtexII
[87][v]	14 000	N/A	239 [ii]	500.5	0	164	+1266%	N/A	O(1)	FPGA Xilinx Virtex7
Trie-Tree based solutions										
[101]	322 000	544.5	180	N/A	63.1	27.6	+146%	N/A	N/A	FPGA Xilinx Virtex6
SHIP	580 737	396	166	1416	0	11.2	0	15 878	O(N)	FPGA Xilinx Virtex7
SHIP	580 737	219	294	1416	0	11.2	0	9515	O(N)	FPGA Xilinx UltraScale+

[i] Equivalent number of M20K block RAMs into 36K block RAMs.
[ii] Results adjusted for a single pipeline.
[iii] Work done using IPv4 prefixes.
[iv] Estimated BRAM consumption.
[v] Performance for a partial IPv6 lookup engine that only identifies the prefix length of the matching prefix.

A FPGA-based TCAM was proposed recently [18], which supports 84 Mpps and uses around 777.5 equivalent Xilinx block-RAMs. Thus, this solution has a 3.5× lower lookup rate than SHIP, while consuming 9× more memory space. However, the update complexity of this solution is lower compared to SHIP.

The Xilinx LPM IP used in SDNet [97] has a low FPGA resource consumption. Compared with our high-throughput architecture with 116 k prefixes, the Xilinx solution lookup rate is 1.6× slower, but has a 1.5× lower lookup latency. In addition, the Xilinx LPM solution requires a shallow data structure to support update, doubling the reported memory consumption compared to the high-throughput architecture. In addition, its update complexity is higher compared to SHIP, where only one HTT needs to be rebuilt after a prefix insertion or deletion.

The 2-3 trees architecture [71] has a lookup latency 1.3× higher than the high-throughput implemented on an UltraScale+. In addition the bandwidth supported is 1.54 × lower than our proposed architecture and is 87% less memory efficient than the high-throughput architecture. However, the 2-3 tree architecture has an update complexity of $O(log(N))$, whereas SHIP requires to rebuild an HTT that contains a subset of the prefix table.

Our high-throughput architecture exhibits a memory consumption that is 2.6× lower than the LayeredTree architecture [72], while supporting a 1.45× higher lookup rate. However, the high-throughput architecture has a higher lookup latency and uses 3.2× more FPGA logic. The lower usage of FPGA logic by the LayeredTree architecture is a consequence of the very small prefix table used, which requires very few pipeline stages and block RAMs. In addition, the LayeredTree architecture has an update complexity of $O(log(N))$, whereas SHIP has an update complexity of $O(N)$.

The memory efficiency of the LC-trie is 4.4× lower than SHIP for the same prefix set. Moreover, for the same prefix set, the LC-trie algorithm requires in the best case 18 memory accesses, whereas SHIP requires 10 memory access. The update complexity of a LC-trie is not reported in [84].

The results presented for FlashTrie [105] were reevaluated using the node size equation presented in [105] due to incoherence with equations shown in [98]. In addition, the bandwidth supported by the FlashTrie architecture drops to 88 Mpps when the DRAM timing characteristics are fully considered. Compared to the high-throughput architecture implemented on the UltraScale+, the FlashTrie architecture has a lookup rate 3.3× lower, uses roughly 3.4× less FPGA resources, but consumes 11× more memory and requires 1.34× more time to complete a lookup. The update complexity of FlashTrie is $O(N)$, which is similar to SHIP.

PopTrie [81], uses 72 bytes per prefixes while supporting 211 million lookups per second. Thus, PopTrie uses almost 7× more memory per prefix, while supporting 1.4× fewer lookups compared to SHIP implemented on an UltraScale+. The latency for IPv6 prefix is not reported.

A new method presented by Rétvári *et al.* [85] yields highly memory-efficient data structures IPv4 prefixes. However, compared to the high-throughput architecture, the memory consumption is roughly 25× lower, but the lookup rate is 42× lower. Thus, the benefit of the compressed data structure does not outweigh the reduced lookup rate compared to SHIP.

The performance reported for the SAIL architecture [87] is not evaluated with a module that identifies the prefix matching for a given prefix length. However, based on an evaluation of the PopTrie's authors [81], a complete lookup solution based on the SAIL framework uses 87 bytes per IPv4 prefix, for a table holding 520 k IPv4 prefixes. Thus, even for IPv4 prefixes, SAIL consumes around 8× more memory compared to SHIP on IPv6 prefixes.

Yang *et al.* proposed the CLIPS architecture [118] extended to IPv6 [101]. Their method uses 27.6 bytes per prefix, which is about 2.5× larger than SHIP. The estimated latency of CLIPS 544.5 ns, which is 2.4× higher compared to the proposed architecture on the UltraScale+. In addition, CLIPS uses more than 2.4× the number of bytes per prefixes than the proposed architecture, with a lookup rate 1.6× lower than the high-throughput architecture implemented on an UltraScale+.

3.10.2 Impact of the FPGA Generation on the Performance

In this section, we discuss the impact of the FPGA generation on the overall performance, which is ignored in the previous works.

A precise comparison would require a full implementation of each work across all FPGA generations, but a simple assumption can be made. In the best case, the performance of the architectures presented in Table 3.5 is limited only by the speed of the on-chip memory, and not by the number of logic levels. Thus, as a first-order approximation, we assume that the architectures presented in Table 3.5 could run at a clock period similar to the SHIP high-throughput architecture on an UltraScale+.

Based on this assumption, the performance of the different works are compared against SHIP. The Xilinx LPM solution [97] is not discussed as this solution is limited to 64 k prefixes. Similarly, as the SAIL architecture is not a complete LPM solution, SAIL is discarded from the following discussion.

A lower clock period resulting from the use of a state-of-the art FPGA would benefit to

pipelined architectures that are not limited by the performance of some external memory. That is, the lookup rate of most of the architectures presented in Table 3.5, would be equal to the one supported on the high-throughput architecture. However, the lookup rate of the FlashTrie architecture would not be improved, as the performance bottleneck is the external DRAM. Similarly, the compressed data structure architecture is not pipelined, and thus will lightly benefit from a higher clock rate. Even with a lower clock period, the CLIPS architecture would have a higher latency compared to SHIP, while the memory consumption would remain higher than SHIP. Similarly, the 2-3 Tree architecture would have a slightly higher latency compared to SHIP, while the memory consumption would remain 80% higher than SHIP.

The LayeredTree architecture would benefit from a lower clock period, as the latency would be 3× smaller than the high-throughput architecture. However, the memory consumption of LayeredTree would remain 2.6× higher than SHIP.

In conclusion, even if the previously published architectures and the high-throughput architecture could support the same lookup rate, the high-throughput architecture still has a higher memory efficiency and a lower lookup latency. Only one solution would have a lower lookup latency compared the high-throughput architecture.

3.11 Conclusion

This paper proposed SHIP, a scalable and high performance IPv6 lookup algorithm to address the performance requirements of current and future network applications. SHIP exploits the prefix characteristics to create a shallow and memory-efficient data structure. First, the allocated IPv6 address space is used to bin the prefixes efficiently on their MSBs. Within each bin, prefixes are sorted in groups based on the FIB prefix length distribution. Each prefix group is then encoded in a HTT. The proposed HTT revisits the concept of multi-bit trie, but adapts the number of nodes built on the prefix density to improve the memory efficiency. The trie leaves are transformed into leaf buckets leaves to further improve the memory efficiency.

The proposed data structure supports incremental updates and is mapped efficiently to hardware. A pipelined high-throughput hardware architecture is proposed, which leverages on-chip memories. Evaluated with real and synthetic prefix tables SHIP exhibits a logarithmic scaling factor in terms of the number of memory accesses and a linear memory consumption scaling. Compared to other well-known approaches, SHIP reduces the amount of memory used per prefix by 87%. Implemented on an UltraScale+ FPGA the proposed high-throughput architecture can support a 588 millon packets per second throughput.

CHAPITRE 4 ARTICLE 2 - A LOW-LATENCY MEMORY-EFFICIENT IPv6 LOOKUP ENGINE IMPLEMENTED ON FPGA USING HIGH-LEVEL SYNTHESIS

Auteurs : Thibaut Stimpfling, JM Pierre Langlois, Normand Bélanger, et Yvon Savaria.

Publié dans : 18ème IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). 2018.

Abstract The emergence of 5G networks and real-time applications across networks has a strong impact on the performance requirements of IP lookup engines. These engines must support not only high-bandwidth but also low-latency lookup operations. This paper presents the hardware architecture of a low-latency IPv6 lookup engine capable of supporting the bandwidth of current Ethernet links. The engine implements the SHIP lookup algorithm, which exploits prefix characteristics to build a compact and scalable data structure. The proposed hardware architecture leverages the characteristics of the data structure to support low-latency lookup operations, while making efficient use of memory. The architecture is described in C++, synthesized with a high-level synthesis tool, then implemented on a Virtex-7 FPGA. Compared to the proposed IPv6 lookup architecture, other well-known approaches use at least 87% more memory per prefix, while increasing the lookup latency by a factor of 2.3×.

4.1 Introduction

Global IP traffic carried by networks is continuously growing, and it is envisioned to increase threefold between 2016 and 2021 [99]. To handle this increasing Internet traffic, network-link working groups have ratified the 100-gigabit Ethernet standard and are working on the next generation Ethernet standard. As a result, IP lookup engines used in the routing process have to process packets at those line rates and to support the ongoing growth of IPv6 networks.

However, IP lookup engines must be optimized not only to support the current line rate but also to have low latency lookup operations. Indeed, next-generation networks such as 5G, and many future applications such as virtual reality, remote object manipulation, eHealth, and autonomous driving require low-latency processing times [17].

In addition, IP lookup algorithms and architectures tailored for IPv4 are not performing well with IPv6 [71], [101], due to the fourfold increase in the number of bits in IPv6 addresses

over IPv4. As a consequence, dedicated IPv6 lookup methods are required.

An IP lookup consists of identifying the next hop information (NHI). The NHI describes the output port to which a packet must be forwarded. During a lookup, the destination IP address is matched against a list of entries stored in a table, called the forwarding information base (FIB). Each entry in the FIB represents a network defined by its prefix address and associated NHI. While a lookup key may match multiple entries in the FIB, only the longest prefix and its NHI are returned, because IP lookup is based on the Longest Prefix Match (LPM) [100].

In this paper, we introduce the architecture of an IPv6 lookup engine designed to meet current and future performance requirements. The IPv6 lookup engine implements the SHIP (Scalable and High Performance IPv6) lookup algorithm that we previously devised [16]. The architecture is described in C++, synthesized with a High-Level Synthesis (HLS) tool, and then implemented on a Virtex-7 FPGA.

The main contribution of this paper is a low-latency hardware architecture that efficiently implements the SHIP lookup algorithm. This is achieved by :

— Exploiting the SHIP data-structure characteristics ;

— Proposing a pipelined organization tailored for efficient logic resource usage ;

— Leveraging code restructuring techniques to describe the proposed architecture for HLS.

Compared to the proposed IPv6 lookup architecture, other well-known approaches use at least 87% more memory per prefix, while increasing the lookup latency by a factor of 2.3×.

The remainder of this paper is organized as follows. Section 4.2 introduces common approaches used for IP lookup and Section 4.3 gives an overview of the SHIP algorithm. The IPv6 lookup architecture is presented in Section 4.4. Section 4.5 introduces the code restructuring method used to describe the presented architecture efficiently for HLS. The implementation results and a comparison against other works are reported in Section 4.6. Lastly, we conclude the work by summarizing our main results in Section 4.7.

4.2 Related Work

Algorithmic solutions proposed in the literature for IP lookup use mainly four types of data structures : hash tables, Bloom filters, tries, and trees. Each of the four data-structure types offers a different tradeoff between time and storage complexity. However, the performance of an IP lookup solution is not only impacted by the time complexity of the data structure used but also by its storage complexity. Indeed, memory performance is highly related to the memory technology selected, which is guided by the storage complexity of a data structure.

Hash tables have a $O(1)$ lookup time and a $O(N)$ space complexity, with N being the number of entries. However, a pure hash based LPM solution can require up to one hash table per IP prefix length. An alternative to reduce the number of hash tables is to use prefix expansion [69], but this drastically increases memory usage. Moreover, a hash function can lead to collisions, which impose extra-matching sequences and drastically decrease performance [105], [106]. A method has been proposed to reduce the number of collisions using a pool of hash tables [69], [105]. The proposed method divides the prefix table into groups of prefixes, and selects a hash function such that it reduces the number of collisions within each prefix group. However, the low memory efficiency of the solution presented in [69] requires the use of DRAM memories, which have a high latency. As a result, it is unclear whether the proposed hash-based data structures can address forthcoming challenges.

Low-memory footprint hashing schemes known as Bloom filters have also been proposed in the literature [106], [107]. Bloom filters are used to select a subgroup of prefixes that may match the input IP address. However, this data structure always generate false positives at some rate that depends on the configuration parameters used. Thus, a Bloom filter can improve the average lookup time, but it can also lead to poor performance in the worst case, as many sub-groups may need to be matched.

Tree solutions based on binary search trees (BSTs), or generalized B-trees have also been explored [71], [101]. Such data structures are tailored to store loosely structured data such as prefixes, as their time complexity is independent from the prefix distribution characteristics. Indeed, BST and 2-3 trees have a time complexity of $O(log(N))$ and an $O(N)$ storage complexity [71]. Nevertheless, the memory consumption of the data structure proposed requires the use of on-chip memory and multiple external SRAMs. Although the bandwidth supported per lookup engine is high, the proposed architecture leads to a high lookup latency.

The trie data structure, also known as radix tree, has regained interest with the tree bitmap [80]. A tree bitmap improves significantly the very poor memory efficiency of a multi-bit trie by using a bitmap to encode each level of a multi-bit trie. An improved bitmap tree, the PC-trie data structure, is proposed for the FlashTrie architecture [98]. Still, a PC-trie requires to be used jointly with a pre-processing hashing stage to reduce the total number of memory accesses and improve the memory efficiency. The FlashTrie architecture is designed to support a large bandwidth exploiting both on-chip FPGA memory and external DRAM memory. However, the architecture has high lookup latency due to multiple DRAM memory accesses.

At the other end of the spectrum of algorithmic solutions, TCAMs have been proposed as a pure hardware solution, achieving $O(1)$ lookup time by matching the input key simultaneously

against all prefixes. Although TCAMs offer $O(1)$ lookup time complexity, these specialized memories do not hold the NHI associated to each prefix. Thus, a memory access in an external memory is required to retrieve the information associated to the matched prefix, which increases in practice the lookup latency. Moreover, these solutions use a very large amount of hardware resources, leading to large power consumption and high cost, which makes them unattractive for routers holding a large number of prefixes [98], [104].

More recently, information-theoretic and compressed data structures have been applied to IP lookup, yielding very compact data structures handling a very large number of prefixes [85]. Even though this work is limited to IPv4 addresses, it is an important shift in terms of concepts. However, interesting as it may be, a hardware implementation of the architecture achieves 7 million lookups per second. In order to support a 100-Gbps bandwidth, an implementation based on these concepts would require many lookup engines, leading to a memory consumption that is similar or higher than previous trie or tree algorithms [69], [71], [98], [105].

Algorithmic LPM solutions have been implemented on a wide range of platforms including CPUs and GPUs, although only FPGA and ASIC implementation were introduced in this section. However, from a system point-of-view, solutions implemented on GPUs and CPUs [74], [87], [94], [95] suffer from high-latency and a relatively limited throughput. Indeed, IP lookup software implementations are limited by the performance of packet I/O frameworks such as netmap [90] or DPDK [91], which cannot forward at high rates small packets to the main memory of the system [82], [90], [92]. Lastly, IP lookup software based solutions have a high-latency, notably because of the packet transfer between the network interface card (NIC) and the DRAM over PCI express, but also because batch processing techniques are used to increase the bandwidth [82], [92], [94].

4.3 The SHIP Algorithm

The proposed architecture is based on the SHIP algorithm that we previously proposed [16]. This section introduces SHIP before the proposed architecture is described.

4.3.1 Overview

SHIP consists of two procedures : the first one is used to build the SHIP data structure, while the second one, the lookup algorithm, is used to traverse the data structure.

To build a compact data structure, SHIP combines prefix clustering methods with memory efficient data structures. Prefixes are clustered according to their characteristics using two-

level prefix grouping ; prefixes are divided into bins on their most significant bits (MSBs), and are further sorted into groups based on their prefix length. The SHIP data structure stores the bins obtained after division, and encodes the prefixes held within each prefix group.

The SHIP data structure is presented in Fig 4.1. An N-entry hash-table holds M valid address block bins (ABBs) obtained after dividing the prefixes on their MSBs. Each valid ABB points to a set of K prefix length sorted (PLS) groups, encoded in hybrid trie-trees (HTTs).

The lookup algorithm, identifying the NHI associated to the longest prefix matched, is presented in Fig 4.1. First, the MSBs of the destination IP address are hashed to select an ABB pointer stored in the hash table. In this example, the m-th ABB is selected in the hash table. The selected ABB points to a set of K HTTs represented with a dashed rectangle. Second, using the least significant bits of the destination IP address, the selected HTTs are traversed in parallel. Because each HTT can hold a prefix matching the IP address, a priority resolution module is used to select the NHI associated to the longest prefix.

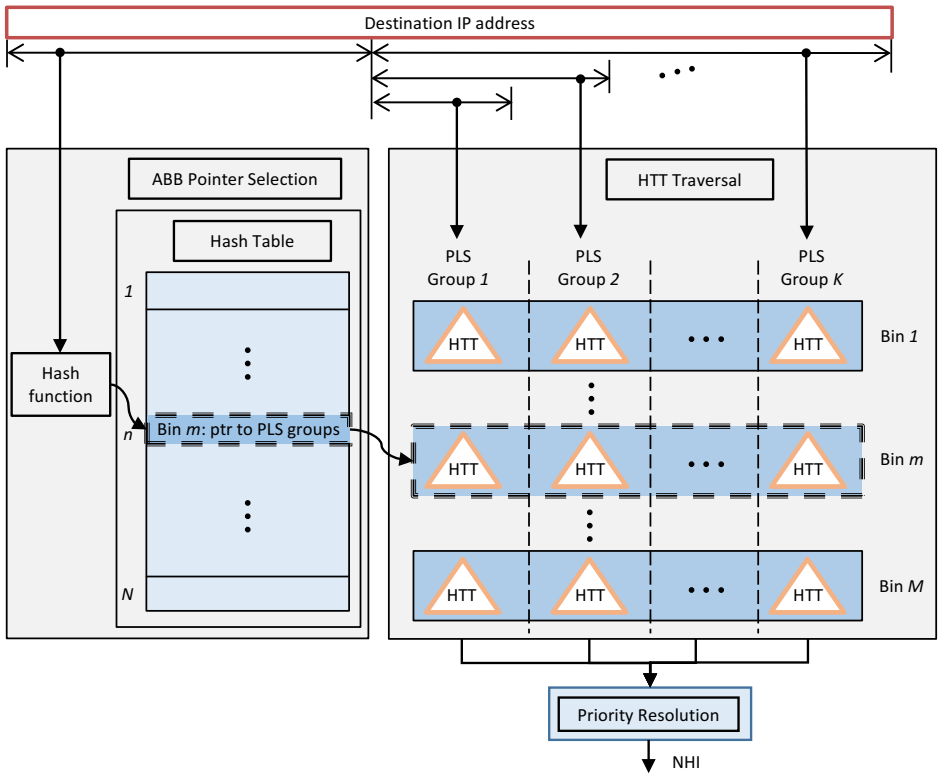


Figure 4.1 SHIP two-level data structure organization and its lookup process with M address block bins and K prefix length sorting groups.

4.3.2 Two-Level Prefix Grouping

This section introduces two-level prefix grouping, which divides prefixes in address block bins, and further sorts prefixes into prefix length groups.

The proposed address block binning method exploits the small IPv6 address space allocated by the Internet Assigned Numbers Authority (IANA) [112] to match the 23 most significant bits efficiently of an IPv6 address against prefixes. For this purpose, prefixes are divided based on their first 23 bits into M non-empty bins. A perfect hash table of N entries stores the address block bin values [113]. Prefixes associated to an ABB are further sorted by their length into K groups of disjoint prefix length ranges, using the PLS method. This method aims at reducing the number of overlapped prefixes, i.e. prefixes with a short prefix length enclosing prefixes with a longer prefix length, within an ABB. The prefix length range covered by each group is selected based on two principles. First, when a prefix length accounts for a large percentage of the total number of prefixes, the prefix length is used as an upper bound of the considered group. Second, prefix length ranges must be chosen such that the K groups are as balanced as possible in terms of number of prefixes. Prefixes held in each non-empty K PLS group are then encoded in an HTT.

4.3.3 Hybrid Trie-Tree data structure

The HTT data structure is designed to adapt its *shape* to the characteristics of the prefixes used. The HTT data structure revisits the well known multi-bit trie and tree data structures. A HTT combines a density-adaptive trie (DAT), a memory efficient multi-bit trie, and a leaf bucket, an extension of a D-tree leaf [114].

All prefixes are initially inserted in a binary trie. The proposed HTT data structure encodes efficiently in an iterative fashion sub-tries of the binary-trie. The depth of each binary trie is determined using an heuristic that analyzes the prefix length distribution to select the deepest sub-trie that can be encoded within a given memory budget. Thus, the depth of a sub-trie is selected such as to minimize the number of levels within a HTT. Moreover, the DAT encodes efficiently each selected sub-trie. While a k-bit trie encodes recursively a k-level sub-trie of a binary trie into 2^k child nodes, the proposed DAT adapts the number of child nodes created to the prefix density within each sub-trie. That is, low-density neighbour child nodes are merged into a single node. In a DAT, two child nodes are merged at a time, although multiple children nodes can be merged together. By merging child nodes together, the DAT improves the memory efficiency of the data structure.

While the number of prefixes held in a sub-trie is higher than a fixed threshold b, the sub-trie

is encoded using a DAT. Otherwise, a leaf bucket (LB) is built. The proposed leaf bucket stores in a linear fashion the prefixes and their associated NHI held in the leaves of a sub-trie. However, to improve memory efficiency, the leaf bucket only holds the prefix bits left unmatched. The sub-trie held in a DAT and leaf bucket are described in memory using DAT nodes and LB nodes, respectively.

4.3.4 HTT lookup procedure

The HTT lookup algorithm starts with a traversal of the DAT until a leaf bucket is reached. Next, the leaf bucket is traversed to identify the longest matching prefix.

The traversal of the density-adaptive trie consists in computing recursively the memory address of the child node matching a portion of the destination IP address. The lookup algorithm first parses the DAT node header. Then, an IP address segment is extracted and used to identify the child node encoding a sub-trie that contains the IP address. Next, the memory position of the matched child node is computed. The memory position of the matched child node is added to the base address, and then returned.

During the DAT traversal, at each level of the DAT, a different segment of the IP address is compared. Thus, to know which IP address bits to use, the number of IP address bits already compared is returned after each comparison. If a merged node is matched, then the lookup algorithm returns the previous number of bits already compared added to the number of bits to extract from the IP address. However, when a merged node is matched, the number of bits effectively matched is smaller than the number of bits extracted from the IP address. Because a merged node covers multiple regular nodes, the number of bits discarded is equal to the logarithm of the number of merged nodes. As a result, when a merged node is matched, the number of bits of the IP address already compared is equal to its previous value added to the number of bits effectively matched.

The DAT is traversed until a leaf bucket is reached. Then, the leaf header is parsed in order to retrieve the number of prefixes stored and the longest prefix length. After computing the position of each prefix in the memory word, prefixes are extracted. Next, all prefixes are matched against the unmatched bit of the destination IP address. Finally, the NHI of the longest prefix match is returned.

4.4 SHIP Lookup Engine Architecture

This section introduces the architecture of the SHIP lookup engine. First, an overview of the architecture is presented. Then, the pipeline stages of the HTT traversal module are covered.

4.4.1 Overview

The SHIP lookup engine architecture is tailored to support low-latency lookup operations. It is composed of two main modules : the ABB pointer selection module and the HTT traversal module. First, the ABB pointer selection module hashes the 23 MSBs of the destination IP address, and retrieves from the hash table memory a root pointer forwarded to the HTT Traversal module. The root pointer selects an ABB, which covers K PLS groups. Second, the HTT traversal module performs a parallel traversal on the K PLS groups, starting from the root pointer. Then, a priority resolution module returns the NHI associated with the longest prefix match.

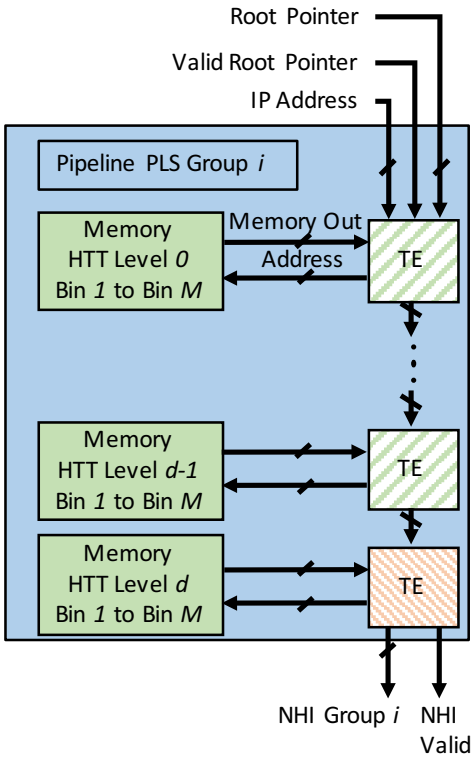


Figure 4.2 Overview of the i-th pipeline of the HTT traversal module.

To achieve low-latency lookup operations, the presented architecture leverages two characteristics of the SHIP data structure. First, because the data structure built by the SHIP algorithm is compact, only on-chip low-latency memory is used. By avoiding off-chip memory, the number of clock cycles is highly reduced for each memory access and shorter access time can be obtained. Second, the small number of levels in the SHIP data structure is used to create pipelines with a few stages. As a result, the latency per lookup is reduced, while one lookup result is outputted at each clock cycle.

In addition, the pipeline organization is tailored for efficient logic resource usage. The architecture of the HTT traversal module is organized in K pipelines. One of the pipelines of the HTT traversal module is shown in Fig. 4.2. Each pipeline $i \in \{1, ..., K\}$ performs the traversal of the HTT selected by the root pointer address held in the i-th PLS group. A pipeline is divided into d stages, such that each pipeline stage is dedicated to the traversal of a single level of a HTT. A pipeline stage $j \in \{1, ..., d\}$ is composed of a Traversal Engine (TE) and a memory, which holds the HTT nodes of level j of all the ABB (Bin 1 to Bin M). The traversal engines of the first $d - 1$ levels are used to process only DAT nodes, whereas LB nodes processing is only done at the level d.

The following sections detail the processing of the ABB modules and the traversal engines.

4.4.2 Address Block Binning

The ABB module implements the perfect hashing function presented by Czech *et al.* [113] on the IP Address. However, only bits $[4 : 23]$ are effectively used by the perfect hashing function. Indeed, this work only focuses on unicast IP addresses, which have their 3 MSBs fixed. In addition, the perfect hashing function was selected because of its low-complexity when used on data represented on a number of bits that is a multiple of 2. The ABB pointer is retrieved in the hash table memory in two steps. First, bits $[4 : 13]$ are used to index the memory that stores base addresses. Second, bits $[14 : 23]$ are added to the base address to index the ABB-pointer memory. Each entry in the ABB-pointer memory contains the 20 MSBs of the key used to create this entry, an ABB pointer, and K valid-root-pointer bits. Each of the K valid-root-pointer bits specifies whether to traverse the associated PLS group. Both the ABB pointer and the base-address memory are using low-latency on-chip memory.

4.4.3 HTT Traversal Pipeline Organization

The HTT traversal pipeline is organized to minimize the processing latency. For this purpose, not only low-latency on-chip memory is used at each pipeline stage but also the processing within a pipeline stage uses parallelization.

A single memory access is required per stage to read a DAT, while a leaf bucket node can be spread-out over multiple memory words. However, to avoid multiple memory accesses within a pipeline stage, a single memory word of a leaf bucket node is read per pipeline stage.

Moreover, the HTT traversal pipeline is organized for efficient logic usage by delaying the logic-costly processing of the leaf bucket nodes that is performed only at the last pipeline stage. Therefore, the TE represented with a green dashed line (stages 1 to d-1) in Fig. 4.2

processes the DAT nodes, whereas the TE represented with an orange dashed line (stage d) processes the leaf bucket nodes. When a memory word of a leaf bucket is extracted from the memory before the stage d, the memory word is forwarded to the next pipeline stage up to the d-th pipeline stage.

DAT node Traversal Engine

The architecture of a TE processing density-adaptive trie node is presented in Fig. 4.3a. To reduce the complexity of the figure, we omitted the module used to identify and forward a leaf bucket memory word to the next pipeline stage. We also omitted the address signal connected to the memory. The TE for a DAT is performed into 4 operations, completed into a single clock cycle.

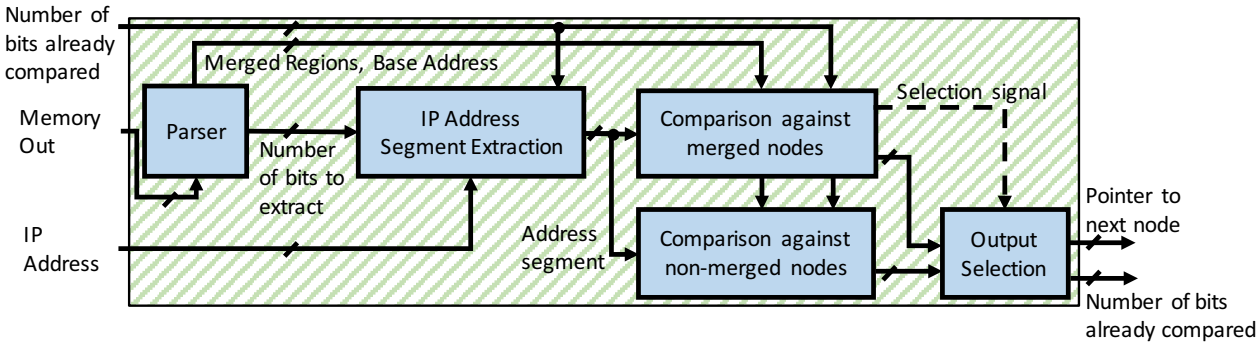
First, the parser extracts the DAT nodes parameters from the memory output : merged nodes indices, base pointer, and the number of bits to extract.

Second, an IP address segment is extracted. The size of this segment is equal to the input number of bits to extract, while the first bit to extract is set to the number of bits already compared.

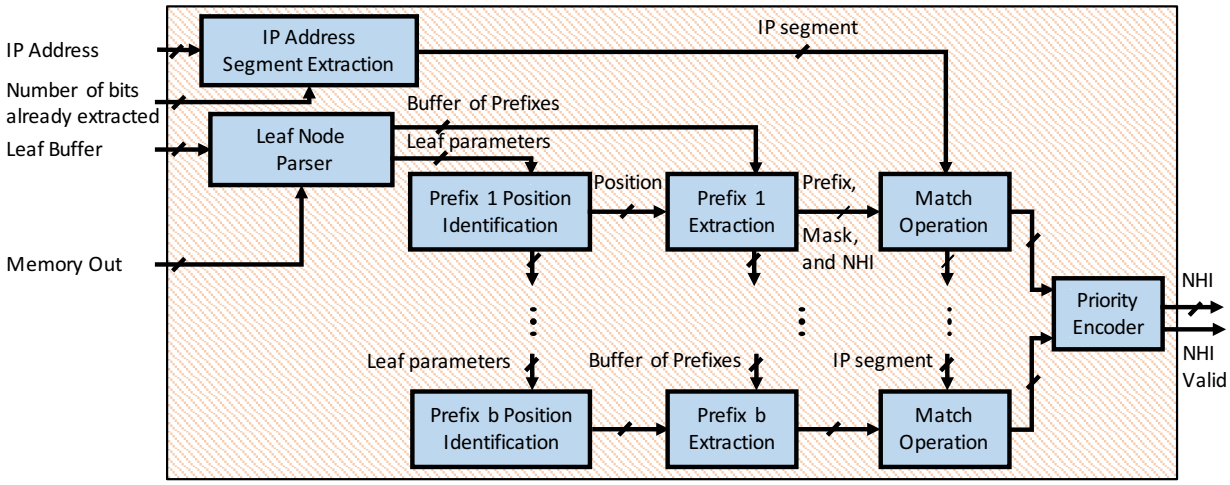
Third, the sub-trie covering the IP address segment is identified. The sub-trie identification process is parallelized to minimize the latency. The identification is done by comparing the IP address segment against the merged nodes indices stored in a DAT node. If the comparison is negative, then a regular node covers the IP address segment. The regular node covering the IP address segment is obtained using the merged node indices. The adopted architecture computes in parallel the pointer address to the children node, and the number of bits already compared for both the merged nodes and the regular nodes.

Fourth, the output selection module selects which address pointer to the children node and number of bits already compared to forward to the next TE. If a merged node was matched, the outputs of the merged node comparison module are selected. Otherwise, because only one node can be matched, the outputs of the regular node comparison module are forwarded.

If a memory word from a leaf bucket node is read from the memory, then the memory word is inserted in a leaf buffer. The leaf buffer is forwarded to the next DAT node TE, until the leaf bucket node traversal engine is reached.



(a) TE for a DAT node



(b) TE for a Leaf Bucket node

Figure 4.3 Traversal engines in the SHIP architecture

Leaf bucket node traversal engine

The architecture of the leaf bucket node processing module is shown in Fig. 4.3b. The leaf bucket node module computes the position of each prefix stored in a leaf, extracts the prefixes, and then returns the NHI associated to the longest prefix matching the IP address. To minimize the processing latency of a leaf bucket node, these five steps are parallelized for handling up to b prefixes stored in a leaf.

Because a leaf packs prefixes with variable mask length, the prefix positions in the buffer of prefixes is not fixed. As a result, the position of each prefix in the buffer of prefixes needs to be computed before prefix extraction. Two parameters given by the leaf node parser are used to compute the position of each prefix within a leaf : the number of prefixes held in the leaf and the largest prefix length stored in the leaf.

The leaf parser forwards to the prefix extraction module a buffer of prefixes containing all the prefixes stored in the leaf. The node parser outputs are extracted from the memory output if no valid leaf is held in the leaf buffer. Otherwise, the leaf buffer concatenated to the memory output is used. Then, each prefix extraction module shifts the buffer of prefixes by the computed prefix position value to retrieve a prefix, a prefix length and a NHI. Each extracted prefix is matched against the unmatched bits of the IP address segment. The IP address segment is obtained after shifting the destination IP address by the number of bits already extracted. Finally, a priority encoder receives the results of the parallel matches and returns the NHI associated with the longest prefix matched. A valid NHI signal is also returned to indicate whether a prefix was matched against the IP address, as shown in Fig. 4.3b.

Only the last stage of a HTT traversal pipeline has a leaf bucket node TE, although a leaf node can be read at any of the previous stages. The motivation to delay the processing of a leaf bucket node to the last stage is twofold. First, although a leaf bucket node can be divided into multiple memory words, the location of the memory word to read at each stage is given in the leaf header and does not require any processing. Thus, at each stage a memory word that is part of a leaf node can be concatenated to a leaf buffer. Second, the logic cost of the leaf bucket node TE is one order of magnitude higher than the DAT node. The higher logic cost relates to the use of b parallel barrel shifters to extract the prefixes held in a leaf. By delaying the processing of a bucket leaf node to the last stage of each pipeline, the logic cost reduction factor is on first approximation equal to $\frac{10 \cdot d}{d+10} = 4.4$, when the HTT pipeline depth $d = 8$ in the implemented architecture. The LB node TE is divided in two physical stages, because of the deeper levels of logic compared to a DAT TE.

4.5 Code Restructuring techniques for HLS

The presented architecture was described in C++, and synthesized with Vivado HLS. The main interest for describing a hardware architecture in C++ lies in the reduced development and verification time obtained by abstracting micro-architectural low-level details. Moreover, design evaluations generated with high-level synthesis tools closely matched the performance obtained with hand-written code using regular hardware description language (VHDL or Verilog) [119]-[122]. However, the main challenge consists of describing the architecture using a high-level language accurately.

Because the standard C++ library does not support bit-accurate data types, nor cycle-accurate models, Vivado HLS provides support for bit-accurate data types through the use of libraries. Cycle-accurate modelling constraints can be applied using a set of directives supported by the Vivado HLS compiler. Although Vivado HLS provides libraries and directives for an accurate hardware description, an algorithm coded in C++ cannot straight-out-of-the-box be synthesized into a high-performance architecture. The HLS compiler needs to be guided by the user to understand the targeted architecture. For this purpose, four code restructuring techniques were used to achieve the performance of the targeted architecture.

First, to respect bit accuracy, each C++ variable is redefined according to its bit size specification in the architecture. Similarly, typecasting is applied to intermediate results to enforce bit-accuracy description. Otherwise, a higher logic consumption is observed due to the usage of larger variables. Therefore, the size of all variables used in the presented architecture is precisely defined using an HLS library.

Second, to respect the architecture specification, directives are applied to enforce the implementation of specific modules such as memories, to describe the pipelined architecture, and to specify a maximum number of clock cycles for a module. For instance, we specify a latency of one for the first $d-1$ stages of the HTT traversal pipeline, while the d-th stage has a latency directive of two. A pipeline directive was used for the whole architecture to output one result per clock cycle. In addition, for each pipeline stage, we forced the use of low-latency on-chip memory (BRAM). All the parallel processing shown in Figures. 4.3a and 4.3b such as prefix position identification, prefix extraction and matching is described using the unroll directive on C++ *for* loop. However, latency and pipeline directives are constraints for the HLS tool that can be violated if the described architecture cannot fulfil the desired directives.

Third, to ensure directives are not violated, a specific HLS coding style is required. The coding style adopted should describe the behaviour of the expected hardware modules as closely as possible. Therefore, a pure software functionality description is highly unlikely to produce

the expected hardware module. For instance, relying on software code-reuse strategies to share a single hardware module with other hardware modules was shown to achieve the opposite goal. That is, the single hardware module that needed to be shared was replicated. Thus, when hardware modules with no data-dependency need to share a common hardware module, explicitly coding a multiplexer connected to the shared hardware module is required. Moreover, when multiple C++ behavioural descriptions of a module are possible, the one with the highest performance after implementation was selected. For instance, the priority encoder shown in Fig. 4.3b was described using a *for* loop, where the last entry matched overrides the previous value matched. In contrast, a priority encoder described using the C++ *switch case* construct achieved a lower performance. In addition, the coding style adopted assigns a value to an output signal only once. Otherwise, the HLS compiler interprets the multiple output assignments as a data dependency, which breaks the pipeline directive and increases the number of clock cycles elapsed before a new data can be processed.

Fourth, as a general rule of thumb, the clock period reported after synthesis under HLS is higher compared to the clock period achieved after implementation. Similar observation was made in [123]. This rule can be leveraged to reduce the number of stages of a pipeline architecture under HLS, while achieving after implementation a clock period close to the one targeted. Indeed, the clock period constraint takes precedence over the pipeline depth constraint. However, the clock period constraints need to be greater or equal to the longest pipeline stage propagation delay evaluated by the HLS tool to guarantee that the pipeline depth constraint is not violated. We used this observation to guide the HLS tool in order to synthesize a design with a small number of pipeline stages by relaxing the clock period constraint, while still achieving a small clock period after FPGA implementation. In addition, we observed that after implementation the minimum clock period achieved on the FPGA is below the clock period set under HLS by a factor ranging from a few percent up to 50%.

4.6 Performance Evaluation and Results

This section presents a performance evaluation of the SHIP lookup algorithm and its implementation on FPGA, and a comparison with recent work.

4.6.1 Benchmarks used for the evaluation

The performance of the lookup algorithm and its FPGA implementation uses 6 IPv6 prefix tables. The architecture is evaluated using SHIP data structures built with 6 IPv6 prefix tables, ranging from 25 k prefixes up to 580 k prefixes. The smallest prefix table, *rrcc*00 was

extracted from RIS remote route collectors [103]. For the other prefix tables used, synthetic prefixes were generated with a method that uses IPv4 prefixes to generate IPv6 prefixes, in a one-to-one-mapping [116]. The IPv4 prefixes used were also extracted from [103]. Using the IPv6 prefix table holding 580 k prefixes, four smaller prefix tables were created, with a similar prefix length distribution, holding respectively 290 k, 116 k, 58 k and 29 k prefixes.

4.6.2 SHIP lookup algorithm

The performance of the SHIP lookup algorithm was based on two metrics : the number of memory accesses to complete a lookup, and the memory consumption. The number of memory accesses reported is the sum of the number of memory accesses for the deepest hybrid trie-trees amongst all address block bins and the perfect hash table. The memory consumption is the sum of all HTT nodes and the perfect hash table. In order to evaluate the SHIP data structure efficiency, the memory consumption is divided by the number of prefixes stored. For both metrics, the number of prefix length group k was set to 2. The lowest value of k that experimentally minimizes both the number of memory accesses and the memory consumption was selected. Indeed, the value of k linearly impacts on the logic resource consumption.

The memory consumption and the number of memory accesses of the SHIP lookup algorithm are reported in Fig. 4.4, for the 6 benchmark prefix tables. The memory consumption ranges between 10.63 and 16.9 bytes per prefix for 580 k prefixes and 29 k prefixes, respectively. The memory consumption per prefix decreases logarithmically with the number of prefixes for synthetic prefixes. However, the real prefix table $rrc00$ leads to a smaller memory consumption per prefix than synthetic prefix tables holding up to 116 k prefixes. In the worst case, the real prefix table leads to data structure 35% more compact than with 29 k synthetic prefixes. The difference between the synthetic and real prefix tables relates to a different prefix distribution between real and synthetic prefixes for prefix tables holding up to 116 k prefixes. Indeed, real prefixes are more distributed along their LSBs, whereas synthetic prefixes are more distributed along their MSB. As a result, the average number of prefixes per ABB is smaller with real prefixes than with synthetic prefixes. Thus, each HTT built with synthetic prefixes holds very few prefixes, leading to a hybrid trie-tree holding a single leaf with part of the allocated node memory left unused. As a consequence, the memory consumption per prefix is higher for synthetic prefix tables up to 116 k prefixes than with the real prefix table. In contrast, the average number of prefixes per ABB with synthetic prefix tables holding more than 116 k prefixes is higher or equal to the real prefix table. Those observations demonstrate that the memory efficiency of the HTT increases with the prefix table size.

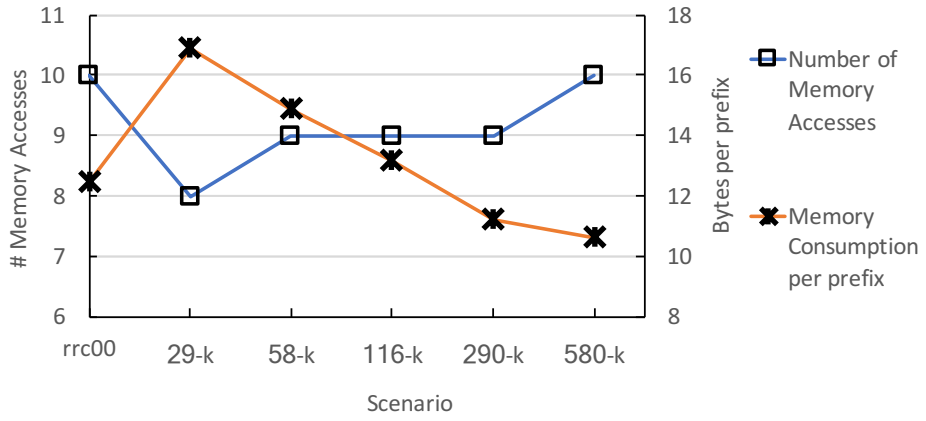


Figure 4.4 Memory consumption and memory accesses for synthetic and real prefix tables.

The number of memory accesses for all the prefix tables used is shown in Fig. 4.4 and ranges from 8 to 10. The number of memory accesses for synthetic prefixes increases logarithmically with the number of prefixes, up to 10. The number of memory accesses for the real prefix table is similar to the largest synthetic prefix table. The difference in the number of memory accesses between the real and synthetic prefix tables lies in the prefix distribution. Synthetic prefixes for scenarios with up to 116 k prefixes are distributed more uniformly along the MSBs, leading to an average number of prefixes per ABB smaller than real prefix tables. As a consequence, synthetic prefix tables with up to 116 k entries lead to shorter HTT, and thus, fewer memory accesses. However, the average number of prefixes per HTT is relatively similar between the two largest synthetic prefix tables and the real prefix table. Thus, the number of memory accesses between these three prefix tables differs only by one memory access.

4.6.3 FPGA Implementation

The presented architecture was synthesized with Vivado HLS 2017.1 and implemented using Vivado 2017.1 with Virtex-7 VX690TFFG1761-2 as a target. Based on the SHIP lookup algorithm performance evaluation, the maximum number of memory accesses for the 6 benchmarks used is 10. The architecture implemented on FPGA contains 11 pipeline stages, that is one pipeline stage per memory block, and an extra pipeline to complete the leaf node processing logical stage, which is divided into two pipeline stages. The proposed architecture has been tuned to minimize the total number of pipeline stages and consequently the lookup latency. Although a higher number of pipeline stages would reduce the clock period, and thus, would increase the number of lookups per second, architecture exploration to

support a higher bandwidth is outside of the scope of this work. For each design presented in Table 4.1, only the size of the data structure inserted in each pipeline stage memory is changed. When a pipeline memory stage is empty, we voluntary inserted a few entries in the memory to avoid design simplification during the implementation. As a consequence, the implementation results shown in Table 4.1 highlights the performance of the architecture with the memory utilization. Three performance metrics are of interest ; the latency, the on-chip memory usage (BRAM) and the logic usage. As the number of pipeline is fixed for all the implementations, the latency is determined only by the clock period.

For the implemented architecture, the slice usage grows linearly with the number of prefixes used, and the clock period decreases non-linearly with the number of prefixes. That is, when the BRAM usage increases, the slice usage increases, but the clock period decreases. Indeed, when the memory consumption increases, more BRAM are instantiated, which requires more logic to select the BRAM output. In addition, when a pipeline stage of a PLS group requires more BRAM than what is available in a single column of the FPGA, the memory is split over multiple BRAM columns. As a result, the route delay increases, because the presented architecture uses the unregistered BRAM output. Still, for the largest scenario, the lookup latency is 125.4 ns, which is very low compared to other work, as presented in Table 4.1, in the next section.

Because the clock period decreases with the number of prefixes, the proposed architecture handles 87.7 million packets per second (Mpps) for the largest scenario. However, using dual-port memories and by doubling the slice usage, the proposed design supports 175.4 Mpps, which is more than the 100 Gbps using an industry benchmark IPv6 packet size of 75 bytes [71]. In addition, the logical stage in each pipeline could be divided in more than one physical pipeline stage to increase the supported bandwidth at the cost of a higher latency. Exploring means of reorganizing the pipeline stages is beyond the scope of this work.

Tableau 4.1 SHIP architecture performance and cost for different table sizes.

Prefix table	# of prefixes	Latency (ns)	Period (ns)	BRAM 36-Kb	Slices
rrc00	25,900	105.6	9.6	96.5	7396
29 k	29,015	104.5	9.5	114.5	7313
58 k	58,054	105.6	9.6	214.5	7648
116 k	116,129	114.4	10.4	311.5	7969
290 k	290,360	119.9	10.9	734.0	8611
580 k	580,737	125.4	11.4	1416.0	9258

The BRAM usage presented in Table 4.1 for all the prefix tables is slightly higher than the algorithmic results shown in Fig. 4.4. Because the memory size is rounded up to the closest power of two, if the number of entries held in each pipeline stage memory is not close to a power of two, the memory efficiency decreases. Still, the memory consumption after implementation is on average around 10% higher compared to the results shown in Section 4.6.2.

Similar to other works [71], [118], if more prefixes need to be stored than the on-chip capacity, then using external memory is required. However, considering the large on-chip memory available on current FPGAs [124] combined with the high memory efficiency of the SHIP data structure, it is highly unlikely that external memories would be required.

4.6.4 Performance comparison

The performance of the proposed architecture is compared to other works in Table 4.2. For each work used as a reference, to normalize the results, the performance is reported for a single lookup engine. For all the works presented in Table 4.2, the latency in cycles was inferred from the published information, except for the 2-3 trees [71] where we reimplemented their work under Vivado HLS. In addition, for solutions using off-chip memory, the memory latency was evaluated using Altera documentation [125], because no similar documentation could be found for Xilinx Virtex-7 FPGA. For solutions based on DDR3-1600 memories [98], the round trip memory latency is equal to 64 clock cycles for a memory bus clock at 800 MHz, when the memory controller runs at 200 MHz. Thus, a DDR3 has an equivalent clock latency of 16 clock cycles at 200 MHz. For solutions using SRAMs [71], [118], the round trip memory latency is 10 clock cycles for a clock running at 200 MHz. The lookup latency for the evaluated works shown in Table 4.2 is detailed in Table 4.3, where the total latency is divided between the computation latency, the on-chip memory accesses, and the off-chip memory latency. However, for our proposed solution, the computation latency and on-chip memory latency are merged in Table 4.3 because the proposed architecture combines in a single clock cycle an on-chip memory access and computation. In addition, due to the lack of architecture details for the solution presented in [85], the computation latency and on-chip memory latency are also merged in Table 4.3.

The 2-3 trees architecture [71] has a latency estimated to 290 ns for a prefix table holding 330 k entries and a clock of 5.35 ns. The high latency of this solution relates to a long pipeline used to process the prefixes stored in the on-chip memory ; 16 clock cycles are spent to retrieve information from the on-chip memory, while 16 clock cycles are require to compute the result of each on-chip memory access. In addition, two external SRAM memory accesses

add 20 clock cycles of latency and 2 clock cycles to compute the result of each of the two off-chip memory accesses. The total lookup latency of the 2-3 trees architecture is 290 ns, which is 2.3× higher than our SHIP based architecture. Moreover, for a prefix table holding 580 k prefixes, the 2-3 trees architecture requires an extra off-chip memory access due to the increased tree depth. As a result, the estimated total latency increases by 11 clock cycles to 301 ns. Although the bandwidth supported by the 2-3 tree architecture is 2.1× higher than the proposed architecture, the former's logic consumption is 65% higher than our proposed architecture. Finally, the 2-3 tree architecture has a memory efficiency of 21 bytes per prefix, which is 87% higher than the one observed with our proposed architecture.

CLIPS [118] is presented as a memory-efficient IPv6 lookup solution, although it consumes 27.6 bytes per prefix. The estimated latency of the CLIPS implementation is 550 ns for 330 k prefixes, with a clock of 5.5 ns. The CLIPS architecture is composed of 4 lookup phases. The first lookup phase requires 10 clock cycles to access an external SRAM, 1 clock cycle to access an on-chip memory, and 2 clock cycles to compute the result of each memory access. Phases 2 and 3 require 17 and 15 on-chip memory accesses. For each memory access, one clock cycle of computation is required in phases 2 and 3. As a result, 32 clock cycles are required to read the memories and 32 clock cycles to compute the result of each memory access. The last phase requires 2 off-chip SRAM memory accesses, that is 20 clock cycles for the memory accesses and 2 clock cycles of computation. Thus, the estimated latency of CLIPS is 99 clock cycles, that is 544.5 ns for a reported clock of 5.5 ns [118]. Extending the CLIPS architecture for supporting 580 k entries would require 3 more memory accesses, due to the increased tree depth in phases 2 and 3. As a result, the CLIPS implementation has a latency 4.6× higher than the proposed architecture, while using more than 2.4× the number of bytes per prefixes. Still, a single lookup engine of CLIPS supports twice the bandwidth of the proposed architecture.

The performance of our architecture is compared against a new method presented by Rétvári *et al.* [85], which yields highly memory-efficient data structures. Although applied to IPv4 prefixes, the very low bit per-prefix ratio achieved by this approach is of interest in the perspective of minimizing the latency. Indeed, a smaller memory footprint means that the data structure can not only be stored in very low latency memory but also replicated to achieve a high bandwidth. Their method consumes only 178 kB of memory for 410 k IPv4 prefixes. Although, the latency of this solution is relatively low with respect to other works presented, the bandwidth supported is limited to 7 Mpps.

Tableau 4.2 Performance comparison with recent work.

Method	# of prefixes	Latency (ns)	Lookup Rate (Mpps)	BRAM 36-Kb	External Memory (Mbits)	Memory Efficiency (Bytes/prefix)	Memory Efficiency Relative to this work	Slices
2-3 trees [71]	332,118	290	188	580	32.5	21	+87%	15358
CLIPS [118]	322,000	544.5	180	N/A	63.13	27.6	+146%	N/A
FlashTrie [98]	318,043	295	84	108	307.2	124.2	+1008%	4583
[85][i]	410,513	142	7	40 [ii]	0	0.4	N/A	N/A
This work	580,737	125.4	87.7	1416	0	11.2	0	9258

[i] Work done using IPv4 prefixes.
[ii] Estimated BRAM consumption.

Tableau 4.3 Lookup latency breakdown.

Method	Computation Latency (cycles)	On-chip Memory (cycles)	Off-chip Memory (clock cycles)	Clock Period (ns)	Total Latency (cycles)	Total Latency (ns)
2-3 trees [71]	18	16	20	5.35	54	290
CLIPS [118]	36	33	30	5.5	99	544.5
FlashTrie [98]	9	2	48	5	59	295
[85]	7.1		0	20	7.1	142
This work	11		0	11.4	11	125.4

FlashTrie [98] divides a lookup into three phases. The first phase uses only on-chip memory, whereas the last two phases use DDR3 memory. The latency of phase 1 is estimated to 8 clock cycles ; 2 clock cycles for the on-chip memory accesses and 6 clock cycles for computation. Theoretically, a single external memory access is required for each of the phases 2 and 3. However, the equation used to model the number of memory accesses in phase 2 is incoherent with the results presented by Bando et Chao [105]. Using this model, two memory accesses are required in phase 2. Thus, 3 external memory accesses to DDR3 memories are required, where each external memory access to a DDR3 requires 16 clock cycles, and 3 clock cycles to compute the results of phases 2 and 3. Thus, the estimated lookup latency with the FlashTrie architecture is 59 clock cycles. Moreover, the bandwidth supported by the FlashTrie architecture drops to 88 Mpps when the maximum number of bank-activate commands that can be issued in a given period of time is considered. Lastly, FlashTrie requires an average of 124.2 bytes per prefix, because each data structure stored in the external memory is duplicated in each memory bank. To summarize, the FlashTrie architecture supports a slightly lower bandwidth, uses roughly half the FPGA resources, but consumes 11× more memory and requires 2.3× more time to complete a lookup.

4.7 Conclusion

Next generation networks such as 5G, and many future real-time applications require low-latency processing. In this paper, we proposed a fully pipelined IPv6 lookup engine architecture offering a low-latency and high memory efficiency. The proposed architecture exploits the characteristics of the SHIP data structure to support low-latency lookup operations with large prefix tables. To minimize the lookup latency, the pipeline stages exploit low-latency on-chip memories and parallel processing. In addition, the pipeline is organized for efficient logic resource usage. Code restructuring techniques were used to describe in C++ the proposed architecture in a form well adapted to high level synthesis.

The proposed architecture was implemented on a Virtex-7 FPGA. Evaluated with real and synthetic prefix tables holding up to 580 k IPv6 prefixes, the proposed architecture has a low lookup latency and high memory efficiency, while supporting the bandwidth of current Ethernet standards. Even for the largest prefix table, a lookup latency of 125.4 ns is achieved while requiring only 11.2 bytes per prefixes and supporting a bandwidth of 100 Gbps. Compared to the proposed IPv6 lookup architecture, other well-known approaches use at least 87% more memory per prefix, while increasing the lookup latency by a factor of 2.3×.

CHAPITRE 5 ARTICLE 3 - EFFICIENT LONGEST PREFIX MATCHING FOR PROGRAMMABLE DATA PLANES ON FPGAs

Auteurs : Thibaut Stimpfling, Jeferson Santiago da Silva, François-Raymond Boyer, JM Pierre Langlois et Yvon Savaria.

Soumis à IEEE/ACM Transactions on Networking le 9 mars 2020.

Abstract FPGAs are becoming ubiquitous in data centers to accelerate network functions. Such functions can be described in P4, a domain specific language, and compiled to a programmable data plane. However, efficiently exploiting FPGAs as programmable data planes remains an open question. In particular, match-tables, a key abstraction of P4, are inefficiently compiled to FPGAs for the Longest Prefix Match (LPM) operation. To address this issue, we present a framework that exploits a B-tree data structure to efficiently compile a match table configured for LPM on FPGAs. First, the B-tree is extended to support the LPM operation. Second, we present a method to allocate at compile time the resources used by the B-tree, while supporting update operations. Third, our framework selects the B-tree parameters maximizing the post implementation memory efficiency using an FPGA architecture description. Finally, the corresponding hardware architecture is generated. Compared to an open-source solution, our approach reduces the memory usage by 50%, while enabling a 6× clock frequency increase and reducing FPGA resource usage by an order of magnitude. Compared to a proprietary solution, our approach allows increasing the clock frequency by up to 30% while reducing the update complexity by up to 200×.

5.1 Introduction

The emergence of programmable data planes is revolutionizing the networking field as they allow to fully program a network. A programmable data plane is *configured* with a domain specific language, such as P4 [4], coupled to a domain specific architecture, such as the Protocol Independent Switch Architecture (PISA) that supports fast and flexible packet processing in hardware [5].

PISA comprises a parser, a pipeline of match-action tables and a deparser. Typically, a parser extracts packet headers, which are then used as lookup keys in the match-action pipeline. In a match-action stage, a lookup key is matched against rules stored in a match table. The lookup result, an action and associated data, is executed by an action stage, which can

modify the packet headers. Finally, a deparser reassembles the updated packet headers and emits the packet. A P4 program only *configures* the PISA architecture at compile time, but the match tables are filled at runtime.

While PISA was originally proposed for ASICs [5], recent efforts in the P4 community have focused on implementing PISA in FPGAs. Indeed, FPGAs are already adopted in data centers to accelerate network functions [6], [126]. P4 compilers for FPGAs have been developed [13], [14] along with PISA micro-architectures optimized for FPGAs [35], [41], [42].

Efficient mircroarchitectures have been proposed for parsers [41], [42], [127] and packet scheduler [35], but the match table microarchitecture performance lags behind (§5.8). Indeed, match tables require associative memories that are emulated on FPGAs because they are not available as built-in logic resources.

The match table abstraction encompasses exact match, Longest Prefix Match (LPM) and ternary match. Previous works have shown that exact match operations can be efficiently emulated using algorithmic solutions [5], [13], [14]. However, existing solutions emulating the LPM operation using an algorithmic approach [71], [72], [85], [87], [98] cannot be used in the context of programmable data planes on FPGAs because they require to know the match table entries before compilation, but match table entries are known only at runtime. Other works have used the transposed memory approach to emulate the LPM operation [18], [19], [128], but they achieved a low performance (§5.8).

Hence, in this paper, we address the following question : *Can the LPM operation be efficiently emulated using an algorithmic approach for programmable data planes on FPGAs ?*

While several data structures can be selected to implement LPM on FPGA, our solution exploits a B-tree, or balanced multi-way tree, because of its low space complexity and low update complexity (§5.2). Our solution is divided in a Prefix Candidate Selection (PCS) module and a Prefix Candidate Resolution module (PCR). The PCS uses a B-tree to identify a prefix candidate that may match a lookup key, while the PCR validates that the candidate prefix matches the lookup key (§5.3).

The resources used by a LPM match table for programmable data planes in FPGAs must be allocated at compile time, as the table entries are unknown at compilation time (§5.2.3). Hence, we present a static memory allocation model that allocates “just enough" ressources at compile time, while supporting update operations (§5.4). To map our solution onto the FPGA resources, given an FPGA architecture description, our framework selects the on-chip memories maximizing post-implementation memory efficiency (§5.6.4).

Finally,the hardware architecture is generated based on the parameters computed by our fra-

mework (§5.6). When implemented on FPGA, the memory efficiency of our solution increases with the table size (§5.7.2). In addition, our solution outperforms the transposed memory approach on all performance metrics (§5.7.3). When compared against a proprietary IP, our architecture can achieve a similar memory efficiency, but supports a higher frequency and reduces the update latency from 100× to 200× (§5.7.3). Lastly, we discuss two other match types that are supported by our solution (§5.9).

The main contribution of this paper is a framework to design and generate efficient algorithmic associative memories supporting LPM for programmable data planes on FPGA. Specifically, this paper proposes :

— A method to emulate an associative memory for LPM on FPGA using B-trees.
— A static memory allocation model that allocates “just enough" memory required for a given match table configuration.
— A method to identify at compile time the parameters of a B-tree maximizing the memory efficiency post-implementation for a given match table configuration.
— A tool that automatically generates a hardware description of the associative memory architecture.
— An open-source implementation of the proposed solution.

5.2 Background

We first describe the context of this work, namely match tables and the longest prefix match. Second, we cover the algorithmic approach used in this work to implement match tables on FPGAs. Finally, we present the B-tree data structure, which we build upon to implement a LPM match table.

5.2.1 Match Table

A match table is an associative array, i.e., an abstract data type composed of a collection of <key : action> pairs. A match table is described in P4 with four properties : a table size, actions, a key field and a match type [34]. A match table declaration is illustrated in Fig. 5.1. The table size is the table depth. The action property is a list of actions applied following a match, where an action is a function with optional parameters. The key field describes the lookup key matched against the table entries, while the match type specifies *how* the lookup key is to be matched. Three match types are supported in the P4 language, but only the LPM is considered in this paper. The two other match types are covered in the language specification [34].

While a P4 program describes the configuration of match tables, the P4 program does not describe the match table content. Instead, the match table content is added only at rune-time using an API, such as P4Runtime [129].

5.2.2 Longest Prefix Match

In the case of the LPM, the keys stored in the match table are called prefix. A prefix comprises an IP address and a prefix length, which specifies the IP address MSB bits to be matched against a lookup key. The LPM operation consists of identifying among a set of prefixes the one with the longest prefix length that matches an IP address. For instance, when matching a 4-bit $IP_{Dst} = 5$ against the prefixes presented in Fig. 5.1, both prefixes P_0 and P_1 are selected. Because P_0's prefix length is greater than P_1's prefix length, P_0 is the longest prefix matching $IP_{Dst} = 5$.

Traditionnaly, the LPM operation is used for IP forwarding [16]. However, in the context of programmable data planes, a LPM match table is a building block upon which networking applications, not limited to IP forwarding, can be build.

5.2.3 LPM Match Table on FPGAs

A LPM match table is implemented as an associative memory [5]. Such memories are not directly available and the must be emulated on FPGAs. To emulate an associative memory, our solution exploits an algorithmic approach, where the content of the match table is encoded in a data structure. We first explain why the resources used by a data structure implementing a match-table must be allocated at compile-time in the case of a programmable data plane on FPGA. The resources allocated for the data structure must be minimized. Thus, we we present the algorithmic complexity of the main data structure types that can be used to

```
1 table ipv4_lpm_match{
2     key = {ipv4.dstAddr : lpm;}
3     actions = {drop; route;}
4     size : 4096;
5 }
6 action drop(...)
7 {...}
8 action route(...)
9 {...}
```

Figure 5.1 An example of an LPM Match Table declaration in P4 where the IPv4 destination address is a lookup key. The action parameters are voluntarily omitted.

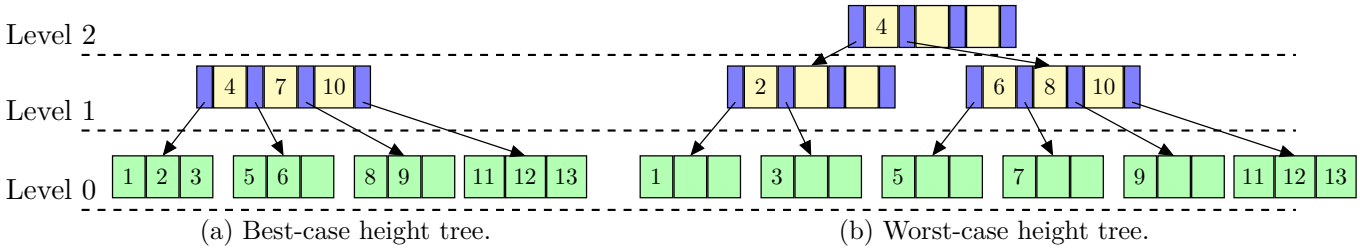
implement a LPM match table.

Resource Allocation at Compile Time. The FPGA resources necessary to implement the match table must be allocated at compile time using only the match table properties (§5.2.1) as the table entries are added only at run time [5]. As a result, in the context of programmable data planes on FPGAs, data structure optimizations exploiting table entry characteristics cannot be used. In addition, the data structure selected must exhibit a low space complexity, to maximize the table size implemented on FPGAs, and a low update complexity, to minimize the update latency.

Data Structures Algorithmic Complexity. Let N be the number of prefixes stored, and let W be the IP address size. We review the algorithmic complexity of tries, hash tables and tree when used for the algorithmic approach. A multi-bit trie where s bits are compared at each level has $O(N \cdot \frac{W}{s} \cdot 2^s)$ space complexity and $O(\frac{W}{s} \cdot 2^s)$ update complexity [100]. Similarly, a solution exploiting a hash table for LPM, inspired from the FlashTrie method [98], has $O(N \cdot (2^s) \cdot \frac{W}{s})$ space complexity and has $O(2^s)$ amortized update complexity, when the W address search space is divided into s bits segments. In contrast, a balanced binary tree has $O(N)$ space complexity, but it suffers from a $O(N)$ update complexity as the full tree must be rebuilt following an update. However, a B-tree has $O(N)$ space complexity, similar to a balanced binary tree, but has $O(log(N))$ update complexity [64]. Our solution exploits a B-tree, because it has the lowest space complexity and a low update complexity compared to tries and hash tables.

5.2.4 B-tree Data Structure

We present the B-tree data structure, we introduce the lookup procedure, and we describe the challenges when using a B-tree to implement LPM on FPGAs.



(a) Best-case height tree. (b) Worst-case height tree.

Figure 5.2 The B-tree height and the number of nodes per level depend on the key insertion order for a given key set. For a B-tree of order *t=2*, (5.2a) the insertion order $\{7, 4, 11, 2, 9, 5, 12, , 13, 1, 6, 8\}$ leads to best-case height tree, while (5.2b) the insertion order $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$ leads to a worst-case height tree.

Invariants. A B-tree is a balanced multi-way tree composed of three node types : a root node, leaf nodes, and non-leaf nodes. A B-tree of order t respects the following invariants :

— The root node has at least two children nodes if it is not a leaf node.

— A non-leaf node has at least t children nodes and has at most $2 \cdot t$ children nodes.

— A non-leaf node with k children nodes holds $k - 1$ keys.

— A leaf node holds $t - 1$ to $2 \cdot t - 1$ keys.

— Every path from the root to a leaf has the same depth.

Lookup Procedure. The tree is traversed iteratively from the root down to the leaf level. A lookup key L_{Key} is compared at each level against a collection of ordered keys$\{K_0, ..., K_j\}, j \leq 2 \cdot t - 1$ stored in the selected node to find the largest $i \in \{0, ..., n\}$ such as $K_i \leq L_{Key} < K_{i+1}$. When $L_{Key} < K_0$ a default child node is selected, else the i-th child node is selected.

Challenges for LPM on FPGAs. First, a B-tree cannot be used as is for LPM as the $<$ comparison operator, used in the B-tree lookup, cannot be extended to overlapped prefixes. Second, allocating the B-tree resources at compilation time is not straightforward as the B-tree height and the number of nodes depend on the key insertion and key deletion order, as illustrated in Fig. 5.2, which is unknown at compile time. Our solution, presented in the next section, jointly addresses these two challenges.

5.3 Proposed Solution

Our solution, shown in Figure 5.3, is divided into two modules : a prefix candidate selection (PCS) module, and a prefix candidate resolution (PCR) module. The PCS module selects a single candidate prefix that may match a lookup key, while the PCR module validates that the candidate prefix matches the lookup key and forwards the associated action. We first present the two modules, then we describe the lookup procedure.

5.3.1 PCS Module

The PCS module exploits a B-tree to selects the longest prefix that may match a lookup key. The B-tree holds only the lower bound of prefixes converted to non-overlapping intervals, as shown in Figure 5.3.

The PCS module is designed to alleviate the B-tree challenges(§5.2.4). First, because prefixes are converted to non-overlapped intervals, a B-tree can be used for LPM. Second, the FPGA resources allocated at compile support any prefix insertion and prefix deletion order (§5.5).

The result of the PCS design is twofold. First, because prefixes are non-overlapped, a lookup key matches either a single prefix or no prefix. Second, the prefix selected is a candidate prefix because the lookup key is matched only on the prefix lower bound, but not on the prefix upper bound, which is stored in the PCR module.

5.3.2 PCR Module

The PCR module validates whether the candidate prefix selected by the PCS module matches the lookup key. The PCR module holds the prefixes and actions in a memory, as illustrated in Fig. 5.3. The candidate prefix selected is matched on its upper bound against the lookup key. In case of a positive match, the associated action is returned.

5.3.3 Lookup Procedure

The lookup procedure is illustrated with the example shown in Fig. 5.3, where a lookup key $IP_{Dst} = 13$ is matched against prefixes P_0 to P_4, described in Table 5.1.

The lookup procedure begins in the PCS module, by a traversal of the B-tree, shown in Fig. 5.3. In this figure, in a B-tree node, a prefix lower bound is represented by a yellow rectangle for a non-leaf node and by a green rectangle for a leaf node. The lookup procedure is initiated at the B-tree root node (step ①), and the tree is recursively traversed top to bottom (step ②). At each level, the IP_{Dst} is compared against the valid sorted lower bounds LB_i to find $\{j = max(i), i \in \{1, .., 2 \cdot t - 1\} | LB_i \leq IP_{Dst}\}$. The index j is used to select the next child node to visit and to select a prefix candidate. A dashed rectangle in the PCS module represents the node entry selected at index j. In the case where there is no j satisfies the condition, a default pointer to a child node is selected, while the pointer to the prefix candidate from the last level is reused. When a leaf is reached, the pointer to the prefix

Tableau 5.1 Prefixes encoded on 4 bits and their conversion to non-overlapping intervals. When multiple prefixes span the same interval, the longest prefix takes precedence. Prefixes are represented using the format `Address` / `Prefix length`. The '$*$' symbol represents a wildcard bit.

Entry	**Action**	**Prefix**	**Interval**	**Non-Overlapping Intervals**
P_0	A_0	$010*/3$	[4,5]	[4,5]
P_1	A_1	$0***/1$	[0,7]	[2,3], [6,7]
P_2	A_2	$000*/3$	[0,1]	[0,1]
P_3	A_3	$10**/2$	[8,11]	[8,11]
P_4	A_4	$1***/1$	[8,15]	[12,15]

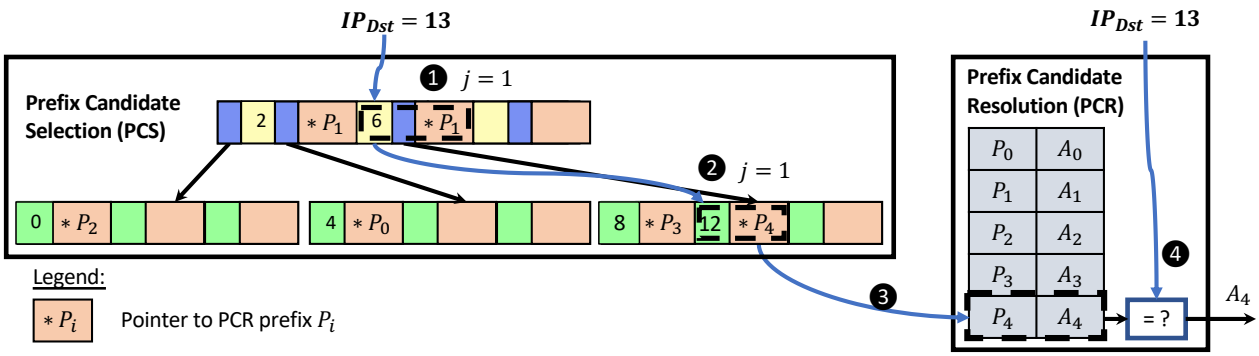


Figure 5.3 Proposed solution for the prefixes presented in Table 5.1. The PCS module uses a B-tree of order $t = 2$. The lookup steps for a lookup key $IP_{Dst} = 13$, ranging from ① to ④, are also shown.

candidate, here P_4, is forwarded to the PCR module (step ③). Then, IP_{Dst} is matched against the prefix candidate (step ④). In the case of a positive match, the action is returned.

5.4 Static Resource Allocation Model

5.4.1 Overview

In the context of a programmable data plane on an FPGA, the resources to implement a P4 program are allocated at compile time as explained in §5.2.3). Thus, using only the match tables properties, covered in §5.2.1, we present a static resource allocation model for the proposed solution. The model is designed to minimize the amount of resources allocated.

We first present our model for both the PCS and the PCR modules. Finally, we demonstrate why our allocation model does not require prior knowledge of the table entries.

5.4.2 PCS Module

The PCS module encodes the prefix lower bounds in a B-tree. To allocate statically the B-tree resources efficiently, the following questions must be addressed : 1) What is the maximum number of prefixes lower bounds stored ? 2) What is the minimum B-tree height ? 3) What is the minimum number of nodes per level ? In the analysis below, we consider a match table of size N and a fixed B-tree order t.

Maximum Number of Lower Bounds. Because the prefix characteristics are unknown at compile time, we consider the worst case scenario, where the N prefixes are overlapped two by two. In this case, the prefix conversion to non-overlapped intervals generates $M = 2 \cdot N - 1$

intervals. Hence, M prefix lower bounds can be inserted in the B-tree.

Height. The B-tree height h and the number of nodes per levels depend on the insertion order as illustrated in Fig. 5.2.

Because the insertion order is unknown at compile time, the resources must be allocated for a B-tree of the maximum height, a worst-case height h_{wc}. With this assumption, the theoretical bound is $h_{wc} = \left\lfloor log_t\left(\frac{M+1}{2}\right)\right\rfloor$ [64].

Nodes Per Level. The number of nodes allocated per level must support any insertion order. Our method allocates at each tree level the exact maximum number of nodes that can exist for any insertion order, i.e. "just enough". This is obtained by simulating the insertion of M prefix Lower Bounds (LBs) using a worst-case insertion order, which maximizes at each level the number of nodes that can be created. The worst-case insertion order used consists in inserting LBs in ascending order (§5.5.3), as illustrated in Fig. 5.2b.

Our allocation method allocates the minimum number of nodes required compared to a naive method where either all nodes are fully filled with $2 \cdot t - 1$ entries, or half-filled with $t - 1$ entries. In the former case, the number of nodes held at the level i is $(2t)^i$. Using this allocation method for the B-tree shown in Fig. 5.2b, the number of node allocated at the leaf level is 16, which is greater than the number of keys stored. In contrast, our method uses only 6 nodes. In the latter case, the number of nodes held at level i is $t \cdot (2)^i$. Using this allocation method for the B-tree shown in Fig. 5.2b generates 4 nodes at the leaf level, which is problematic as it is less than the 6 nodes required.

Node Size The node layout is presented in Fig. 5.4 for a non-leaf node. To respect the B-tree invariants (§5.2.4), a non-leaf node holds $2 \cdot t - 1$ prefix lower bounds LB_i, $2 \cdot t - 1$ pointers to the PCR prefixes, noted $*PCR_i$, and $2 \cdot t$ pointers to the B-tree children nodes, noted $*node_i$. A leaf node layout is similar to the a non-leaf node layout except that no pointers to children nodes are stored.

The proposed node layout does not encode whether a lower bound is valid. Here, the validity of a lower bound is inferred by comparing each lower bound against a default value, set to 0, representing an invalid lower bound. One extra bit is required, for a single leaf, for disambiguation in the case where a leaf holds a valid lower bound that has a value of 0.

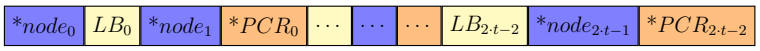


Figure 5.4 PCS non-leaf node layout.

5.4.3 PCR Module

The PCR module holds the N prefixes and actions inserted in the match table. Hence, the resource used by the PCR is the sum of the prefix size and the action size multiplied by N.

5.5 Update Support

The proposed solution supports *incremental* updates. The update rates to support ranges from 10 k to 100 k updates/s [45], [46], [130]. In our solution, updates are computed in software, and then pushed to the hardware architecture, similarly to solutions implementing ternary match tables and exact match tables [68], [97], [131].

We first present the update procedure for prefix insertion and prefix deletion, then we demonstrate theoretically that the proposed static allocation method supports any update operation.

5.5.1 Prefix Insertion

After converting a prefix into non-overlapping intervals, the resulting LBs are inserted in the PCS module, while the prefix is inserted in the PCR module. In the PCS module, a LB is inserted with a pointer to the PCR module, using the B-tree insertion procedure, at the leaf. The B-tree may then be rebalanced to respect its invariants. The complete B-tree insertion procedure is covered in [64]. The B-tree insertion complexity is $0(log(M))$ [64]. At most, two B-tree nodes are modified per level following an insertion. In the case where an LB insertion triggers a tree rebalancing up to the root node, the highest number of modified nodes is $2 \cdot h_{wc}$.

In the PCR module, inserting a prefix and its actions consists simply in a single write operation to the PCS memory, leading to an $0(1)$ insertion complexity. Hence, the insertion complexity of the proposed solution is $0(log(N))$.

5.5.2 Prefix Deletion

The LBs associated with the prefix to delete are iteratively deleted in the PCS module, and the prefix is deleted from the PCR module memory.

In the PCS module, the LB deletion may trigger a B-tree rebalancing to respect the B-tree invariants 5.2.4. The detailed B-tree deletion procedure is covered in [64]. The B-tree deletion complexity is $O(log(M))$ [64], and the highest number of B-tree nodes modified is $2 \cdot h_{wc}$, when a deletion triggers a tree rebalancing up to the root node. In the PCR, a single memory

write operation is required to delete the prefix. Hence, the deletion complexity of the solution is $O(log(N))$.

5.5.3 Proof

We demonstrate why sufficient resources are allocated by our model to support any update operation. The demonstration is limited to the PCS module, as the resource used depends on the insertion and deletion order, which is not the case for the PCR module. Due to the lack of space, we cover only the main ideas of the demonstration. For the demonstration, we consider a tree of maximal height, for which the number of nodes is maximal at each level (§5.4).

Prefix Insertion. We demonstrate by induction that the number of nodes is maximum at each level of the B-tree when prefix LBs are inserted in increasing order. When LBs are inserted in ascending order, LBs are inserted in the right most leaf at level 0. When, this leaf holds $2 \cdot t - 1$ LBs, the next insertion triggers a node split. That is, the median LB is promoted to the upper level, level 1, while the leaf is split into two leaves. The left leaf holds $t - 1$ LBs and the right one holds t LBs. All leaves but the right one will thus hold $t - 1$ LBs (0 extra LBs). Hence, all extra LBs are in the rightmost node, where the LBs are inserted. Thus, the number of split is maximized, maximizing also the number of leaves created. In addition, the LBs sequence promoted to level 1 is sorted in ascending order. Thus the same reasoning applies to level 1 and higher levels.

Sequence of Prefix Deletion and Prefix Insertion. We demonstrate by induction that the number of nodes deleted is always greater or equal to the number of nodes created. For this demonstration, we use Lemma 1.

Lemma 1. *Let us assume a B-tree where the number of nodes is maximal at each level. Let* $|n_{i,j}|$ *the number of LBs held in a node* i *at level* j*. Then,* $ExtraLB_j = \sum_i(|n_{i,j}| - (t-1)) \leq t, \forall j$.

The difference $(|n_{i,j}| - (t-1))$ represents the number of extra LBs stored in a node, as the minimum number of LBs in a node is $t - 1$. Lemma 1 can be demonstrated using a proof by contradiction, where we assume that $\text{ExtraLB}_j = t + 1$. Then, $t + 1$ LBs could be removed without deleting a node, but reinserting them in any single node would create a new node at level j, which violates the assumption that the number of node was maximal. Hence, by contradiction Lemma 1 holds.

A leaf is deleted when one LB is removed from a leaf holding $t - 1$ LBs, while both its left leaf and right leaf also hold $t - 1$ LBs. Then, the leaf to be deleted is merged with one of its

neighbour leaves and it holds $2 \cdot -2$ LBs. Let k be the maximum number of LBs removed to delete a single leaf. Using Lemma 1, at most t LBs can be removed without triggering a leaf deletion. Deleting one more LBs removes a leaf, which is then merged with its neighour leaf. The merged leaf holds $2 \cdot t - 2$ LBs. Then, $t - 1$ LBs can be deleted from the merged leaf without deleting the leaf, for a total of $2 \cdot t$ deleted LBs. After deleting $2 \cdot t$ LBs, all leaves holds $t - 1$ LBs. When the $2 \cdot t$ LBs are inserted in any leaf, a single new leaf is created. In addition, if more LBs were to be deleted and inserted in a different leaf, we can demonstrate by using a similar reasoning that the number of leaves created is at most equal to the number of leaves deleted. Thus, the theorem holds for the leaf level.

A node holding $t-1$ LBs at level $i+1$ is deleted when one of its child-nodes is deleted, while its right and left nodes at level $i + 1$ hold only $t - 1$ keys. Hence, by adopting a reasoning similar to the leaf level, but using the maximal number of nodes deleted at level i to delete a node at level $i + 1$, we can demonstrate that the theorem holds at level $i + 1$.

5.6 Hardware Architecture

5.6.1 High-Level Architecture

The proposed architecture, shown in Figure 5.5, is fully pipelined to maximize the lookup rate. The PCS and PCR modules are organized in processing elements (PEs) connected to dedicated on-chip memories. The architecture supports lookup and update operations, although only the lookup circuitry is shown in Figure 5.5.

Lookup. The PCS module implements a pipelined traversal of a B-tree. The traversal begins in pipeline level 0, where the on-chip memory holds the B-tree root node, and goes down to pipeline level h, where the on-chip memory holds the B-tree leaves. Pipeline stage n outputs a pointer to a PCR candidate prefix and a validity control signal. Then, the PCR candidate prefix is matched against the lookup key. An action is forwarded, including a validity control signal.

Update. Following a prefix update, the memory content to push to the PCS module and the PCR module is computed in software (§5.5). The memory content to write is encoded in a sequence of write bubbles [115]. A write bubble is a triplet {pipeline stage, memory address, memory word} that is inserted in the pipeline. When a write bubble reaches the pipeline stage specified in the triplet, a control circuitry updates the memory address with the new word. During an update, the pipeline is stalled, and no lookup operation can be performed. The maximum number of writte bubbles sent to insert a non-overlapped range is $2 \cdot h_{wc} + 1$. Hence, the worst-case update latency is $2 \cdot h_{wc} + 1$.

5.6.2 PCS Module Architecture

At each level of the tree, except the bottom level, a PE implements three operations : matching the lookup key against the prefix lower bounds, identification of the next child node to traverse, and selection of a prefix candidate. The Figure 5.6 shows the PE architecture for a non-leaf node.

Key Matching. The lookup key S_k is compared in parallel against the valid prefix lower bounds ; a lower bound set to 0 is deemed invalid (§5.4.2). Then, a priority encoder selects the index $\{j = max(i), i \in \{1, .., 2 \cdot t - 1\} | L_i \leq S_k\}$.

Child Node Pointer Selection. The index j identified in the key matching module is used to select the j-th entry in the child node pointer array. If no lower bounds are matched, the default child node pointer is selected.

Prefix Candidate Pointer Selection. The index j identified in the key matching module is used to select the j-th entry in the prefix candidate pointer array. Otherwise, the prefix candidate pointer from the previous tree level is selected.

5.6.3 PCR Module Architecture

The prefix candidate pointer forwarded by the PCS module is dereferenced in the PCR module memory. Then the prefix is matched against the lookup key. An action and associated data, and a validity control signal are forwarded.

5.6.4 Post Implementation Memory Efficiency Maximization

No assumptions about the FPGA resources available were made in the static allocation model (§5.4), and for the on-chip memories used in the PCS and PCR modules, shown in Fig. 5.5. However, in current high-end FPGAs, numerous on-chip memory types are available with different densities and configurations. For instance, with Xilinx's FPGAs, flip-flops, distributed RAMs, Block RAM, and Ultra RAM are available [124]. Our method uses a description of the on-chip memory available in an FPGA to identify the memory type maximizing the memory efficiency of our proposed solution. The memory efficiency, noted M_{eff}, is defined as the ratio of the amount of information stored in a memory to the size of the memory holding this information.

PCR Module A single memory is used to store the prefixes, actions and associated data (§5.3). Hence, we used a function that given an abstract memory width and memory depth identifies the memory type, memory configuration, and number of memory primitives maxi-

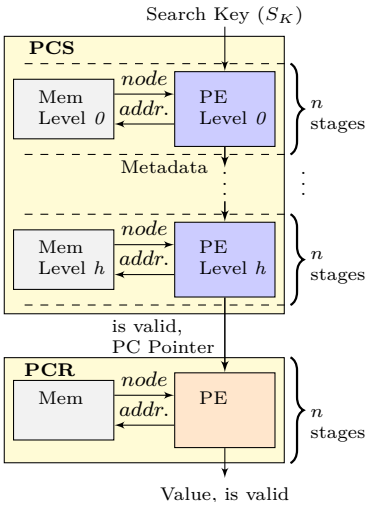


Figure 5.5 High-level architecture. The PCS module is pipelined by level of the B-tree, where a level is divided over n pipeline stages.

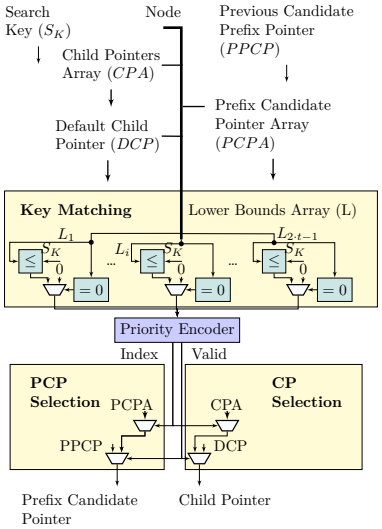


Figure 5.6 A PCS Processing Element (PE) for a B-tree of order t. The valid control signal is not shown.

mizing M_{eff}. A threshold value, determined experimentally, is set for the maximum memory size that can be built ouf of flip-flops and distributed RAMs to avoid having a low clock frequency.

PCS Module The memory efficiency post-implementation is not only impacted by the memory type used per tree level but also by the tree order t. The proposed method uses an exhaustive search to identify both the tree order and the memory type per tree level maximizing M_{eff}. Although the proposed method has a time complexity of $O(t \cdot h \cdot |memory\ \ types| \cdot |memory\ \ configuration|)$, in practice the order t and the tree height h remain low (§5.7). Hence, the computation time of the presented model is low compared to the implementation time. The B-tree order and the memory types and configuration selected for each tree level are used to generate the hardware architecture.

5.7 Experimental Evaluation

5.7.1 Experimental Setup

We first evaluated the performance scalability of our solution with a table size ranging from 4 k entries to 128 k entries, and with a prefix size ranging from 32 bits to 128 bits (§5.7.2). We measured the memory efficiency, the FPGA resource usage, and the clock frequency. For each configuration, the tree order t selected by our model, the memory efficiency predicted by our model, Mod. M_{eff}, and the experimental memory efficiency, Exp. M_{eff}, are presented in Table 5.2. The generated architecture is implemented using Vivado 2019.1 targeting a Virtex Ultrascale+ FPGA (xcvu9p-flga2104-3-e).

Then, we compare the performance of our method against a proprietary LPM IP from Xilinx [97] and an open-source solution, based on the transposed-RAM approach [18] (§5.7.3). To the best of our knowledge, these are the only two solutions that can be used in the context of programable data planes on FPGAs. To provide a fair comparison with previous work, we implemented our solution on a Virtex-7 FPGA (xc7vx690tffg1761-2). We also evaluated the lookup latency (LL), the worst-case update latency (UL), and reported whether the solutions support non-blocking updates. A solution is deemed to support non-blocking updates if both a lookup operation and an update operation can be scheduled at the same cycle. To evaluate the update latency of our solution, we assumed that update messages are transmitted over a 256-bit wide bus connected to the LPM solution. The update latency is the maximum number of write bubbles multiplied by the size of the write bubble divided by the bus width. Xilinx solution implements a binary search tree [97]. In addition, because the lookup latency reported by Xilinx is $LL = log_2(N) + K$, where K is a constant, we can infer that Xilinx

IP is based on a binary balanced tree. Hence, following an update the whole tree must be rebuilt, as supported by their software documentation [97].The parameters selected for this set of experiments are summarized in Table 5.2.

5.7.2 Scalability Evaluation

We first analyze the experimental memory efficiency reported in Table 5.2. We observe that the memory efficiency increases with the table size, as predicted by our model. The lower memory achieved on tables with fewer than 16 k entries relates to the mapping of some level of our solution to BRAMs that are only partially filled. We observe in Table 5.2 that the difference between the expected memory efficiency reported by the model against the experimental memory efficiency is less than 3% on all scenarios evaluated.

We now evaluate the clock frequency and the resources used by our solution. Figure 5.7a shows that the maximum frequency decreases almost linearly with respect to the table size. Larger tables require more BRAMs spanning multiple physical memory columns, which increases net delay and reduces clock frequency. However, the table size has little impact on the FFs/LUTs usage. Ultra RAMs are selected by our model for key sizes of 32 bits and 128 bits, which decreases the number of BRAM instantiated and increases the clock frequency, as shown in Fig. 5.7b. In contrast, for a key size of 64 bits, we observe a higher BRAM consumption leading to a lower frequency. The significant augmentation of FFs is due to the growing node size as they directly impact the size of pipeline registers. Similarly, larger keys means larger comparators, which explains the increased LUT usage.

5.7.3 Comparison with Previous Works

The results are summarized in Table 5.3. Compared against Xilinx solution [97] with non-blocking update support, our solution has a similar memory efficiency, but the frequency is increased by 30%. However, our solution consumes 2× more FFs and has higher latency due to a deeper pipeline. Our solution uses more LUTs than Xilinx solution, because multiple keys are compared in parallel in a B-tree level, while a single key is compared in a binary tree level. Yet, the UL of our solution is reduced by 100× over Xilinx solution. Indeed, a balanced binary tree is fully rebuilt following a prefix update. Evaluated against Xilinx's IP with blocking update support, our solution reduces the UL by over 200× at the cost of a memory efficiency increased by 2×. Indeed, memory is over-allocated in the PCS module to support incremental updates. Using the UL we derive that the update rate supported by Xilinx ranges from 3 k to 12 k updates/s, which is barely the lowest update rate required for some network applications 5.5. Lastly, Xilinx's IP is limited to a maximum of 64 k prefixes,

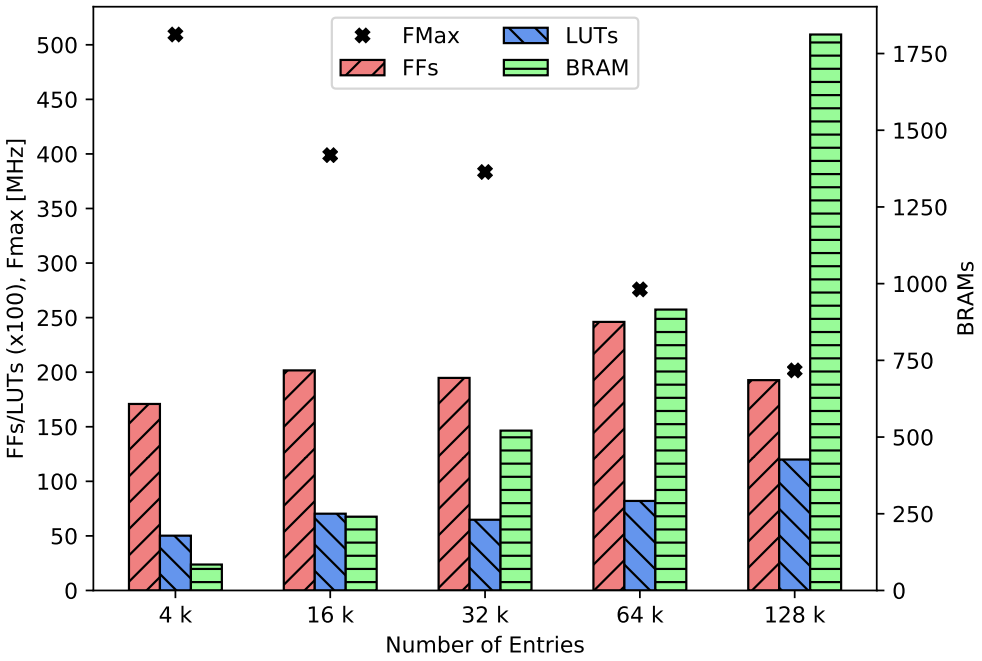
Tableau 5.2 Scenarios evaluated. We also reported the order and the tree height selected by our model, as well as Mod. M_{eff} the expected memory efficiency reported by our model, and Exp. M_{eff},the experimental memory efficiency.

Number Entries	**Prefix Size**	**Action Size**	**Order**	**Tree Height**	**Mod. M_{eff}**	**Exp. M_{eff}**
i) Comparison Scenarios						
4 k	32	16	10	4	0.089	0.116
16 k	64	32	9	5	0.152	0.176
32 k	128	64	9	5	0.182	0.181
64 k	36	0	8	5	0.123	0.119
ii) Scalability Scenarios						
4 k	64	32	6	5	0.095	0.126
16 k	64	32	9	5	0.152	0.176
32 k	64	32	9	5	0.164	0.163
64 k	64	32	8	6	0.195	0.186
128 k	64	32	8	6	0.201	0.188
64 k	32	32	8	6	0.207	0.216
64 k	128	32	8	6	0.186	0.191

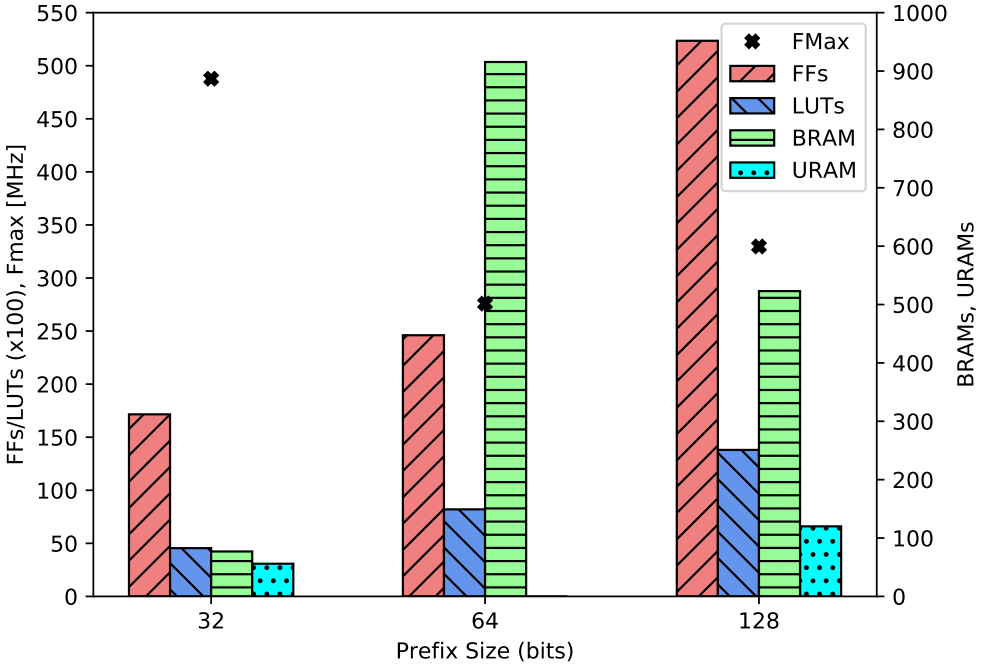
Tableau 5.3 Comparison against previous works.

Work	**M_{eff}**	**LUT [K]**	**FF [K]**	**Bram**	**Fmax [MHz]**	**LL[1]**	**U[2]**	**UL[3]**
Entries = 16 k, Prefix Size = 64 bits, Action Size = 32 bits								
[97][4]	0.18	3	9	234	191	23	Y	16k
Our	0.18	7	20	240	244	29	N	81
Entries = 32 k, Prefix Size = 128 bits, Action Size = 64 bits								
[97][4]	0.40	3	10	420	170	24	N	64k
Our	0.18	33	20	950	190	30	N	97
Entries = 64 k, Prefix Size = 36 bits, Action Size = 0 bits								
[18][4,5]	0.08	340[6]	N/A	1500	40	8	N	512
Our	0.12	43	16	553	245	34	N	73

[1] Lookup Latency in cycles.
[2] Non-blocking update support.
[3] Update Latency in cycles.
[4] The M_{eff} reported accounts only the block RAMs.
[5] Implemented on an Altera Stratix V.
[6] Number of ALMs.



(a) Prefix size and action size are fixed to 64 bits and 32 bits, respectively.



(b) The number of entries is set to 64 k.

Figure 5.7 Resources required to implement the proposed solution on the scalability scenarios.

which is not the case for our solution.

Evaluated against the largest scenario reported in [18], a transposed-RAM based solution, our solution improves the performance on all metrics. In particular, our solution improves the memory usage by 50%, the clock frequency (the lookup rate) by 6×, reduces the FPGA resource usage by an order of magnitude, and reduces the update latency by almost 6×.

5.8 Related Work

Transposed-RAM. The memory buses are *transposed* with this approach : the address bus is used as a data bus, while the data bus returns a one-hot encoded match-line value [18], [19], [132]. A priority encoder returns the matched line for the longest prefix matched. To emulate an $W \times N$ associative memory, a $2^N \times W$ on-chip memory is required. Recent works have focused on building large associative memories out of small transposed RAM primitives [18], [19]. However, to maximize the memory efficiency, shallow and wide on-chip memories would be required [18], [19]. Hence, the memory efficiency achieved using current FPGA architectures remains low.

Algorithmic Solutions. Tries [87], [98], compact and succinct data structures [85], trees [133], and B-tree [71], [72] have been proposed to implement LPM on FPGAs. However, these solutions are tightly coupled to known prefix sets. The data structures are filled offline with a prefix set, then they are implemented on FPGA. No static memory allocation method is presented with update guarantees. Thus, they cannot be used in the context of programmable data planes on FPGAs. One solution using B-trees for exact match [134] proposed to allocate 2^i nodes at level i, but no guarantee can be provided on the number of keys that can be stored (§5.4).

5.9 Discussion

Support for other Match Types. Our solution can be used to implement both exact match and range match operations. In the former case, the main benefit of our solution lies in its low update latency (§5.6) compared to a cuckoo hashing, the defacto data structure for exact match, which can require hundreds of memory accesses to insert a new key [68]. The flip side lies in the lower memory efficiency of our solution over a cuckoo hashing. In the latter case, because our solution encode prefixes as ranges, range match are natively supported. Traditionnaly, range match is inefficiently implemented on top of TCAMs [19], [135], which suffer from a very low memory efficiency on FPGAs [19].

5.10 Conclusion

We presented a framework to generate algorithmic associative memories for LPM on FPGAs. The framework leverages a B-tree where prefixes are converted into non-overlapping ranges to support the LPM operation. Although the B-tree resource usage depends on the prefix insertion order, we presented a method to allocate "just enough" resources for the B-tree at compile time, while supporting update operations. In addition, the framework selects the B-tree order and on-chip memory types to maximize the memory efficiency when mapping our solution to FPGAs. Compared to the state-of-the-art open-source solution, our approach reduces the memory usage by 50%, while enabling a 6× clock frequency increase and reducing the FPGA resource usage by an order of magnitude. Compared to a proprietary solution, our approach allows increasing the clock frequency by up to 30% while reducing the update latency by up to 200×.

CHAPITRE 6 DISCUSSION GÉNÉRALE

Les solutions présentées dans les chapitres 3, 4 et 5 contribuent à améliorer l'efficacité de l'opération LPM sur FPGA, répondant ainsi à la problématique étudiée dans cette thèse. Les solutions présentées dans les chapitres 3, 4 ont été conçues pour les contraintes de routage de paquets IPv6 dans le réseau Internet, tandis que la solution couverte dans le chapitre 5 a été développée pour les contraintes de plan de données programmables.

Le contexte du travail présenté dans les chapitres 3 et 4 découle d'un projet mené conjointement avec le groupe de recherche d'Ericsson Montréal, visant à réduire la consommation mémoire et la latence de recherche d'une solution de recherche LPM, dans le cas spécifique du routage de paquets IPv6 dans le réseau Internet. La solution proposée adopte une approche algorithmique, où une structure de données est utilisée pour identifier le plus long préfixe associé à une adresse IP de recherche. Dans le chapitre 3, l'emphase est mise sur la structure de données SHIP, tandis que le chapitre 4 présente l'architecture matérielle implémentant la traversée de SHIP.

L'idée principale de SHIP, présentée dans chapitre 3, consiste à exploiter les caractéristiques des préfixes contenus dans un FIB pour créer une structure de données compacte. Premièrement, SHIP adopte une approche «diviser pour régner», permettant de séparer les préfixes en groupes de faible cardinalité et partageant des caractéristiques similaires. Deuxièmement, SHIP utilise une structure de données hybride pour encoder efficacement les préfixes ayant des caractéristiques similaires, contenus dans un groupe. Bien que l'emphase du chapitre 3 porte sur la structure de donnée SHIP, une implémentation de l'architecture matérielle couverte dans le chapitre 4 y est présentée. Cette implémentation est conçue pour augmenter le débit de recherches, réduire la latence de recherche, et peut soutenir un débit proche de 300 Gb/s.

L'architecture matérielle et les principes de conceptions utilisés sont l'objet du chapitre 4. La faible empreinte mémoire de SHIP et la régularité du traitement effectué lors d'une opération de recherche permettent la conception d'une architecture matérielle efficace implémentant la traversée de SHIP. Ainsi, l'architecture proposée est entièrement pipelinée, utilise uniquement de la mémoire sur puce à faible latence et est organisée afin de réduire la consommation en ressources logiques. L'architecture est générée à partir d'une description de haut niveau. Plusieurs techniques, présentées dans le chapitre 4, sont utilisées afin d'augmenter la performance d'une architecture générée par un outil de synthèse de haut niveau. Finalement, une implémentation de l'architecture conçue pour réduire la latence de recherche et capable

de soutenir 100 Gb/s est couverte dans le chapitre 4. Comparativement à l'implémentation présentée dans le chapitre 3, la latence de recherche est réduite par un facteur ≈ 1.7 à 3.1.

Depuis les travaux réalisés spécifiquement pour l'opération de routage de paquets IPv6 dans le réseau Internet (3 et 4), le langage P4 [4] et les plans de données programmables PISA [5] ont pris une place majeure dans le monde de la réseautique. Ainsi, le chapitre 5 présente une solution LPM développée spécifiquement pour les contraintes des plans de données programmables sur FPGA. En effet, la majorité des solutions conçues pour le routage de paquets IPv6, incluant les solutions présentées dans les chapitres 3 et 4, exploitent les caractéristiques des préfixes contenus dans un FIB pour réduire la consommation de ressources logiques. Néanmoins, dans le contexte des plans de données programmables sur FPGA, les ressources logiques allouées pour une solution LPM doivent être indépendantes des caractéristiques des préfixes contenus dans un FIB [1].

Ainsi, l'objet du chapitre 5 est le développement d'une solution LPM ayant une faible consommation en ressources logiques indépendamment des caractéristiques des préfixes contenus. La solution proposée adopte l'approche algorithmique et exploite un arbre B. Une méthode est présentée permettant d'allouer le minimum de ressources requises pour l'arbre B, indépendamment des caractéristiques des préfixes contenus, tout en permettant d'effectuer des mises à jour incrémentales. La validité de la méthode d'allocation des ressources est démontrée mathématiquement. Par ailleurs, une méthode est présentée pour réduire la consommation de ressources logiques lorsque la structure de données est implémentée sur FPGA. L'architecture matérielle présentée est pipelinée et pour l'ensemble des scénarios évalués soutient un débit supérieur à 100 Gb/s. L'architecture est générée automatiquement et nécessite de connaitre uniquement la taille d'un FIB et les ressources logiques disponibles sur le FPGA ciblé.

Considérant que la solution présentée dans le chapitre 5 peut être utilisée pour n'importe quelle application, y compris pour du routage de paquets IPv6, il est intéressant de quantifier l'écart de performance par rapport aux solutions présentées dans les chapitres 3 et 4, qui font levier sur la connaissance des caractéristiques des préfixes contenus dans un FIB. En particulier, l'efficacité mémoire de SHIP post implémentation varie entre $\approx 50\%$ et 75%, là où la solution générique proposée varie entre $\approx 9\%$ et 20%. Ainsi, le surcoût d'une solution générique est d'environ $8\times$. Le coût d'une solution de LPM utilisant uniquement la logique programmable d'un FPGA peut être comparé au coût d'intégration d'une TCAM dans un FPGA. Tel que présenté dans le chapitre 2, le surcoût d'intégration d'une mémoire TCAM

1. Les seules caractéristiques considérées sont le nombre de préfixes à stocker, la taille d'un préfixe, et la taille de l'information associée à un préfixe

est d'environ 8× le coût d'intégration de mémoire SRAM. Par conséquent, en considérant uniquement le surcoût mémoire d'une solution LPM, les travaux présentés dans cette thèse tendent à indiquer que l'intégration de TCAM n'est pas requise dans un FPGA.

Nous présentons maintenant les travaux complémentaires réalisés durant cette thèse, mais qui ne constituent pas la contribution majeure de celle-ci [8], [20], [21].

Dans cette thèse, nous avons considéré que les FPGA étaient utilisés comme des NIC programmables accélérant l'exécution d'opérations réseaux, incluant l'opération LPM. Cependant, nous pouvons nous poser la question de la place des FPGA comme plan de données programmable : *Est-ce que les FPGA peuvent être utilisés au-delà des NIC ? Est-ce que l'architecture actuelle des FPGA est adaptée aux plans de données programmables ?*. Ces deux questions sont étudiées dans des travaux complémentaires effectués durant cette thèse [8], [20], [21].

Premièrement, nous nous sommes intéressés à l'architecture de plan de données programmables hétérogènes, comprenant des ASIC et des FPGA [8], [20]. La motivation associée à l'utilisation d'une architecture hétérogène est d'augmenter les capacités de traitement d'un ASIC par l'ajout de FPGA. Toutefois, un des défis majeurs d'une telle architecture hétérogène est la différence de débit soutenu par chacune de ces deux plateformes. Là où un ASIC supporte une dizaine de Tb/s, un FPGA est limité une centaine de Gb/s. Une telle architecture hétérogène risque donc d'être limitée par le débit soutenu par un FPGA. Pour réduire l'impact de ce problème, une solution de cache est proposée, visant à maximiser le trafic traité sur l'ASIC, tout en profitant des bénéfices d'une architecture hétérogène [20].

Deuxièmement, nous avons étudié les limites de performances d'un plan de données programmable implémenté sur FPGA [21]. Nous montrons que la performance de plusieurs blocs de l'architecture PISA [5] est limitée par l'architecture actuelle des FPGA. Pour pallier à ces limitations, deux avenues sont explorées. Premièrement, nous identifions un sous-ensemble d'applications réseaux n'utilisant pas ces blocs souffrant d'une faible performance, pouvant faire levier sur l'architecture actuelle des FPGA. Deuxièmement, nous proposons de spécialiser l'architecture des FPGA afin de supporter un plus grand ensemble d'applications réseaux.

CHAPITRE 7 CONCLUSION

Les FPGA sont massivement déployés dans les centres de données afin d'accélérer un grand éventail d'applications, allant de l'apprentissage machine jusqu'aux réseaux. Dans cette thèse, nous nous sommes intéressés à l'accélération de l'opération LPM sur FPGA, considérant que cette opération est essentielle pour de nombreuses applications réseaux. La problématique étudiée est l'implémentation efficace de l'opération LPM sur FPGA. Cette problématique est abordée dans deux contextes. Ainsi, la première partie de cette thèse est consacrée au routage de paquets IPv6 dans le réseau Internet, tandis que la seconde partie est consacrée aux plans de données programmables. Nous décrivons maintenant une synthèse des travaux, puis nous étudions les limites des travaux effectués. Finalement, nous discutons des avenues de recherche futures.

7.1 Synthèse des travaux

Les solutions présentées dans cette thèse adoptent l'approche algorithmique, où une structure de données est utilisée pour implémenter l'opération LPM.

Pour le contexte de routage de paquets IPv6 dans le réseau Internet, nous avons proposé SHIP, une structure de données compacte, qui exploite les caractéristiques des préfixes. SHIP est construite autour de deux idées principales. Premièrement, une approche «diviser pour régner»est utilisée pour séparer les préfixes en groupes de faible cardinalité et partageant des caractéristiques similaires. Deuxièment, SHIP utilise une structure de données hybride combinant la force d'une arbre préfixe et de larbre binaire pour encoder efficacement les préfixes contenus dans chaque groupe. L'arbre préfixe est utilisé pour identifier en quelques niveaux un sous-ensemble de préfixes pouvant générer une comparaison positive par rapport à une adresse IPv6. Toutefois, les arbres préfixes sont inefficaces pour identifier le plus long préfixe associé à une adresse IP parmi un sous-ensemble d'une dizaine de préfixes. Ainsi, lorsque le nombre de préfixes couvert par une branche de l'arbre préfixes est inférieur à une valeur seuil fixée, ces préfixes sont simplement encodés dans une unique feuille d'un arbre binaire.

L'architecture matérielle implémentant la traversée de SHIP est construite autour de trois principes. Premièrement, la traversée de SHIP est pipelinée par niveau, afin d'augmenter le débit de recherches. Deuxièment, considérant la faible empreinte mémoire de SHIP, uniquement de la mémoire sur puce à faible latence est utilisée afin de réduire la latence de

recherche. Finalement, afin de diminuer la consommation en ressources logiques, l'opération la plus coûteuse, pouvant être exécutée à chacun des niveaux du pipeline, est différée jusqu'au dernier niveau du pipeline. Deux implémentations de l'architecture proposée ont été présentées, la première, conçue pour atteindre une faible latence de recherche, et la seconde, pour atteindre un débit de recherches élevé. Ces deux implémentations supportent des débits supérieurs à 100 Gb/s.

Pour le contexte de plan de données programmable, nous avons présenté une solution efficace, indépendamment des caractéristiques des préfixes considérés. Cette solution exploite un arbre B, en raison de sa faible complexité algorithmique d'espace, de recherche et de mise à jour. Le coeur de cette solution est une méthode permettant d'allouer statiquement, sans connaissance préalable des caractéristiques des préfixes, le minimum de ressources requises, tout en permettant d'effectuer des mises à jour incrémentales. La validité de la méthode d'allocation des ressources est démontrée mathématiquement à l'aide de preuves. Par ailleurs, une méthode est présentée pour réduire la consommation de ressources logiques lors de l'implémentation. Ainsi, la forme de l'arbre B, c'est-à-dire sa hauteur et le nombre de noeuds par niveaux, est sélectionnée pour être minimale. Par ailleurs, considérant qu'une mémoire dédiée est utilisée pour contenir les noeuds associés à un même niveau de l'arbre, les mémoires (types et configurations) à chacun des niveaux sont sélectionnées afin de réduire la consommation totale de ressources mémoires. L'architecture matérielle présentée est pipelinée afin d'augmenter le débit de recherches. L'architecture matérielle est générée automatiquement à partir d'une description d'une table de comparaison et d'une description architecturale du FPGA ciblé. Sur l'ensemble des scénarios évalués, un débit supérieur à 100 Gb/s est soutenu.

Par ailleurs, les résultat présentés dans cette thèse tendent à démontrer que le surcoût de l'opération LPM implémenté à l'aide de l'approche algorithmique est similaire où inférieur au surcoût lié à l'implémentation de mémoire TCAM dans l'architecture d'un FPGA. Les résultats de cette conclusion sont cependant limités au surcoût en nombre de transistors, et ce en première approximation.

7.2 Limitations

Nous discutons ci-dessous des limitations des solutions proposées pour le routage de paquets IPv6 dans le réseau Internet, puis de la solution conçue pour le contexte de plan de données programmable.

Premièrement, l'efficacité de SHIP n'est pas garantie en dehors de ce contexte, par exemple dans le cas d'un réseau privé où la méthode de regroupement proposée pourrait avoir un

impact nul. En effet, SHIP exploite spécifiquement les caractéristiques des préfixes contenus dans les tables de routage utilisées dans le réseau Internet pour réduire la consommation mémoire. Deuxièment, il n'est pas possible de savoir si les caractéristiques des préfixes IPv6 générés pour caractériser le facteur de mise à l'échelle de SHIP sont réalistes. En effet, bien que la méthode employée pour générer les préfixes synthétiques soit commune à l'ensemble des références auxquels SHIP est comparé, la génération synthétique de préfixes IPv6 est basée sur les caractéristiques des préfixes IPv4 utilisés actuellement dans le réseau Internet. Finalement, bien que le surcoût des mises à jour soit évalué de manière théorique, l'évaluation du surcoût pratique des mises à jour n'a pas été effectuée.

La seconde solution proposée, conçue pour le contexte de plan de données programmables, souffre d'une exploration de l'espace de conception incomplète, ainsi qu'une évaluation partielle de la performance de mise à jour en pratique. Premièrement, l'exploration de l'espace de conception est incomplète pour l'étape de génération de l'architecture matérielle. En effet, lors de cette étape, un algorithme identifie pour chaque niveau de l'arbre B, la ressource mémoire à sélectionner afin de réduire la consommation totale de mémoire. En utilisant uniquement une contrainte de réduction de la consommation totale de mémoire, les ressources de type registres et mémoires distribuées sont dans la majorité des cas préférés comparativement aux autres types de mémoire. Néanmoins, de larges mémoires construites uniquement à partir de registres ou de blocs mémoires distribués supportent une faible fréquence d'horloge. Par conséquent, pour trouver un compromis entre l'efficacité mémoire et la fréquence d'horloge, une taille maximale de mémoire basée sur des registres ou sur des mémoires distribuées sont définies. Toutefois, la taille maximale de mémoire pouvant instancier ces deux types de mémoire a été sélectionnée arbitrairement. Deuxièmement, cette solution a été conçue pour réduire la consommation mémoire, mais ne permet pas de générer une architecture selon des contraintes d'une latence de recherche faible ou des contraintes de débit minimum de recherches à garantir.

7.3 Travaux futurs

Suite aux travaux réalisés dans cette thèse, nous présentons deux avenues de travaux futurs. La première avenue, directement alignée avec la problématique présentée dans cette thèse, consiste à réduire le surcoût d'une solution de comparaison LPM pour un plan de données programmable. L'idée proposée consiste à exploiter une structure de données succincte et d'identifier s'il est possible de concevoir une solution avec un coût en ressource logique faible, indépendamment des données stockées, tout en garantissant un débit de recherches élevée. Dans la littérature, nous ne connaissons pas de travaux abordant cette problématique.

La seconde avenue consiste à répondre à une question plus large : *est-ce que l'architecture des FPGA doit intégrer de la mémoire TCAM ?*. Pour répondre à cette question, il est nécessaire de comprendre *pourquoi*, la mémoire TCAM est utilisée. En réseautique, on peut identifier trois usages : opération de comparaison (ternaire ou LPM), ou intégrée à l'architecture d'un ordonnanceur de paquets. Afin de répondre à la question posée, il est donc nécessaire d'évaluer le coût d'intégration de la mémoire TCAM et de le comparer au coût d'émulation pour chacun de ces trois usages. Cette thèse apporte une réponse pour l'opération LPM mais pas pour les deux autres usages. Considérant le surcoût très élevé de l'opération ternaire sur FPGA, lié à l'utilisation de l'approche mémoire transposée, nous proposons tout d'abord de concevoir une solution adoptant une approche algorithmique. Une telle solution serait par ailleurs d'intérêt en dehors du domaine de réseautique, notamment pour les processeurs implémentés en logique programmable, et pour certaines architectures d'apprentissage profond. Pour un ordonnanceur de paquets, nous proposons d'évaluer dans un premier temps si les architectures d'ordonnanceur de paquets pour un plan de données programmable requièrent *réellement* une mémoire de type TCAM. En effet, ce champ d'études est relativement récent et il n'y a pas encore de consensus quant à l'architecture permettant d'exprimer la majorité des algorithmes d'ordonnancement de paquets, tout en garantissant un débit élevé de traitement. Après avoir proposé une solution pour l'opération de recherche ternaire, et pour l'ordonnancement de paquets, nous pourrions répondre, partiellement à la question de l'intégration de mémoire TCAM dans un FPGA.

RÉFÉRENCES

[1] D. Clark, "The Design Philosophy of the DARPA Internet Protocols", *ACM SIGCOMM Computer Communication Review*, t. 18, nº 4, p. 106-114, 1988.

[2] N. McKeown *et al.*, "OpenFlow : Enabling Innovation in Campus Networks", *ACM SIGCOMM Computer Communication Review*, t. 38, nº 2, p. 69-74, 2008.

[3] D. Kreutz *et al.*, "Software-Defined Networking : A Comprehensive Survey", *Proceedings of the IEEE*, t. 103, nº 1, p. 14-76, 2014.

[4] P. Bosshart *et al.*, "P4 : Programming Protocol-Independent Packet Processors", *ACM SIGCOMM Computer Communication Review*, t. 44, nº 3, p. 87-95, 2014.

[5] P. Bosshart *et al.*, "Forwarding Metamorphosis : Fast Programmable Match-Action Processing in Hardware for SDN", *ACM SIGCOMM Computer Communication Review*, t. 43, nº 4, p. 99-110, 2013.

[6] D. Firestone *et al.*, "Azure Accelerated Networking : SmartNICs in the Public Cloud", in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, p. 51-66.

[7] D. Firestone, "VFP : A Virtual Switch Platform for Host SDN in the Public Cloud", in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA : USENIX Association, mar. 2017, p. 315-328, ISBN : 978-1-931971-37-9. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone.

[8] J. S. da Silva *et al.*, "One for All, All for One : A Heterogeneous Data Plane for Flexible P4 Processing", in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, IEEE, 2018, p. 440-441.

[9] Arista Networks. (2012). 7124FX Application Switch Data Sheet, [En ligne]. Disponible : https://www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf.

[10] S. Trimberger. (2014). The Three Ages of the FPGA, [En ligne]. Disponible : https://www.youtube.com/watch?v=4ntXSyOhlBY&feature=emb_logo.

[11] A. M. Caulfield *et al.*, "A Cloud-Scale Acceleration Architecture", in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, p. 1-13.

[12] J. Fowers *et al.*, “A Configurable Cloud-Scale DNN Processor for Real-Time AI”, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, p. 1-14.

[13] H. Wang *et al.*, “P4FPGA : A Rapid Prototyping Framework for P4”, in *Proceedings of the Symposium on SDN Research*, ACM, 2017, p. 122-135.

[14] Xilinx. (2018). UG1252 : P4-SDNet User Guide, [En ligne]. Disponible : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1252-p4-sdnet.pdf.

[15] Geoff Huston. (2020). BGP in 2019 The BGP Table, [En ligne]. Disponible : {https://labs.ripe.net/Members/wilhelm/so-long-last-8-and-thanks-for-all-the-allocations}.

[16] T. Stimpfling *et al.*, “SHIP : A Scalable High-performance IPv6 Lookup Algorithm that Exploits Prefix Characteristics”, *IEEE/ACM Transactions on Networking*, t. 27, n° 4, p. 1529-1542, 2019.

[17] Next Generation Mobile Networks (NGMN) Alliance, “5G White paper - Version 1.0”, rapp. tech., 2015.

[18] A. M. Abdelhadi *et al.*, “Modular Block-RAM-Based Longest-Prefix Match Ternary Content-Addressable Memories”, in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, p. 243-2437.

[19] W. Jiang, “Scalable Ternary Content Addressable Memory Implementation Using FPGAs”, in *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, IEEE Press, 2013, p. 71-82.

[20] J. S. da Silva *et al.*, “Virtually Infinite Match Tables on Programmable Dataplanes”, in *Submitted to the ACM SIGCOMM Computer Communication Review*, 2020.

[21] T. Luinaud *et al.*, “Bridging the Gap : FPGAs as Programmable Switches”, in *2020 IEEE 21th International Conference on High Performance Switching and Routing (HPSR)*, 2020.

[22] N. Feamster *et al.*, “The Road to SDN”, *Queue*, t. 11, n° 12, p. 20-40, 2013.

[23] S. Shenker. (2011). An Attempt to Motivate and Clarify Software-Defined Networking (SDN), [En ligne]. Disponible : https://www.youtube.com/watch?v=WVs7Pc99S7w.

[24] ——, *The Future of Networking, and the Past of Protocols*, 2011. [En ligne]. Disponible : https://www.youtube.com/watch?v=YHeyuD89n1Y.

[25] H. Song, “Protocol-Oblivious Forwarding : Unleash The Power of SDN Through a Future-Proof Forwarding Plane”, in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, p. 127-132.

[26] Open Networking Foundation (ONF). (2015). OpenFlow Switch Specification - Version 1.5.1, [En ligne]. Disponible : https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[27] M. Casado *et al.*, “Ethane : Taking Control of the Enterprise”, *ACM SIGCOMM computer communication review*, t. 37, n° 4, p. 1-12, 2007.

[28] N. Gude *et al.*, “NOX : Towards an Operating System for Networks”, *ACM SIGCOMM Computer Communication Review*, t. 38, n° 3, p. 105-110, 2008.

[29] T. Koponen *et al.*, “Onix : A Distributed Control Platform for Large-Scale Production Networks.”, in *OSDI*, t. 10, 2010, p. 1-6.

[30] R. Sherwood *et al.*, “Flowvisor : A Network Virtualization Layer”, *OpenFlow Switch Consortium, Tech. Rep*, t. 1, p. 132, 2009.

[31] N. Foster *et al.*, “Frenetic : A Network Programming Language”, *ACM Sigplan Notices*, t. 46, n° 9, p. 279-291, 2011.

[32] M. Abadi *et al.*, “Tensorflow : A System for Large-Scale Machine Learning”, in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, p. 265-283.

[33] N. P. Jouppi *et al.*, “In-Datacenter Performance Analysis of a Tensor Processing Unit”, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, p. 1-12.

[34] The P4 Language Consortium. (2019). P4 16 Language Specification, [En ligne]. Disponible : https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf.

[35] V. Shrivastav, “Fast, Scalable, and Programmable Packet Scheduler in Hardware”, in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, p. 367-379.

[36] A. Sivaraman *et al.*, “Programmable Packet Scheduling at Line Rate”, in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, p. 44-57.

[37] A. G. Alcoz *et al.*, “SP-PIFO : Approximating Push-In First-Out Behaviors using Strict-Priority Queues”, in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA : USENIX Association, fév. 2020. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi20/presentation/alcoz.

[38] “Programmable Calendar Queues for Packet Scheduling”, in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA : USENIX Association, fév. 2020. [En ligne]. Disponible : https://www.usenix.org/conference/nsdi20/presentation/sharma.

[39] E. Peer, “Mapping P4 to SmartNICs”, rapp. tech., 2017.

[40] M. Shahbaz *et al.*, “PISCES : A Programmable, Protocol-Independent Software Switch”, in *Proceedings of the 2016 ACM SIGCOMM Conference*, sér. SIGCOMM 16, Florianopolis, Brazil : Association for Computing Machinery, 2016, p. 525538, ISBN : 9781450341936. DOI : 10.1145/2934872.2934886. [En ligne]. Disponible : https://doi.org/10.1145/2934872.2934886.

[41] J. Santiago da Silva *et al.*, “P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs”, in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2018, p. 147-152.

[42] J. Cabal *et al.*, “Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput”, in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2018, p. 249-258.

[43] L. Kekely *et al.*, “Fast Lookup for Dynamic Packet Filtering in FPGA”, in *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, IEEE, 2014, p. 219-222.

[44] H. T. Dang *et al.*, “Paxos Made Switch-y”, *ACM SIGCOMM Computer Communication Review*, t. 46, n° 2, p. 18-24, 2016.

[45] R. Miao *et al.*, “SilkRoad : Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs”, in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, sér. SIGCOMM 17, Los Angeles, CA, USA : Association for Computing Machinery, 2017, p. 1528, ISBN : 9781450346535. DOI : 10.1145/3098822.3098824. [En ligne]. Disponible : https://doi.org/10.1145/3098822.3098824.

[46] X. Jin *et al.*, “NetCache : Balancing Key-Value Stores with Fast In-Network Caching”, in *Proceedings of the 26th Symposium on Operating Systems Principles*, sér. SOSP 17, Shanghai, China : Association for Computing Machinery, 2017, p. 121136, ISBN : 9781450350853. DOI : 10.1145/3132747.3132764. [En ligne]. Disponible : https://doi.org/10.1145/3132747.3132764.

[47] T. Jepsen *et al.*, “Fast String Searching on PISA”, in *Proceedings of the 2019 ACM Symposium on SDN Research*, sér. SOSR 19, San Jose, CA, USA : Association for Computing Machinery, 2019, p. 2128, ISBN : 9781450367103. DOI : 10.1145/3314148.3314356. [En ligne]. Disponible : https://doi.org/10.1145/3314148.3314356.

[48] V. Sivaraman *et al.*, “Heavy-Hitter Detection Entirely in the Data Plane”, in *Proceedings of the Symposium on SDN Research*, sér. SOSR 17, Santa Clara, CA, USA : Association for Computing Machinery, 2017, p. 164176, ISBN : 9781450349475. DOI : 10.1145/3050220.3063772. [En ligne]. Disponible : https://doi.org/10.1145/3050220.3063772.

[49] N. Katta *et al.*, “HULA : Scalable Load Balancing Using Programmable Data Planes”, in *Proceedings of the Symposium on SDN Research*, sér. SOSR 16, Santa Clara, CA, USA : Association for Computing Machinery, 2016, ISBN : 9781450342117. DOI : 10.1145/2890955.2890968. [En ligne]. Disponible : https://doi.org/10.1145/2890955.2890968.

[50] H. T. Dang *et al.*, “NetPaxos : Consensus at Network Speed”, in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, sér. SOSR 15, Santa Clara, California : Association for Computing Machinery, 2015, ISBN : 9781450334518. DOI : 10.1145/2774993.2774999. [En ligne]. Disponible : https://doi.org/10.1145/2774993.2774999.

[51] J. Hawkinson et T. Bates, *Rfc1930 : Guidelines for Creation, Selection, and Registration of an Autonomous System (as)*, 1996.

[52] G. Malkin *et al.*, “RIP version 2”, STD 56, RFC 2453, November, rapp. tech., 1998.

[53] R Coltun *et al.*, “Rfc 5340 : Ospf for ipv6”, *IETF. July*, t. 24, 2008.

[54] M. P. Clark, *Data Networks, IP and the Internet : Protocols, Design and Operation*. John Wiley & Sons, 2003.

[55] E. W. Dijkstra *et al.*, “A Note on Two problems In Connexion With Graphs”, *Numerische mathematik*, t. 1, n° 1, p. 269-271, 1959.

[56] Y Rekhter *et al.*, “IETF RFC 4271 : A Border Gateway Protocol 4 (BGP-4)”, 2006.

[57] G Trotter, *RFC3222 : Terminology for Forwarding Information Base (FIB) based Router Performance*, 2001.

[58] K. Pagiamtzis et A. Sheikholeslami, “Content-Addressable Memory (CAM) Circuits and Architectures : A Tutorial and Survey”, *IEEE journal of solid-state circuits*, t. 41, n° 3, p. 712-727, 2006.

[59] P. W. Bosshart, *Low Power Ternary Content-Addressable Memory (TCAM)*, US Patent 8,125,810, 2012.

[60] R. Ozdag, “Intel® Ethernet Switch fm6000 Series-Software Defined Networking”, 2012. [En ligne]. Disponible : {https://people.ucsc.edu/~warner/Bufs/ethernet-switch-fm6000-sdn-paper.pdf}.

[61] Lattice Semiconductor. (2002). Application Note AN8071 : Content-Addressable Memory (CAM) Applications for ispXPLD Devices, [En ligne]. Disponible : https://www.latticesemi.com/en/Support/MatureAndDiscontinuedDevices/ispXPLD5000MX.

[62] *UG573 : Xilinx UltraScale Architecture Memory Resources User Guide*, version 1.10, 2019.

[63] S. Richter *et al.*, “A Seven-Dimensional Analysis of Hashing Methods and Its Implications On Query Processing”, *PVLDB*, t. 9, n° 3, p. 96-107, 2015.

[64] T. H. Cormen *et al.*, *Introduction to Algorithms*. MIT press, 2009.

[65] S. S. Skiena, *The Algorithm Design Manual*. Springer Science & Business Media, 1998, t. 1.

[66] R. Pagh et F. F. Rodler, “Cuckoo Hashing”, in *European Symposium on Algorithms*, Springer, 2001, p. 121-133.

[67] B. Fan *et al.*, “Cuckoo Filter : Practically Better Than Bloom”, in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, p. 75-88.

[68] Xilinx. (2019). Exact Match Binary CAM Search IP for SDNet, [En ligne]. Disponible : https://www.xilinx.com/support/documentation/ip_documentation/cam/pg189-cam.pdf.

[69] M. Bando *et al.*, “FlashLook : 100-Gbps Hash-Tuned Route Lookup Architecture”, in *International Conference on High Performance Switching and Routing, 2009.*, 2009, p. 1-8. DOI : 10.1109/HPSR.2009.5307429.

[70] R. Bayer et E. McCreight, “Organization and Maintenance of Large Ordered Indices”, in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, sér. SIGFIDET 70, Houston, Texas : Association for Computing Machinery, 1970, p. 107141, ISBN : 9781450379410. DOI : 10.1145/1734663.1734671. [En ligne]. Disponible : https://doi.org/10.1145/1734663.1734671.

[71] H. Le et V. K. Prasanna, “Scalable Tree-Based Architectures for IPv4/v6 Lookup Using Prefix Partitioning”, *IEEE Transactions on Computers*, t. 61, nº 7, p. 1026-1039, 2012, ISSN : 0018-9340. DOI : 10.1109/TC.2011.130.

[72] Y. K. Chang *et al.*, “LayeredTrees : Most Specific Prefix-Based Pipelined Design for On-Chip IP Address Lookups”, *IEEE Transactions on Computers*, t. 63, nº 12, p. 3039-3052, 2014, ISSN : 0018-9340. DOI : 10.1109/TC.2013.109.

[73] P. Warkhede *et al.*, “Multiway Range Trees : Scalable IP Lookup with Fast Updates”, *Computer Networks*, t. 44, nº 3, p. 289-303, 2004.

[74] M. Zec *et al.*, “DXR : Towards a Billion Routing Lookups per Second in Software”, *ACM SIGCOMM Computer Communication Review*, t. 42, nº 5, p. 29-36, 2012.

[75] R. De La Briandais, “File Searching Using Variable Length Keys”, in *Papers presented at the the March 3-5, 1959, western joint computer conference*, 1959, p. 295-298.

[76] D. E. Knuth, *The Art of Computer Programming*. Pearson Education, 1997, t. 3.

[77] D. R. Morrison, “PATRICIAPractical Algorithm to Retrieve Information Coded in Alphanumeric”, *Journal of the ACM (JACM)*, t. 15, nº 4, p. 514-534, 1968.

[78] M. Degermark *et al.*, “Small Forwarding Tables for Fast Routing Lookups”, *SIGCOMM Comput. Commun. Rev.*, t. 27, nº 4, p. 314, 1997, ISSN : 0146-4833. DOI : 10.1145/263109.263133. [En ligne]. Disponible : https://doi.org/10.1145/263109.263133.

[79] V. Srinivasan et G. Varghese, “Faster IP Lookups Using Controlled Prefix Expansion”, in *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, sér. SIGMETRICS 98/PERFORMANCE 98, Madison, Wisconsin, USA : Association for Computing Machinery, 1998, p. 110, ISBN : 0897919823. DOI : 10.1145/277851.277863. [En ligne]. Disponible : https://doi.org/10.1145/277851.277863.

[80] W. Eatherton *et al.*, “Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates”, *ACM SIGCOMM Computer Communication Review*, t. 34, nº 2, p. 97-122, 2004.

[81] H. Asai et Y. Ohara, “Poptrie : A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup”, *ACM SIGCOMM Computer Communication Review*, t. 45, n° 4, p. 57-70, 2015.

[82] A. Kalia *et al.*, “Raising the Bar for Using GPUs in Software Packet Processing”, in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, p. 409-423.

[83] M. Waldvogel *et al.*, “Scalable High Speed IP Routing Lookups”, in *Proceedings of the ACM SIGCOMM'97 Conference on Applications, technologies, architectures, and protocols for computer communication*, 1997, p. 25-36.

[84] S. Nilsson et G. Karlsson, “IP-address Lookup Using LC-Tries”, *IEEE Journal on selected Areas in Communications*, t. 17, n° 6, p. 1083-1092, 1999.

[85] G. Rétvári *et al.*, “Compressing IP Forwarding Tables : Towards Entropy Bounds and Beyond”, *IEEE/ACM Transactions on Networking*, t. 24, n° 1, p. 149-162, 2016, ISSN : 1558-2566. DOI : 10.1109/TNET.2014.2357051.

[86] E. Papadogiannaki *et al.*, “Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures”, *IEEE/ACM Transactions on Networking*, t. 25, n° 3, p. 1593-1606, 2017.

[87] T. Yang *et al.*, “Constant IP lookup with FIB explosion”, *IEEE/ACM Transactions on Networking*, t. 26, n° 4, p. 1821-1836, 2018.

[88] M. Zec et M. Mikuc, “Pushing The Envelope : Beyond two Billion IP Routing Lookups per Second on Commodity CPUs”, in *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, IEEE, 2017, p. 1-6.

[89] G. J. Jacobson, “Succinct Static Data Structures”, 1988.

[90] L. Rizzo, “Netmap : A Novel Framework for Fast Packet I/O”, in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA : USENIX Association, 2012, p. 101-112, ISBN : 978-931971-93-5.

[91] “Intel DPDK : Data Plane Development Kit”, rapp. tech. [En ligne]. Disponible : http://dpdk.org..

[92] S. Gallenmüller *et al.*, “Comparison of Frameworks for High-Performance Packet IO”, in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015, p. 29-38.

[93] S. Han *et al.*, “PacketShader : a GPU-accelerated Software Router”, *ACM SIGCOMM Computer Communication Review*, t. 40, n° 4, p. 195-206, 2010.

[94] S. Zhou et V. K. Prasanna, "Scalable GPU-Accelerated IPv6 Lookup Using Hierarchical Perfect Hashing", in *2015 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2015, p. 1-6.

[95] Y. Li *et al.*, "GAMT : A Fast and Scalable IP Lookup Engine for GPU-based Software Routers", in *Architectures for Networking and Communications Systems*, IEEE, 2013, p. 1-12.

[96] J. Kim *et al.*, "NBA (network balancing act) a High-Performance Packet Processing Framework for Heterogeneous Processors", in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, p. 1-14.

[97] Xilinx, "PG191 : Longest Prefix Match (LPM) Search IP for SDNet", rapp. tech., 2017. [En ligne]. Disponible : https://www.xilinx.com/support/documentation/ip_documentation/lpm/pg191-lpm.pdf.

[98] M. Bando *et al.*, "FlashTrie : Beyond 100-Gb/s IP Route Lookup Using Hash-Based Prefix-Compressed Trie", *IEEE/ACM Transactions On Networking*, t. 20, nº 4, p. 1262-1275, 2012.

[99] Cisco. (2016). Cisco Visual Networking Index : Global Mobile Data Traffic Forecast Update, 20152020 White Paper, [En ligne]. Disponible : http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[100] H. J. Chao et B. Liu, *High Performance Switches and Routers*. Wiley, 2007, ISBN : 9780470113943.

[101] D. Tong *et al.*, "A Memory Efficient IPv6 Lookup Engine on FPGA", in *2012 International Conference on Reconfigurable Computing and FPGAs*, 2012, p. 1-6. DOI : 10.1109/ReConFig.2012.6416760.

[102] (2017). IPv6 BGP Table Data, [En ligne]. Disponible : http://bgp.potaroo.net/v6/as2.0/index.html.

[103] (2017). RIS Raw Data Set, [En ligne]. Disponible : https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[104] K. Zheng *et al.*, "A TCAM-Based Distributed Parallel IP Lookup Scheme and Performance Analysis", *IEEE/ACM Transactions on Networking*, t. 14, nº 4, p. 863-875, 2006.

[105] M. Bando et H. J. Chao, "FlashTrie : Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps.", in *2010 Proceedings IEEE INFOCOM*, 2010, p. 1-9. DOI : 10.1109/INFCOM.2010.5462142.

[106] H. Song *et al.*, “IPv6 Lookups Using Distributed and Load Balanced Bloom Filters for 100gbps Core Router Line Cards”, in *INFOCOM 2009, IEEE*, IEEE, 2009, p. 2518-2526.

[107] H. Yu et R. Mahapatra, “A Power and Throughput-Efficient Packet Classifier with n Bloom Filters”, *IEEE Transactions on Computers*, t. 60, n° 8, p. 1182-1193, 2011, ISSN : 0018-9340. DOI : 10.1109/TC.2010.213.

[108] H. Lim *et al.*, “On Adding Bloom Filters to Longest Prefix Matching Algorithms”, *IEEE Transactions on Computers*, t. 63, n° 2, p. 411-423, 2014, ISSN : 0018-9340. DOI : 10.1109/TC.2012.193.

[109] J. H. Mun et H. Lim, “New Approach for Efficient IP Address Lookup Using a Bloom Filter in Trie-Based Algorithms”, *IEEE Transactions on Computers*, t. 65, n° 5, p. 1558-1565, 2016, ISSN : 0018-9340. DOI : 10.1109/TC.2015.2444850.

[110] P. Gupta *et al.*, “Routing Lookups in Hardware at Memory Access Speeds”, in *INFOCOM ’98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, t. 3, 1998, 1240-1247 vol.3. DOI : 10.1109/INFCOM.1998.662938.

[111] L. Luo *et al.*, “A Hybrid Hardware Architecture for High-speed IP Lookups and Fast Route Updates”, *IEEE/ACM Transactions on Networking*, t. 22, n° 3, p. 957-969, juin 2014, ISSN : 1063-6692.

[112] (2017). IPv6 Global Unicast Address Assignments, [En ligne]. Disponible : http://www.iana.org/assignments/ipv6-unicast-address-assignments/ipv6-unicast-address-assignments.xhtml.

[113] Z. J. Czech *et al.*, “Perfect Hashing”, *Theoretical Computer Science*, t. 182, n° 1-2, p. 1-143, août 1997, ISSN : 0304-3975.

[114] P. Gupta et N. McKeown, “Classifying Packets with Hierarchical Intelligent Cuttings”, *IEEE Micro*, t. 20, n° 1, p. 34-41, 2000. DOI : 10.1109/40.820051.

[115] A. Basu et G. Narlikar, “Fast Incremental Updates for Pipelined Forwarding Engines”, *IEEE/ACM Transactions on Networking*, t. 13, n° 3, p. 690-703, 2005, ISSN : 1063-6692. DOI : 10.1109/TNET.2005.850216.

[116] M. Wang *et al.*, “Non-Random Generator for IPv6 Tables”, in *12th Annual IEEE Symposium Proceedings on High Performance Interconnects, 2004. Proceedings*, Washington, DC, USA : IEEE Computer Society, 2004, p. 35-40, ISBN : 0-7803-8686-8.

[117] T. Stimpfling *et al.*, “A Low-Latency Memory-Efficient IPv6 Lookup Engine Implemented on FPGA Using High-Level Synthesis”, in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, p. 402-411.

[118] Y. H. E. Yang *et al.*, “High Performance IP Lookup on FPGA with Combined Length-Infix Pipelined Search”, in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, p. 77-80. DOI : 10.1109/FCCM.2011.61.

[119] G. Wang *et al.*, “Performance and Productivity Evaluation of Hybrid-Threading HLS Versus HDLs”, in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, IEEE, 2015, p. 1-7.

[120] E. Homsirikamol et K. Gaj, “Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain ? A Case Study”, in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, IEEE, 2014, p. 1-8.

[121] E. Homsirikamol *et al.*, “C vs. VHDL : Benchmarking CAESAR Candidates Using High-Level Synthesis and Register-Transfer Level Methodologies”, *Directions in Authenticated Ciphers (DIAC)*, 2015.

[122] D. J. Warne *et al.*, “Comparison of High Level FPGA Hardware Design for Solving Tri-Diagonal Linear Systems”, *Procedia Computer Science*, t. 29, p. 95-101, 2014.

[123] R. Nane *et al.*, “A Survey and Evaluation of FPGA High-Level Synthesis Tools”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, t. 35, n° 10, p. 1591-1604, 2016, ISSN : 0278-0070. DOI : 10.1109/TCAD.2015.2513673.

[124] Xilinx, “DS890 : Ultrascale Overview”, rapp. tech., 2019. [En ligne]. Disponible : https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[125] Altera, *External Memory Interface Handbook Volume 1 : Altera Memory Solution Overview, Design Flow, and General Information*, 2016. [En ligne]. Disponible : https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/external-memory/emi.pdf.

[126] R. Shu *et al.*, “Direct Universal Access : Making Data Center Resources Available to {FPGA}”, in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, p. 127-140.

[127] P. Benácek *et al.*, “P4-to-vhdl : Automatic Generation of 100 Gbps Packet Parsers”, in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, p. 148-155.

[128] 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*, t. 22, nº 10, p. 2067-2080, 2013.

[129] T. P. A. W. Group. (2019). P4Runtime Specification, [En ligne]. Disponible : https://p4.org/p4runtime/spec/v1.1.0/P4Runtime-Spec.pdf.

[130] M. Reitblatt *et al.*, “Abstractions for Network Update”, *ACM SIGCOMM Computer Communication Review*, t. 42, nº 4, p. 323-334, 2012.

[131] Xilinx. (2017). Ternary Content Addressable Memory (TCAM) Search IP for SDNet, [En ligne]. Disponible : https://www.xilinx.com/support/documentation/ip_documentation/tcam/pg190-tcam.pdf.

[132] A. M. S. Abdelhadi et G. G. F. Lemieux, “Modular SRAM-Based Binary Content-Addressable Memories”, in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, p. 207-214. DOI : 10.1109/FCCM.2015.69.

[133] T. Shen *et al.*, “High-Performance IPv6 Lookup with Real-Time Updates Using Hierarchical-Balanced Search Tree”, in *2018 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2018, p. 1-7.

[134] Y.-H. E. Yang et V. K. Prasanna, “High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA”, in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2010, p. 83-92.

[135] V. Demianiuk et K. Kogan, “How to Deal with Range-Based Packet Classifiers”, in *Proceedings of the 2019 ACM Symposium on SDN Research*, ACM, 2019, p. 29-35.