UNIVERSITÉ DE MONTRÉAL

ACCECUTS : UN ALGORITHME DE CLASSIFICATION DE PAQUETS CONÇU POUR TRAITER LES NOUVEAUX PARADIGMES DES RÉSEAUX DÉFINIS PAR LOGICIEL

THIBAUT STIMPFLING
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ACCECUTS : UN ALGORITHME DE CLASSIFICATION DE PAQUETS CONÇU POUR TRAITER LES NOUVEAUX PARADIGMES DES RÉSEAUX DÉFINIS PAR LOGICIEL

présenté par : STIMPFLING Thibaut
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. FRIGON Jean-François, Ph.D., président
M. SAVARIA Yvon, Ph.D., membre et directeur de recherche
M. CHERKAOUI Omar, Ph.D., membre et codirecteur de recherche
Mme SANSÒ Brunilde, Ph.D., membre

Au partage de la connaissance ...

REMERCIEMENTS

Je tiens à remercier l'ensemble des personnes qui m'ont permis de mener à bien mes travaux de maîtrise, que cela soit de manière directe ou non. Je souhaite remercier tout d'abord mon directeur de recherche, le professeur Yvon Savaria, pour m'avoir écouté, conseillé, et de m'avoir fait confiance. J'aimerais aussi remercier mon co-directeur de recherche, le professeur Omar Cherkaoui, pour m'avoir aidé à bien des égards, et notamment afin de faire murir ma réflexion sur le travail que je réalisais.

J'aimerais adresser mes remerciements à Normand Bélanger, qui a toujours été là pour m'aider, et me donner d'innombrables conseils, et avec lequel j'ai eu de très nombreuses discussions, techniques ou non, mais au combien intéressantes !

Je souhaiterais remercier les professeurs Brunilde Sansò et Jean-François Frigon pour leur participation au jury en charge de l'évaluation.

J'aimerais remercier les personnes qui m'ont permis de travailler avec du matériel informatique fonctionnel, et qui ont corrigé de nombreux bugs ; Réjean Lepage ainsi que Jean Bouchard. De même je tiens à remercier les différentes personnes qui m'ont aidé au niveau administratif, Marie-Annick Laplante, Louise Clément, Nathalie Lévesque et Ghyslaine Ethier-Carrier.

J'ai une pensée pour les étudiantEs avec lesquel(lE)s j'ai partagé le local de recherche, Mona, David, Meng, Federico, et avec lesquels j'ai partagé la peine et la joie de tout étudiantE aux cycles supérieurs. Une pensée toute particulière va au comité Poly Party, et notamment à ceux et celles qui doutaient que je termine un jour ma maîtrise ! Je remercie aussi mes amiEs rencontréEs à Polytechnique et avec lesquels j'ai passé tant de bons moments.

Je tiens tout particulièrement à remercier mes amiEs et camarades de l'UQAM, avec qui j'ai passé tant de temps (et de moments inoubliables), qui m'ont ouvert à d'autres enjeux, et au travers desquellEs j'ai beaucoup appris.

Je remercie mes collocatairEs de les avoir rencontréEs !

Enfin je tiens à remercier mes parents, et ma soeur qui m'ont soutenus, et qui n'ont jamais cesser de m'encourager.

RÉSUMÉ

La classification de paquets est une étape cruciale et préliminaire à n'importe quel traitement au sein des routeurs et commutateur réseaux ("switch"). De nombreuses contributions sont présentes dans la littérature, que cela soit au niveau purement algorithmique, ou ayant mené à une implémentation. Néanmoins, le contexte étudié ne correspond pas au virage du Software Defined Networking (SDN, ou réseau défini par logiciel) pris dans le domaine de la réseautique. Or, la flexibilité introduite par SDN modifie profondément le paysage de la classification de paquets. Ainsi, les algorithmes doivent à présent supporter un très grand nombre de règles complexes. Dans le cadre de ce travail, on s'intéresse aux algorithmes de classification de paquets dans le contexte de SDN. Le but est d'accélérer l'étape de classification de paquets et de proposer un algorithme de classification, capable d'offrir des performances de premier plan dans le contexte de SDN, mais aussi, offrant des performances acceptables dans un contexte classique. A cet égard, une évaluation d'EffiCuts, un des algorithmes offrant la meilleure performance, est effectuée dans un contexte de SDN. Trois optimisations sont proposées ; le*Adaptive grouping factor* qui permet d'adapter l'algorithme aux caractéristiques de la table de classification utilisée, le *Leaf size modulation*, visant à déterminer la taille optimale d'une feuille dans le contexte de SDN et enfin, une modification de l'heuristique utilisée pour déterminer le nombre de découpe à effectuer au niveau de chacun des nœuds, permettant de réaliser un nombre de découpes réduit. Ces trois optimisations permettent une augmentation des performances substantielle par rapport à EffiCuts. Néanmoins, de nombreuses données non pertinentes demeurent lues. Ce problème, inhérent à certains algorithmes utilisant des arbres de décision (plus précisément HiCuts et ses descendants), tend à ajouter un nombre significatif d'accès mémoire superflus. Ainsi, un nouvel algorithme, est proposé. Cet algorithme nommé AcceCuts, s'attaque à l'ensemble des problèmes identifiés. Ce dernier reprend les optimisations précédentes, et ajoute une étape de prétraitement au niveau de la feuille, permettant d'éliminer les règles non pertinentes. Une modification majeure de la structure des feuilles, ainsi que de la technique du parcours de l'arbre de décision est donc présentée. AcceCuts augmente les performances de manière substantielle. Ainsi, AcceCuts diminue le nombre d'accès mémoire moyen par un facteur 3, tout en diminuant la taille de la structure de donnée générée de 45% par rapport à EffiCuts. AcceCuts permet donc d'améliorer l'efficacité de la classification dans le contexte de SDN, tout en performant mieux que les algorithmes présents dans la littérature, dans un contexte classique.

ABSTRACT

Packet Classification remains a hot research topic, as it is a fundamental function in telecommunication networks, which are now facing new challenges. Many contributions have been made in literature, focusing either on designing algorithms, or implementing them on hardware. Nevertheless, the work done is tightly coupled to a outdated context, as Software Defined Networking (SDN) is now the main topic in networking. SDN introduces a high degree of flexibility, either in processing or parsing, which highly impact on the packet classification performance: algorithms have now to handle a very large number of complex rules.

We focus this work on packet classification algorithms in SDN context. We aim to accelerate packet classification, and create a new algorithm designed to offer state of the art performance in SDN context, while performing in a classical context.

For this purpose, an evaluation of EffiCuts, a state of the art algorithm - in a classical context -, is performed in SDN context. Based on this analysis, three optimizations are proposed: "Adaptive Grouping Factor", in order to adapt the algorithm behavior to dataset characteristic, "Leaf size modulation", allowing to choose the most relevant leaf size, and finally adopting a new heuristic to compute the number of cuts at each node, in order to determine an optimal number of cuts. Those three optimizations improve drastically the performance over EffiCuts. Nevertheless, some issues are still not addressed, as many irrelevant data are still read, incurring multiples useless memory accesses. This inherent problem to decision tree based algorithms (HiCuts related algorithms) tends to add unnecessary memory accesses for each tree considered. Therefore, in SDN context, this becomes more critical as many clock cycles are wasted.

Consequently, a new algorithm, AcceCuts, addressing those issues, is proposed. This algorithm reuses previous optimizations, and adds a new pre-processing step before leaf processing, in order to remove irrelevant memory accesses. A major modification of leaf data structure is done, including a new method for leaf traversal. AcceCuts increases the performance over other algorithms, cuts down the number of memory accesses on average by a factor of 3, while decreasing the size of the data structure generated by 45 % on average, over EffiCuts. AcceCuts allows to perform packet classification in SDN context while reaching state of the art performance, in both SDN and classical context.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

QoS	Quality of Service
IaaS	Infrastructure as a Service
RFC	Recursive Flow Classification
SDN	Software Defined Networking
SDRAM	Synchronous Dynamic Random Access Memory
TCAM	Ternary Content Adressable Memory

CHAPITRE 1

INTRODUCTION

Internet, acronyme de "*INTERnational NETwork*", ou réseau international, est un réseau téléinformatique international, issue de l'interconnexion d'équipements informatiques, utilisant un protocole commun d'échange de données. Ces équipements interconnectés peuvent êtres des ordinateurs, tels que ceux utilisés par des particuliers, des serveurs, c'est-à-dire des dispositifs informatiques permettant d'offrir des services, ou n'importe quel matériel capable de communiquer selon ce même protocole commun. Néanmoins, cette collection de périphériques n'est pas directement reliée à un unique réseau. Internet est constitué d'un maillage de réseaux interconnectés, adoptant un schéma pyramidale, ou des "petits" réseaux, sont interconnectés à des réseaux de tailles moyennes qui eux même sont reliés à de plus gros réseaux.

De tels réseaux de télécommunication tirent leur puissance d'une interconnexion avec une multitude d'autres réseaux, permettant d'accéder à n'importe quel site Internet, sans égard à l'opérateur de télécommunication choisi par l'abonné. Ainsi, les équipements réseaux permettant de faire le lien entre plusieurs réseaux apparaissent donc comme la pierre angulaire de ce maillage. Les équipements réseaux utilisés sont nombreux, mais, dans le cadre de ce mémoire, on se concentrera majoritairement sur les routeurs et commutateur réseaux, étant couramment utilisés pour interconnecter plusieurs sous-réseaux entre eux. Ces derniers se chargent d'identifier, de traiter et diriger l'ensemble des données véhiculées au travers des réseaux. Les routeurs et commutateurs réseaux se décomposent suivant un plan de contrôle et, d'autre part, un plan de données. Le plan de contrôle gère les informations de routage (i.e. il définit comment diriger les paquets), les protocoles de routage et peut aussi définir un ensemble de paquets à être ignorés ou traités en respectant une certaine qualité de service (QoS). Le plan de données concerne la partie de l'architecture du routeur ou commutateur qui décide comment traiter les paquets entrant au niveau d'une interface. De manière générale, cela réfère à une structure donnée (souvent une table) dans laquelle l'équipement réseau effectue une recherche qui lui permet de déterminer, entre autres, la destination du paquet entrant, ainsi que les informations nécessaires pour acheminer le paquet au travers du routeur.

Les équipements de type routeur et commutateur, traditionnellement, transmettent les paquets sur les interfaces de sortie, en traitant chacun des paquets de manière identique. Néanmoins, en raison de l'émergence de nouvelles applications, un besoin de fournir différents niveaux de qualité de service est apparu. Or, Internet fournit un service de type "meilleur

effort" et ne peut offrir de garantie d'aucune sorte. Ainsi, afin de répondre à ces nouveaux besoins, les routeurs ont intégrés de nouveaux mécanismes, tels que la réservation de ressources, le contrôle d'admission, un ordonnancement équitable entre les différentes queues de sorties, une gestion des queues par trafic . Néanmoins, ces nouveaux mécanismes impactent grandement l'étape d'identification, puisque le trafic entrant doit pouvoir être différencié avec une fine granularité. Cette étape préliminaire (l'identification des données entrantes en vue d'appliquer un traitement spécifique) est appelée classification de paquets. En effet, les données échangées au travers des réseaux se composent de paquets, contenant, d'une part, une entête, et d'autre part de la donnée brute . L'entête contient plusieurs champs permettant de caractériser les données, ainsi que de permettre leur cheminement dans le réseau . Ainsi, l'identification des données circulant sur le réseau passe par l'identification des entêtes associées à chacun des paquets.

La classification de paquets vise à comparer l'entête de chacun des paquets à un ensemble de filtres ou règles contenues dans une ou plusieurs tables de classification. Une règle est constituée d'un ensemble de contraintes portant sur plusieurs champs de l'entête du paquet. Une contrainte est associée à chacun des k champs de l'entête du paquet considéré, F_1 à F_k. De plus, l'ensemble des contraintes doivent être respectées pour que la règle soit sélectionnée.

On distingue quatre types de contraintes qui peuvent résulter d'un appariement : exact, par intervalle, par préfixe ou par masque. Un paquet P est dit associé à une règle T_i si et seulement si l'ensemble des contraintes F_1 à F_k sont vérifiées par chacun des champs respectifs du paquet P. À chacune des règles est associée une action, notée $Action_i$, précisant comment le paquet P doit être traité. Traditionnellement, des 5-uplets, c'est-à-dire composé de cinq champs, sont utilisés : adresse IP Source, adresse IP Destination, Protocole de transport, Port Source et Port de Destination. Les différents types de comparaisons utilisées pour chacun des champs, ainsi que leur taille, sont détaillés dans le tableau 1.1.

Tableau 1.1 Champs utilisés pour la classification à 5 champs

Champ	Taille du champ (bit)	Taille de la règle (bit)	Comparaison
`IP Source`	32	38	Adresse avec préfixe
`IP Destination`	32	38	Adresse avec préfixe
`Protocole de Transport`	8	16	Valeur avec Masque
`Port Source`	16	32	Intervalle quelconque
`Port Destination`	16	32	Intervalle quelconque

Le tableau 1.2, présenté à titre d'exemple, contient plusieurs règles utilisées pour classifier des paquets. Un tel tableau est désigné comme étant une table de classification. Le signe

* spécifie que le champ peut contenir n'importe quelle valeur, comprise entre les valeurs extrêmes associées à ce champ. Dans l'exemple ci-dessous, on peut constater qu'à une action peuvent correspondre plusieurs règles. Il n'y a donc pas une unique règle par action. Par ailleurs, les valeurs présentées correspondantes à la traduction sous forme d'intervalles des différentes comparaisons à effectuer. Par ailleurs, les règles peuvent contenir des contraintes proches et similaires, de telle sorte qu'un paquet peut générer une comparaison positive avec plusieurs règles. Dans un tel cas, dépendamment du contexte, soit l'ensemble des règles associées au paquet sont retenues, ou soit seule la règle avec la priorité la plus élevée est conservée.

Tableau 1.2 Exemple de table de classification utilisant 3 champs

Règle	Champ 1	Champ 2	Champ 3	Action
R_0	[2 ; 3]	[14 ; 15]	TCP	$Action_0$
R_1	[11 ; 11]	[14 ; 15]	UDP	$Action_1$
R_2	*	[6 ; 7]	TCP	$Action_0$
R_3	[0 ; 1]	[0 ; 1]	TCP	$Action_3$
R_4	[8 ; 9]	*	TCP	$Action_4$
R_5	[0 ; 1]	*	UDP	$Action_3$
R_6	[0 ; 1]	[11 ; 11]	UDP	$Action_6$
R_7	[0 ; 1]	[0 ; 1]	UDP	$Action_7$
R_8	*	[13 ; 13]	TCP	$Action_8$
R_9	*	[0 ; 1]	TCP	$Action_9$
R_10	*	*	*	$Action_{10}$

Considérons maintenant un paquet avec les caractéristiques présentées dans le tableau 1.3. L'étape de classification consiste à trouver quelle(s) règle(s) est associée au paquet entrant. Dans le cas de la table 1.2, le paquet considéré correspond à une seule règle, R_10. En effet, pour plusieurs règles, le paquet satisfait à certaines contraintes, mais ne les respecte pas intégralement. Au contraire, la règle R_10, est associée à n'importe quel paquet entrant, en raison de l'utilisation de contraintes de type "générique", ne spécifiant aucune valeur. Le paquet entrant dans notre cas sera ensuite traitée selon le traitement prévu par l'$Action_{10}$.

Tableau 1.3 Exemple d'un entête de paquet

Champ 1	Champ 2	Champ 3
4	10	TCP

1.1 Métriques

Dans ce contexte, différentes techniques de classification de paquets ont été présentés dans la littérature (Voir Section 2). Au-delà des approches présentées, il est important de pouvoir les comparer. À cet égard, afin d'évaluer et de comparer les solutions présentes dans la littérature, plusieurs métriques doivent être considérées :

- **Vitesse de recherche**
  Le débit offert pas les liens dépassant allègrement 40 Gbps, l'engin de classification doit être capable de soutenir un tel débit. Afin de garantir un tel débit, même dans le pire des cas, le raisonnement suivant sera effectué en considérant des paquets de taille minimale. Faisant l'hypothèse de paquet de taille minimale de 40 octets (taille d'un paquet utilisant le protocole TCP, portant un message de type "accusé de réception TCP" mais pas de donnée utile), cela requiert un traitement de 125 millions de paquets par secondes, soit un paquet traité toute les 8 ns. Le nombre de paquets devant être traité est par conséquent très élevé, et on comprend toute l'importance de concevoir des alogirthmes ayant une vitesse de fonctionnement extrêmement élevée.
- **Occupation mémoire**
  Une occupation mémoire faible est importante dans la mesure où cela permet de minimiser la structure matérielle de stockage, ou pour un même espace, de contenir de plus grosses tables de classification. À cet égard, mesurer le facteur de réplication - défini comme étant le rapport du nombre de règles contenues dans la structure de données par rapport au nombre de règles contenues dans la table de classification - permet d'identifier l'origine de la consommation mémoire. En effet, cela permet de déterminer si la consommation de mémoire provient du stockage des règles ou de la structure de données menant aux règles.
- **Extensibilité avec le nombe de règles**
  D'un part, la taille des tables de classification ne cesse d'augmenter et, d'autre part, dépendamment du contexte d'utilisation des équipements de télécommunication, les tables de classification peuvent être composées de quelques centaines d'entrées, voire plusieurs milliers, ou une centaine de milliers. Par conséquent, il est important d'évaluer le critère d'extensibilité afin de comprendre dans quels contextes l'algorithme peut s'appliquer. On comprend qu'il est souhaitable que la complexité de l'algorithme n'explose pas avec le nombre de règles considérées.
- **Extensibilité du nombre de champs considérés**
  L'émergence de nouveaux services et applications, que cela soit pour les particuliers ou au niveau des entreprises, nécessite une différenciation fine des trafics et, par conséquent,

la prise en compte d'un nombre supplémentaire de champs dans les règles des tables de classification. Ainsi, il sera pertinent d'évaluer la capacité des solutions à traiter un nombre de champs accru.

- **Flexibilité en termes de type de règles supportées**
  Les règles peuvent spécifier des contraintes de type exact, par intervalle, par préfixe ou par masque. Or, certains algorithmes, ne peuvent pas supporter l'ensemble des contrainte, ou parfois au prix d'un surcoût. Il est donc important d'évaluer les conséquences de la prise en charge de chacun des types de règles au niveau de l'algorithme.
- **Capacité de réaliser des mises à jour** Dans un contexte, ou de nouvelles machines sont constamment ajoutées, retirées d'un parc informatique, ou le déploiement de nouveaux services influe directement sur le traffic évoluant, il est nécessaire que les méthodes de classification soient en mesure de supporter la mise des jours des tables de classification sur lesquelles elles s'appuient. Dépendemment du contexte d'utilisation, la vitesse d'exécution des mises à jour sera un critère prédominant.
- **Flexibilité d'implémentation** L'ensemble des méthodes proposées visent à être déployé dans des réseaux et donc à traiter un nombre très élevé de paquets par seconde. Dans un tel contexte, l'algorithme doit être implémentable en logiciel. L'implémentation logicielle permet de réaliser divers tests, mais de nombreuses limitations matérielles empêchent d'atteindre un degré élevé de performance. Par conséquent, les différentes méthodes de classification doivent être implémentable sur une plateforme matériel, afin de répondre aux différents critères de performance actuels.

De nombreux travaux ont été réalisés afin de concevoir des algorithmes optimisés pour les métriques présentées mais restreints aux 5-uplets. Par conséquent, la contrainte "extensibilité du nombre de champs considérés" a été rarement considérée dans la littérature. De plus, comme on le verra dans la section 2, l'ensemble des métriques ne sont pas évaluées de manière systématique, mais certaines métriques suffisent à donner une idée de la performance des algorithmes. Enfinm, la liste établie ci-dessus n'est pas exhaustive, et dans certains contextes, on pourrait considérer des métriques plus pertinentes.

1.2 Compromis entre espace de stockage et rapidité

Le processus de classification se fait en éliminant ou sélectionnant, de manière itérative, les règles qui ne correspondent pas à l'entête du paquet entrant. Intuitivement, on comprend qu'il s'agit d'une opération de localisation complexe et pouvant être longue lorsque l'on considère un espace de grande taille. Par ailleurs, il faut penser aussi que la phase de recherche ne peut se limiter à une simple recherche linéaire dont la complexité est en $O(m)$ pour m règles

considérées et, par conséquent, que la recherche doit s'appuyer sur une structure accélérant les prises de décision dans cet espace multidimensionnel. Néanmoins, on comprend aussi que l'ajout d'information tend à augmenter la taille des données manipulées (règles comprises), et qu'un des principaux compromis, que les algorithmes tenteront d'optimiser, concerne la taille de la structure versus la vitesse de convergence. Afin de justifier ces propos, abordons le problème d'un point de vue théorique, et plus particulièrement mathématique.

En adoptant un point de vue géométrique, les champs sont vus comme des dimensions d'un espace multi-dimensionnel, et le paquet comme un point dont l'entête représente les coordonnées de ce dernier dans l'espace. Ainsi dans l'ensemble de ces travaux, un champ de l'entête d'un paquet sera aussi appelé dimension.

Si l'on considère m règles, basées sur d dimensions (nombre de champs employés par les règles), alors, dans (Overmars et van der Stappen, 1994; Gupta et McKeown, 2001) pour un espace de stockage limité à *O(m)*, le processus de recherche de règle peut prendre un temps $O(\log(m)^d - 1)$ et, si l'on souhaite que la recherche s'effectue en un temps $O(\log m)$, alors l'espace de stockage peut être de l'ordre de $O(m^d)$. A la lumière de ces modélisations, on constate qu'il y a un compromis important entre temps de recherche et consommation mémoire, et qu'un tel compromis risque d'être limitant dans des situations où l'on cherche une vitesse d'exécution élevée, tout en ayant une contrainte de faible ocucpation mémoire.

Par ailleurs, les résultats théoriques présentés sont valables seulement lorsqu'il n'y a aucun chevauchement entre les différents règles. Or, d'une part les tables de classification réelles (Yaxuan *et al.*, 2009) permettent de constater que de nombreux chevauchements sont présents et les chevauchements entre règles sont fréquents, mais dans le même temps, la complexité globale est moindre que celle prévue d'un point de vue théorique. De manière générale, on peut constater que les algorithmes utilisée pour la classification tendent à privilégier soit une faible consommation mémoire, soit une classification rapide. Néanmoins, ces derniers sont bien souvent construits et orientés selon un seul critère, ce qui peut limiter leur utilisation à certains contextes particuliers. Au-delà du compromis soulevés ici, il est important de comprendre quel est le contexte actuel dans lequel ces différents algorithmes seront utilisés, afin de ne retenir que les algorithmes pertinents.

1.3 Évolution du contexte de la classification

L'évolution de l'utilisation des ressources informatiques, que cela soit au niveau des centres de traitement de données ou au niveau des points d'interconnexion de réseaux des opérateurs de télécommunication, a eu un impact majeur au niveau de la classification de paquets. Tout d'abord, de nombreux nouveaux services offerts aux particuliers ont fait exploser la consom-

mation de bande passante : *streaming* vidéo, téléconférence, partage de données, etc., ce qui a un impact direct sur les fournisseurs d'accès à Internet. En effet, une explosion de la consommation de la bande passante nécessite une augmentation du débit des liens. Or, le déploiement de nouveaux liens, nécessite de lourds investissements. Néanmoins, en réseautique, le taux d'utilisation moyen des liens est de l'ordre de 50%. Ce faible taux d'utilisation s'explique par une alternance de période de faible utilisation, contrastant avec des périodes de "rafales" intensives de données. Par conséquent, plutôt que d'augmenter massivement la capacité des liens, la tendance actuelle est d'augmenter le taux d'utilisation de ces derniers. Néanmoins, cela nécessite la capacité d'identifier avec une très fine granularité l'ensemble du trafic transitant sur le réseau, et d'avoir une vue unifiée des réseaux, afin de pouvoir optimiser le taux d'utilisation des liens.

D'autre part, les liens interconnexions voient leur capacité augmenter. Là où il était courant de voir des liens 0C-192 (10 Gbps), au niveau des dorsales, il est devenu courant de voir des liens 0C-768 (40 Gbs). De même, l'Ethernet 100 Gb/s a été standardisé et le 400 Gb/s est en voie de l'être. La tendance est donc à une utilisation de liens à très haut débit, tout en maximisant leur utilisation, afin d'éviter de déployer des réseaux surdimensionnés. En faisant l'hypothèse d'un débit de 100 Gb/s, un engin de classification devrait capable de traiter plus de 300 millions de paquets par secondes, ce qui demande inévitablement de développer des algorithmes répondants à de telles spécifications. Aussi, certaines compagnies (telles que le leader de la recherche en ligne, Google et des réseaux sociaux, Facebook) rêvent de pouvoir utiliser de l'Ethernet Terabit pour leurs centres de traitement de données (Stephen, 2010) ; on comprend donc que la classification de paquets demeure un sujet de recherche d'actualité.

Parallèlement, dans les centres de donnée ou "Data Center", la virtualisation a été une révolution majeure dans la gestion des ressources matérielles. Cette technique a en effet permis de s'affranchir de la couche matérielle et de mieux gérer les ressources matérielles en partageant celles-ci entre plusieurs utilisateurs et applications. La virtualisation a donc constitué le premier tournant dans la gestion des ressources de calcul d'un centre de données. Par ailleurs, le virage de l'infonuagique ("cloud computing") a été pris par de nombreuses compagnies, en utilisant des applications et des services en ligne. Par conséquent, d'importants efforts ont été accomplis dans la gestion des ressources matérielles au niveau des centres de traitement de données. À titre d'exemple, on peut citer l'entreprise Amazon, qui a été la première entreprise à offrir la possibilité à d'autres compagnies de louer de la puissance de calcul, ou de stockage, de manière flexible, au travers de son offre Amazon Web Services ("Elastic Compute Cloud", "Simple Storage Service"). Ainsi, l'ensemble des ressources matérielles sous-jacentes (sur lesquelles s'exécutent les opérations de stockage ou de calcul) est complètement abstrait et unifié. L'infrastructure matérielle est dorénavant fournie sous forme

de service. Afin de rendre accessible la révolution de l'infonuagique à l'ensemble des acteurs, un projet de logiciel à sources-ouvertes, nommé Open Stack a vu le jour dans le but de fournir l'infrastructure comme un service. Ce dernier est une compilation de modules logiciels, pensés pour un usage dans les nuages et optimisés pour être massivement extensibles. Il peut être considéré comme un système d'exploitation pour l'informatique dans les nuages, capable de contrôler un nombre très important de ressources de calcul, de stockage et de réseau pour un centre de traitement de données.

Les ressources matérielles sont maintenant gérables plus facilement et redimensionnables à la volée, mais la flexibilité introduite a de lourdes conséquences au niveau réseautique. En effet, d'un point de vue réseautique, il est nécessaire d'optimiser la gestion du réseau afin de ne pas surcharger les liens et de router les données de manière intelligente en fonction du type de trafic à transporter et en fonction du patron de trafic généré par une application. Cette gestion du trafic devant être effectuée au niveau global (i.e. pour l'ensemble des ressources utilisées, physiques ou virtuelles) et par rapport à chaque type de trafic (comprendre chaque application), il est inconcevable que cela soit effectué de manière manuelle. Une gestion automatisée, basée sur la connaissance de l'utilisation des différentes ressources, est nécessaire.

L'infonuagique a vu l'émergence du modèle "Infrastructure as a Service" (IaaS), dans lequelle, les entreprises se concentrent sur la couche applicative et passent par un fournisseur qui se charge de gérer l'infrastructure matérielle. De manière analogue, Le stockage et les réseaux, le monde de la réseautique est en voie de prendre le virage du "Software Defined Networking" (SDN). Le SDN est une approche visant à abstraire le plan de données du plan de contrôle. Ce découplage permet ainsi de faire abstraction des fonctionnalités de bas niveau et de se concentrer sur une gestion à haut niveau des réseaux. Grace à un tel fonctionnement, les réseaux sont gérés de manière plus intelligente, ce qui permet par exemple, en optimisant le transit des paquets selon le taux de congestion dans les liens, d'atteindre un taux d'utilisation des liens de l'ordre de 90%. Le SDN permet d'avoir une vue unifiée des réseaux et, par là même, de se concentrer sur la fonctionnalité à haut niveau (couche applicative des réseaux), ce qui est d'autant plus crucial dans le contexte de l'infonuagique. En effet, les grappes de serveurs sur lesquels les opérations de calculs sont effectuées travaillent conjointement afin d'offrir un haut degré de performance, et sont amenés à échanger de grande quantités information, qui doivent transiter par le réseau. Dans un tel contexte, le trafic réseau doit être traité et routé de manière intelligente, afin de tenir compte des pics de trafics de certaines applications, lorsque des résultats partiels de calculs distribués sont combinés et transmis vers le même serveur. De telles opérations exécutées sur de grosses grappes de serveurs génèrent d'importants transferts sur le réseau, et par conséquent la gestion du réseau doit être faite en

adéquation avec les applications exécutées. De plus dans un contexte de IaaS, où les ressources sont allouées de manière plus dynamique, de nombreuses instanciations de machines virtuelles peuvent être effectuées (création ou suppression), ce qui entraine d'important transfert de données sur le réseau. Par ailleurs, la gestion de l'ensemble des machines virtuelles nécessite de configurer le réseau de manière adéquate (sécurité, isolement, QoS, trafic à identifier). Ainsi, la couche réseautique doit être reconfigurée en adéquation avec la modification de l'utilisation des ressources. Par conséquent, le développement du IaaS a un impact direct et majeur sur la gestion du réseau, qui doit se faire en ayant conscience des applications exécutées. SDN et IaaS devront fonctionner de pair, afin de gérer l'ensemble des ressources matérielles de manière unifiée .

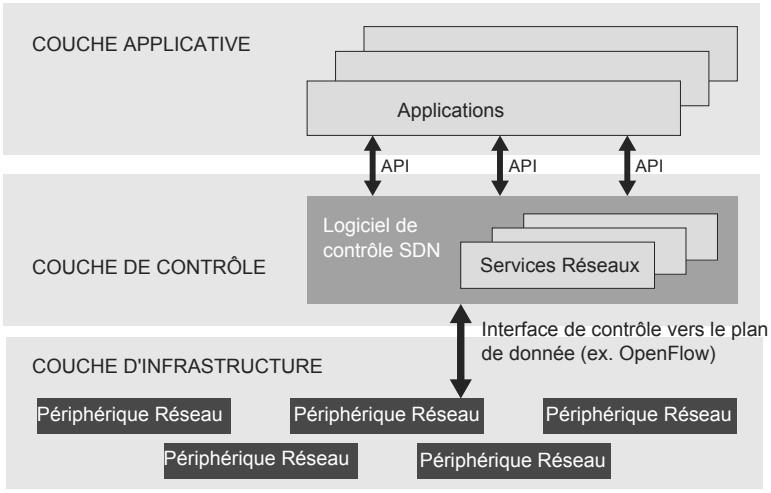


Figure 1.1 Composants SDN. Figure traduite de (Open Networking Foundation, April 13, 2012)

Comme mentionné précédemment, l'avantage du SDN pour les usagers, est de pouvoir se concentrer sur le développement d'applications haut niveau pour la réseautique. Ces dernières interagissent avec un contrôleur, qui fait le lien entre les applications, et la couche matérielle. Le contrôleur utilise ensuite un protocole préétabli pour communiquer avec le matériel et transmettre les différents traitements à effectuer par ce dernier. Une illustration provenant d'un document technique *Software-Defined Networking : The New Norm for Networks* de la fondation *Open Networking Foundation*, ayant conduit à la rédaction de la spécification d'un protocole présenté ci-dessous est illustré dans la figure 1.1.

Un des protocoles utilisés dans le contexte de SDN, pour communiquer entre le contrôleur et routeurs, est OpenFlow. Ce protocole génère d'importantes contraintes au niveau de la classification de paquets. OpenFlow implémente le traitement d'un paquet comme une succession de recherches et d'actions à exécuter, et ce, de façon entièrement programmable, de telle sorte

que le traitement apporté à un paquet est entièrement personnalisable et flexible. Néanmoins, dans le cadre du travail réalisé ici, on se concentre davantage sur les conséquences au niveau de la classification de paquet. Ainsi, davantage de champs peuvent être pris en compte pour la classification de paquets dans un tel contexte ; 12 champs et plus pour les dernières versions d'OpenFlow (v1.0.0 et plus récentes). A cet égard, les différents champs utilisés dans un tel contexte sont présentés dans le tableau 1.2.

La prise en compte d'un nombre élevé de champs a un impact direct sur la taille des règles et donc l'utilisation mémoire, mais par la même occasion, risque de diminuer de manière drastique la performance des algorithmes existants, en raison de l'augmentation du nombre d'accès mémoire généré.

De plus, le nombre de règles contenues dans une table de classification tend à augmenter à des niveaux élevés, pouvant atteindre plus de 100 000 entrées dans certains cas. Par ailleurs, la fine granularité utilisée pour différentier les différents trafics entre-eux, tend à augmenter la complexité des règles. Par conséquent, la complexité du problème de la classification explose selon plusieurs angles : nombre de règles élevées, nombre de champs pris en compte, règles avec des contraintes complexes, débits à offrir en très forte augmentation. Dans un tel contexte, les techniques de classifications présentées dans la littérature présentent un potentiel de sous performer, en raison d'une complexité accrue des règles à supporter. Au travers de cette introduction, on a présenté pourquoi le "Software Defined Networking" génère de nouvelles problématiques pour la classification de paquet ; règles complexes, nombre élevés d'entrées. Les algorithmes présents dans la littérature n'ayant été conçus pour ce contexte, la conception d'un algorithme applicable au contexte de SDN apparait comme inévitable. Ainsi, le but de ce travail est de concevoir un algorithme de classification optimisé pour le contexte du SDN (sous certaines conditions). Néanmoins, avant de concevoir un algorithme, il est nécessaire d'analyser la performance offerte par les algorithmes actuels dans le contexte SDN, d'identifier d'éventuels défauts, pour ensuite concevoir un algorithme adressant ces différentes limitations.

Ce mémoire se compose d'une revue de littérature présentée dans le Chapitre section 2. Le Chapitre 3 expose la démarche de l'ensemble du travail réalisé ici, et vise à justifier la cohérence des articles inclus dans ce mémoire, par rapport aux objectifs de recherche. La première contrubution de ce mémoire, exposée dans le Chapitre 4, consiste en une analyse des performances d'un algorithme de classification dans le contexte de SDN. Ce même chapitre présente aussi les résultats des optimatisations réalisées. Une analyse complémentaire est réalisée pour la conception de l'algorithme AcceCuts, et est présentée dans le Chapitre 5. AcceCuts permet de réduire le nombre moyen d'accès mémoire par un facteur 3 en moyenne, et la consommation de mémoire par 40% en moyenne par rapport à EffiCuts, un algorithme de l'état de l'art de la classication de paquets. Ensuite, dans le chapitre 6, une discussion

générale en regard des résultats et des aspects méthodologique est présentée. Le Chapitre 7 résume le travail réalisé et aborde les enjeux futurs à considérer pour donner suite à ce travail.

Champ	Bits	Applicable	Remarques
Port d'entrée	(Dépend de l'implémentation)	Tous les paquets	Tous les paquets
Adresse source Ethernet	48	Tous les paquets selon les ports activés	
Adresse destination Ethernet	48	Tous les paquets selon les ports activés	
Type Ethernet	16	Tous les paquets selon les ports activés	Un commutateur OpenFlow est requis pour détecter le type dans les deux standards, Ethernet et 802.2, avec une entête SNAP et OUI de 0x000000. La valeur spéciale de 0x05FF est utilisée pour détecter tous les paquets 802.3 sans entête SNAP.
VLAN id	12	Tous les paquets de type Ethernet 0x8100	
Priorité VLAN	3	Tous les paquets de type Ethernet 0x8100	Champ VLAN PCP
Adresse IP source	32	Tous les paquets IP et ARP	Peut être des sous-réseaux masqués
Adresse IP de destination	32	Tous les paquets IP et ARP packets	Peut être des sous-réseaux masqués
Protocole IP	8	Tous les paquets IP et ARP	Seuls les premiers 8 bits de l'entête ARP sont utilisés
bits de IP ToS	6	Tous les paquets IP	Spécifié comme une valeur de 8-bits pour laquelle les 6 bits supérieurs proviennent de ToS
Port source de la couche Transport / Type ICMP	16	Tous les paquets TCP, UDP, and ICMP	Seuls les 8 bits inférieurs sont utilisés pour le type ICMP
Port de destination de la couche transport / Type ICMP	16	Tous les paquets TCP, UDP, and ICMP	Seuls les 8 bits inférieurs sont utilisés pour le type ICMP

Figure 1.2 Champs pris en compte dans OpenFlow v1.0.0. Figure traduite provenant de (Open Networking Foundation, 2009)

CHAPITRE 2

REVUE DE LITTÉRATURE

Ce chapitre vise à présenter dans un premier temps les concepts généraux de la classification de paquet, en présentant un point de vue global, puis en introduisant les approches phares étudiées dans la littérature. Dans la seconde partie de ce chapitre, l'emphase est mise sur les algorithmes adoptant le point de vue géométrique et utilisant des arbres de décision, en raison du degré de performance atteint. A cet égard, les concepts et techniques nécessaires à la compréhension de ces dits algorithmes, sont détaillés, puis une analyse approfondie de ces mêmes algorithmes est effectuée.

2.1 Découpage et projection

De manière globale, deux techniques sont employées par les algorithmes de classification : le découpage ou les projections. Une troisième technique peu utilisée, procède à un regroupement des règles initialement contenue dans la table de la classification. Cette technique, est désignée par le nom "Group Set" ou groupement d'ensemble.

De nombreuses méthodes se basent sur une approche géométrique du problème de la classification de paquets. Dans un tel cas, chaque champ de l'entête d'un paquet peut être vu comme définissant une dimension d'un espace. La prise en compte de plusieurs champs dans le cas de la classification de paquets permet donc définir des espaces multidimensionnels, comportant autant de dimensions que de champs pris en compte par les règles de classification. À cet égard, dans le chapitre suivant le mot champ et dimension sont utilisés comme des synonymes. Dans un tel espace multidimensionnel, un paquet est vu comme un point de l'espace. Les règles étant composées de contraintes pouvant s'exprimer sous la forme d'intervalles, celles-ci définissent des volumes dans l'espace multidimensionnel. De manière plus précise, chaque règle décrit un hyper-cube de l'espace. Par conséquent, dans un tel cas, le problème de la classification de paquets revient à identifier la ou les hyper-cubes (règle) contenant un point défini par les valeurs de chacun des champs de l'entête du paquet.

L'approche globale de la classification de paquets vise à éliminer de manière itérative les règles non reliées au paquet entrant. Pour les deux techniques suivantes, les explications sont illustrées au travers de la figure 2.1 et 2.2. Chaque figure définit un espace sur deux dimensions (équivalent à deux champs), dans lequel chaque rectangle de couleur représente une règle. La distribution des règles est identique pour chacune des deux figures. Chaque axe

est associé à un champ de l'entête du paquet. Néanmoins, ici, le type de champ considéré a peu d'importance, c'est pourquoi les axes sont volontairement omis. Par ailleurs, pour ces deux exemples, les valeurs numériques important peu, aucune graduation n'est présente.

La technique de découpe, consiste ainsi à diminuer l'espace de recherche global en sous-ensembles, contenant chacun un nombre restreint d'hyper-cubes (règles) par rapport à l'ensemble multidimensionnel parent. A titre d'illustration, dans la figure 2.1, la région initiale est découpée en plusieurs sous-régions (délimitées par des lignes hachurées), chacune contenant un nombre de règles inférieur à la configuration initiale. La méthode de classification consiste maintenant à identifier quelle sous-région sélectionner, et ensuite parcourir les règles à l'intérieur de cette sous-région.

Une autre technique utilisée par les algorithmes de classification consiste à projeter les bornes des différents intervalles sur chacune des dimensions considérée. La projection permet aussi de réduire le nombre de règles considérées et de progressivement réduire l'espace de règles se projetant sur le même sous-ensemble géométrique. L'idée sous-jacente est de considérer la classification sur plusieurs dimensions comme l'intersection de la classification sur chacune des dimensions. Pour ce faire, les règles sont projetées sur chacune des dimensions, et pour chaque intervalle définit, une liste des règles couvrant cet intervalle y est associé. Il suffit ensuite de regarder dans quel intervalle le paquet entrant se situe, et de sélectionner l'intersection des règles selon l'ensemble des dimensions.

Afin d'illustrer cette technique, considérons la figure 2.2, contenant la même distribution de règles utilisée pour l'exemple précédent. Les traits hachurés représentent ici les projections des règles sur chacun des deux axes.

La technique de projection a une granularité plus fine que la technique dite de découpe comme illustré dans la figure 2.1, 2.2. Néanmoins, le processus de localisation d'un intervalle projeté est plus compliqué que la localisation d'une sous-région résultant du processus de découpe, en raison d'un nombre plus important d'intervalles à comparer. De même, il est évident qu'une telle méthode pose des problèmes d'extensibilité avec le nombre de règles considérées.

Les algorithmes adoptant les techniques de découpage se basent sur la valeur de certains bits, provenant d'un ou plusieurs champs, pour découper et ainsi diviser l'ensemble initial de règles, en plusieurs sous-ensembles de règles. Le choix des bits ainsi que du ou des champs utilisés est déterminé de sorte à maximiser la séparabilité des règles ; c'est-à-dire de façon à ce que chaque sous-ensemble contienne un nombre similaire de règles et donc que la division de l'espace soit équilibrée. Lorsque la sous-région en cours de découpe possède un nombre de règles inférieur à une valeur seuil établie de manière empirique, alors le processus de découpe est stoppé. Afin de conserver la trace de ce découpage de l'espace, une structure de type

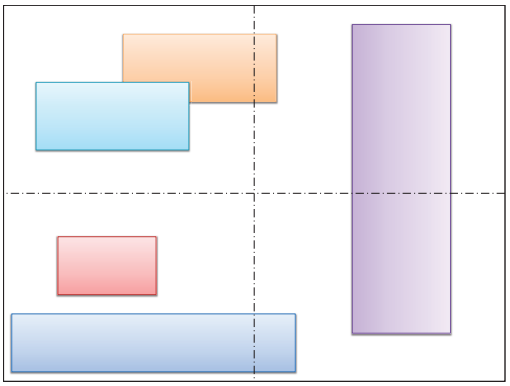


Figure 2.1 Découpage

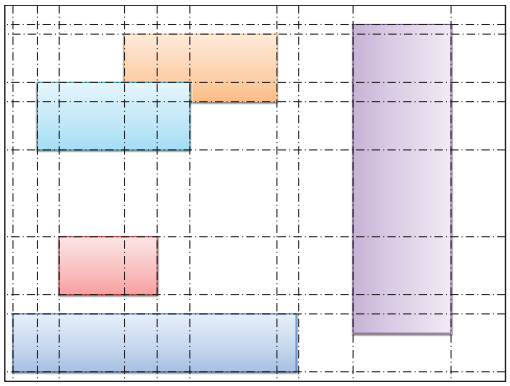


Figure 2.2 Projection

arbre de décision est construite. Par conséquent, les algorithmes adoptant ces techniques de découpe et séparation des règles, sont désignés comme des algorithmes basés sur des arbres de décision. On peut citer HiCuts (Gupta et McKeown, 2000), HyperCuts (Singh *et al.*, 2003), EffiCuts (Vamanan *et al.*, 2010), Adaptive Binary Cutting (Song et Turner, 2013) ainsi que AcceCuts, soit l'algorithme présenté dans ce mémoire.

Une approche dans la continuité de la technique de projection, consiste à décomposer le processus d'identification en plusieurs recherches parallèles (Srinivasan *et al.*, 1998), dans le but d'identifier la ou les règles associées au paquet entrant. En effet, l'idée sous-jacente est qu'il est plus simple de réaliser une opération de comparaison sur une partie d'une règle, que sur la règle en entier. Par conséquent, en suivant cette idée, l'entête du paquet peut être décomposée en plusieurs sous champs et, ainsi, chacun de ces sous-champs peut être comparé à un sous-ensemble de règles. L'intersection de chacune des comparaisons partielles permet de reconstituer la comparaison avec l'intégralité de l'entête du paquet. Cette méthode, faisant levier sur des comparaisons partielles, a vu l'émergence d'autres d'algorithmes (Gupta et McKeown, 1999; Taylor et Turner, 2004), avec différents compromis entre stockage et débit.

Ces différentes approches algorithmiques ont d'abord vu leur performance évaluée d'un point de vue théorique, puis des implémentations matérielles de ces algorithmes ont ensuite été réalisées dans la littérature (voir la sous-section 2.4). La performance obtenue varie grandement suivant la plateforme sur laquelle l'implémentation est effectuée et la manière dont l'implémentation est réalisée, comme nous le verrons dans la suite de cette section.

Parallèlement aux approches algorithmiques, dans la littérature, une solution purement matérielle a été proposée et développée spécifiquement pour atteindre un niveau de performance très élevé en termes de débit de paquets, la TCAM, ou "Ternary Content Addressable Memory" Comme mentionné précédemment, la classification de paquets vise à comparer l'en-

tête du paquet à l'ensemble des règles présentes dans la table de classification. Néanmoins, suivant ce principe, si l'analyse se fait de manière linéaire, le temps de traitement est beaucoup trop élevé. Une approche proposée consiste à pousser à l'extrême en réalisant le parcours intégral des règles de manière parallèle. Un type de mémoire spécifique, la TCAM a été développé dans ce dessein, soit d'effectuer de façon totalement parallèle la comparaison des règles.

2.2 TCAM : solution purement matérielle

La mémoire de type TCAM est massivement utilisée dans l'industrie des routeurs de haute performance pour réaliser la classification de paquets en raison de la bande passante fournie. La TCAM est une mémoire dont la structure particulière réalise une opération de comparaison sur l'ensemble des entrées (règles) présentes en mémoire et ce, de manière simultanée. Les règles sont représentées dans une TCAM sous forme de chaines de bits triées par priorité décroissante. Lors d'une recherche, l'index de la règle sélectionnée est ensuite utilisé pour accéder à la donnée associée à la règle. La TCAM permet d'identifier une règle en un temps *O(1)* et, par conséquent, elle offre un niveau de performance très élevé. Néanmoins, malgré une telle performance en termes de recherche, un problème majeur associé à la TCAM pour classifier des paquets concerne l'efficacité de la gestion des règles utilisant des intervalles. En effet, la difficulté est liée au fait que plusieurs entrées de la TCAM doivent être allouée en mémoire pour représenter une règle décrite par des intervalles. En effet, la TCAM est très performante lorsqu'il s'agit de faire des comparaisons de type binaire ou "wildcard" mais pas dans le cas d'intervalles. Un intervalle associé à un champ est défini par une borne supérieure et une borne inférieure, dont les valeurs extrêmes sont encadrées par les bornes du champ associé. À titre d'exemple, si l'on suppose un intervalle de type *>1023*, pour un champ dont la valeur extrême supérieure est 65536, alors six règles sont nécessaires pour décrire cette entrée, comme présenté dans le tableau 2.1 (règles sous forme binaire). Dans cet exemple, un symbole "*" désigne un bit qui n'est pas examiné.

Or, les tables de classification utilisées comportent un nombre important de règles décrites par des intervalles (Yaxuan *et al.*, 2009). Bien qu'il s'agisse d'analyses non récentes, les conclusions demeurent valides. En effet, la tendance actuelle vise à considérer un plus grand nombre de champs au niveau des tables de classification, tout en gardant les champs utilisés précédemment et le type de contraintes associées. Ainsi, des analyses réalisées dans (Chao et Liu, 2007), ont permis de mettre en évidence que l'efficacité de stockage de la TCAM peut atteindre des taux très faibles, proche de 16% dans certains cas. Néanmoins, la prise en charge optimisée d'intervalles tend à diminuer la bande passante offerte par la TCAM. Par ailleurs,

Tableau 2.1 Entrées dans la TCAM associées à la contrainte *>1023* pour un champ sur 16 bits

Contrainte
0000 01** **** ****
0000 1*** **** ****
0001 **** **** ****
001* **** **** ****
01** **** **** ****
1*** **** **** ****

au problème de la prise en charge de règles avec intervalles s'ajoute une consommation très élevée et un cout d'achat prohibitif. Le facteur cout de la TCAM s'explique par plusieurs raisons. Tout d'abord le stockage d'un bit de donnée dans un TCAM nécessite jusqu'à 16 transistors (Narayan *et al.*, 2003), là où la SRAM en nécessite 6 et la SDRAM seulement 1. Par conséquent, la densité de stockage de la TCAM est sensiblement inférieure à beaucoup d'autres types de mémoire usuels. Par ailleurs, le marché relativement limité de la TCAM , par rapport aux autres types de mémoire, rend cette technologie fortement dispendieuse, avec un cout estimé par bit de l'ordre de 20 fois supérieur à la SRAM et presque cent fois plus élevé par rapport à la SDRAM. L'utilisation élevée de transistors est principalement due au parcours parallèle de l'ensemble des entrées contenues dans la TCAM. Des optimisations ont été faites afin de regrouper les entrées en différents segments, qui peuvent être parcourue sélectivement afin de réduire la consommation énergétique. Néanmoins, de telles solutions, bien que réduisant la consommation de mémoire Chao et Liu (2007), tendent à diminuer la bande passante offerte et donc réduisent l'avantage principal de la TCAM.

La TCAM pose des problèmes pour le support du "multiple match", ce qui peut restreindre son utilisation. En effet, les nouvelles applications de type sécurité, telles que la détection d'intrusions, nécessitent que l'ensemble des règles associées au paquet entrant soit rapporté. Or, dans le cas de la TCAM, seule la règle avec la plus haute priorité est conservée. L'émergence de ce nouveau besoin fait défaut à une bonne partie des TCAMs actuellement disponibles. Différentes solutions ont été proposées, selon différentes perspectives (Chao et Liu, 2007). Néanmoins, cette limitation de la TCAM la place clairement à l'écart de l'ensemble des autres solutions algorithmiques qui dans la presque totalité supportent nativement le "multiple match". Enfin, en plus des divers problèmes présentés ci-dessus, la TCAM peut présenter des problèmes pour réaliser des mises à jours ne perturbant pas le trafic (Wang *et al.*, 2004). Par conséquent, de manière conjointe au développement d'optimisations pour la TCAM, différentes approches algorithmiques ont été proposées et implémentées. Elles sont basées sur des mémoires plus conventionnelles, afin de faire levier sur le faible prix de ces mémoires et

de la grande capacité de stockage qu'elles offrent.

2.3 Approche algorithmique de la classification de paquet

De cette revue de littérature, certains algorithmes sont volontairement exclus, car ils ont été conçus pour un contexte très restreint qui n'est plus utilisé à l'heure actuel. En effet, plusieurs algorithmes faisant levier sur des arbres binaires ne supportent pas, ou avec beaucoup trop de limitations les règles exprimées sous la forme d'intervalles arbitraires, ainsi qu'un nombre de règles élevés. Ainsi dans (Feldman et Muthukrishnan, 2000), une structure de donnée est proposée, mais spécifiquement pour un contexte de classification où seules deux dimensions sont utilisées. Par ailleurs, bien que cet algorithme offre une performance acceptable en termes de consommation mémoire et de nombre d'accès mémoire, ce dernier est uniquement adapté à des tables de classification sur deux dimensions. En effet, la performance obtenue lorsque davantage de dimensions sont considérées est faible. Par conséquent, cet algorithme ne sera pas présenté plus en détails, car la performance offerte ne permet ni une utilisation dans le contexte traditionnelle de la classification de paquet, ni dans le contexte présenté dans le chapitre 1, et plus particulièrement dans la figure 1.2.

En adoptant une démarche similaire, on élimine les algorithmes conçus pour supporter un faible nombre de champs ou ne pouvant effectuer certains types de comparaisons. À cet égard, l'algorithme présenté dans "Tradeoffs for Packet Classification" (Feldman et Muthukrishnan, 2000) est éliminé, ainsi que des algorithmes utilisant des fonctions de hachage.

2.3.1 Algorithmes basés sur la projection ou la décomposition

Une famille d'algorithmes utilisant la technique dite de projection a été largement développée dans la littérature. On peut ainsi citer "Aggregated Bit Vector" (Baboescu et Varghese, 2001) ainsi que "Bit Vectors" (Lakshman et Stiliadis, 1998). Ces deux algorithmes décomposent la recherche multidimensionnelle en autant de recherches unidimensionnelles que de champs traités. Ainsi, pour chaque dimension utilisée, un arbre est créé, dont les nœuds contiennent les informations permettant de connaitre la ou les règles associées à la comparaison partielle. A chaque recherche est associé un ensemble de règles pouvant correspondre à la comparaison partielle. Afin de déterminer la règle résultante du processus de classification, il suffit d'identifier la ou les règles partagées pour l'ensemble des résultats.

Un autre algorithme basé sur cette approche, "Cross producting scheme" (Srinivasan *et al.*, 1998). Il a été conçu afin de supporter un nombre arbitraire de champs et d'être indépendant du type d'applications. Il vise à effectuer une recherche selon chacune des dimensions d (ou champs) utilisés et la combinaison des résultats partiels de chacune des recherches permet

d'accéder à la règle correspondante contenue dans une table dont les entrées ont été pré-calculées initialement. Cet algorithme offre une complexité en temps de recherche faible en $O(d \cdot t_r l)$, où $t_r l$ est la complexité en temps de recherche pour identifier un intervalle selon un champs. Cette faible complexité de recherche s'obtient au prix d'une consommation de mémoire gigantesque, puisque la table de "Cross-product" peut avoir $O(m^d)$ entrées. De même, le processus de mise à jour est complexe car il demande la reconstruction complète de la table pré-calculée. Ainsi, en raison d'un cout de stockage prohibitif, l'algorithme "Cross Producting Scheme" n'est pas adapté à une utilisation avec un nombre élevé de règles.

Bien que ces algorithmes offrent une performance élevée en termes de vitesse de classification, ils consomment néanmoins une quantité excessive de mémoire. Cela rend donc ces algorithmes peut compatibles avec des applications nécessitant un nombre important de règles de classification. Dès lors, de nouveaux algorithmes ont vu le jour, afin de diminuer la surconsommation mémoire.

Comme mentionné précédemment, les algorithmes de classification doivent généralement faire un compromis entre rapidité de traitement, débit et consommation mémoire. À cet égard, "Recursive Flow Classification" (RFC) (Gupta et McKeown, 1999) et "Distributed Cross-producting of Field Labels" (Taylor et Turner, 2004) offrent un compromis plus intéressant entre la consommation mémoire et la bande passante, par rapport aux algorithmes précédents, mais souffre néanmoins de limitations. Malgré une consommation mémoire améliorée par rapport aux algorithmes antérieurs, celle-ci demeure importante (Singh *et al.*, 2003). En effet, RFC est une variation de l'algorithme "Cross producting scheme" et, à cet égard, hérite de son défaut relatif à une grande consommation mémoire. RFC utilise une méthode heuristique pour classer les paquets. Considérons que le processus de classification implique l'association de l'entête d'un paquet de H bits à un identifiant d'action de T bit et que N règles sont présentes dans la table de classification utilisée. Une méthode simple, mais clairement sous-optimale, consiste à pré-calculer l'action associée à chacun des 2^H paquets différents. Un tel processus nécessiterait l'utilisation d'un tableau de taille $2^H \cdot 2^T$. Un seul accès mémoire serait alors nécessaire pour identifier l'action associée à un paquet entrant, mais la consommation mémoire serait tout simplement astronomique. Le but de RFC est de procéder à la même association, mais au travers de plusieurs étages ou phases comme présenté dans la figure ci-dessous. Ainsi, l'association s'effectue de manière récursive : à chaque étage, l'algorithme procède à une réduction de la largeur de bits considérés.

Dans la figure 2.3, trois étages sont utilisés, mais RFC propose de faire varier le nombre d'étages suivant la performance envisagée. Dans le cas de l'exemple ci-dessus, les H bits utilisés pour identifier un paquet sont initialement découpés en plusieurs "morceaux", qui seront utilisés pour indexer différentes tables en mémoire lors de la première itération, c'est-

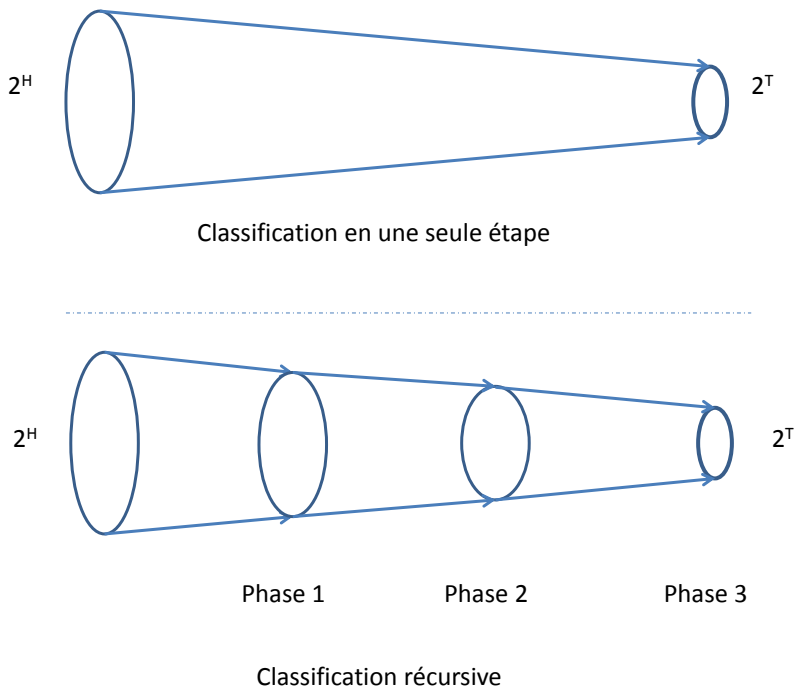


Figure 2.3 Illustration du principe de l'algorithme RFC. Figure traduite et provenant de Gupta et McKeown, 1999

à-dire menant vers la phase 1. Chacune de ces recherches va associer à un "morceau" un identifiant noté *eqID*, qui sont ensuite combinés et à nouveau découpés en "morceaux". Ces derniers vont être utilisés pour indexer différentes tables, lors de la même phase. Ce processus est répété jusqu'à ce que la dernière phase soit atteinte, et permette de spécifier la ou les règles associées au paquet entrant.

RFC permet d'atteindre en pratique un débit de 10 Gbps une fois implémenté en matériel mais, d'une part, ce débit offert semble relativement faible par rapport la vitesse des liens actuels et, d'autre part, RFC souffre d'une consommation mémoire élevée, ainsi que d'un temps de pré-calcul important lorsque le nombre de règles devient supérieur à 6000. Ces deux algorithmes présentent une capacité à monter en charge limité, ainsi qu'un compromis de performance médiocre entre temps de recherche et l'utilisation mémoire. Par conséquent, il est pertinent de se tourner vers des algorithmes adoptant d'autres approches et minimisant les compromis extrême entre temps de recherche et utilisation mémoire.

2.3.2 Algorithme adoptant l'approche "Group Set"

Une autre technique, peu utilisée, a été proposée sous le nom de "Tuple Space Search" (TPS) (Srinivasan *et al.*, 1999), et vise à regrouper les règles contenues dans une table de

classification en des ensembles disjoints, selon une propriété, puis de traverser de manière parallèle chacun de ces ensembles. Le but est de décomposer l'étape de classification en plusieurs recherche de type comparaison "exactes" dans des tables de hachage. Là encore, l'identification de la règle est effectuée à partir du résultat partiel de chacune des comparaisons. La motivation sous-jacente à cet algorithme est liée aux caractéristiques des tables de classification.

Plus précisément, TPS regroupe les règles en n-uplets, en fonction de la longueur des préfixes pour les différents champs considérés. Chaque groupe est ensuite stocké dans une table de hachage. L'opération de classification consiste en une opération de comparaison exacte sur chacune des tables de hachage. À chaque règle R, contenue dans la table de classification, peut être associé un *d-uplet*, pour lequel la *i-ème* composante exprime la longueur du préfixe du *i-ème* champ de R. Un espace de tuplets est défini comme étant l'ensemble des tuplets créés. Les adresses IP d'une règle sont toujours spécifiées à l'aide d'un préfixe et l'utilisation d'un tel algorithme est donc optimale pour ces deux champs. Néanmoins, pour les champs Port Source et Port Destination (ou tout autre port exprimé sous la forme d'un intervalle), l'algorithme propose deux méthodes : la première, permettant de savoir dans quel tuplet placer la règle, et une autre afin d'identifier la règle au sein du tuplet. Une fois les règles classées dans le bon tuplet, le processus de classification reposant sur une méthode de hachage est relativement simple. En effet, toutes les règles associées au même tuplet partagent le même masque et ce, pour l'ensemble des champs considérés. Par conséquent, il suffit de concaténer le nombre de bits requis pour chacun des champs afin de construire une clé de hachage pour chaque règle. Toutes les règles associées à un certain tuplet sont stockées dans une même table de hachage. Une requête dans une table de hachage est effectuée en sélectionnant le nombre de bits adéquats de l'entête du paquet entrant puis en appliquant la fonction de hachage avec la clé ainsi créée. De manière identique, les clés construites à partir de l'entête du paquet sont hachées dans chacune des tables afin de trouver la règle correspondante. Pour un paquet donné, il suffit donc de parcourir l'ensemble des tuplets et de garder la règle avec la plus haute priorité parmi les règles ayant causées une comparaison positive.

Performance

La complexité du temps de recherche est de $O(M)$, où M est le nombre de tuplets contenus dans l'espace des tuplets, sous l'hypothèse d'un hachage parfait (i.e. permettant d'éviter toute collision). Par contre, M demeure important et très variable pour beaucoup de cas réels testés (Srinivasan *et al.*, 1999; Song, 2006). Par conséquent une optimisation a été présentée afin d'augmenter la vitesse des requêtes ainsi que la performance des mises à jour (Srinivasan *et al.*, 1999). La complexité de stockage de la structure donnée créée est de

$O(N \cdot d \cdot W)$, avec W la longueur du plus long préfixe, N le nombre de règles et d le nombre de champs considérés, permettant d'atteindre un temps de recherche de $O(\log W)$. Par ailleurs, en pratique, on remarque que la performance varie grandement en fonction du type de règles considéré.

Par ailleurs, les résultats présentés ci-dessus sont valables lorsque l'on considère des fonctions de hachage parfaites, n'entrainant aucun faux positif. Néanmoins, les fonctions de hachage sont particulièrement utilisées pour la classification de paquets et, plus particulièrement, dans les fonctions de type "Filtres de Bloom" (Dharmapurikar *et al.*, 2006). Néanmoins, ce type de filtre, bien qu'il permette de savoir avec certitude qu'un élément est absent d'un ensemble considéré, permet avec une certaine probabilité seulement de savoir qu'un élément est éventuellement présent dans l'ensemble. Ces solutions de hachage ne permettent pas d'offrir une performance déterministe en raison de collisions potentielles (Jiang et Prasanna, 2009) et de la sous-efficacité à gérer les règles ayant des entrées de type générique ("Wildcard") ou ayant des intervalles quelconques (Dharmapurikar *et al.*, 2006). Donc, une seconde opération de recherche est requise afin de résoudre les faux positifs, ce qui tend à augmenter la complexité de la classification du paquet et donc à limiter la performance globale atteignable (Sourdis et Sourdis, 2007).

2.3.3 Algorithmes découpant l'espace de recherche

La technique de découpage vise à identifier la ou les règles associées au paquet entrant, en réduisant de manière itérative l'espace de recherche, en effectuant des découpes, guidée par des heuristiques exploitant la structure de la table de classification. Les découpes et autres traitements excécutés sont utilisées pour construire un ou plusieurs arbres de décision. Ainsi ces algorithmes fonctionnent en deux temps : création d'un ou plusieurs arbre(s) de décision, associé(s) à une table de classification, puis parcours de la structure générée, basée sur la valeur des champs du paquet d'entrée. Par conséquent, de tels algorithmes s'appuient sur un algorithme de construction d'arbre, et un algorithme pour la traversée de l'arbre. L'étape de traversée de l'arbre constitue ici l'étape de classification de paquets.

De nombreux algorithmes sont présents dans la littérature, mais la plupart des algorithmes publiés héritent de trois algorithmes majeurs, qui seront étudiés ici en détails : HiCuts, HyperCuts et EffiCuts (Gupta et McKeown, 2000; Singh *et al.*, 2003; Vamanan *et al.*, 2010).

Construction

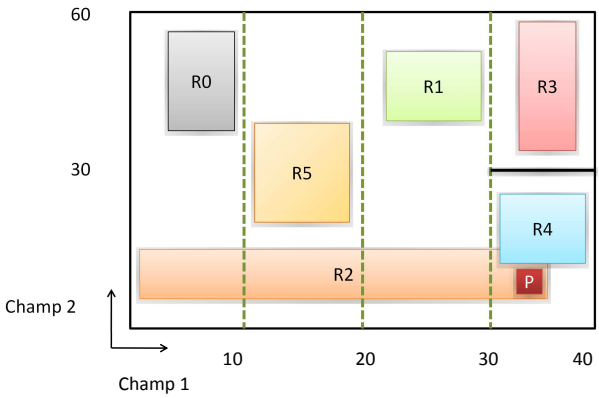
L'ensemble de ces algorithmes divise l'espace des règles en sous-espaces, de manière itérative, jusqu'à ce que chacun des sous-espaces contienne un nombre de règles inférieur ou égal

à une valeur seuil définie préalablement à l'exécution de l'algorithme. De tels algorithmes construisent un ou plusieurs arbres de décision lors de la découpe de l'espace de recherche. Un tel arbre est composé de nœuds et, à ses extrémités, de feuilles. Chacun des nœuds couvre une zone de l'espace de recherche et, à cet égard, contient différentes informations permettant de définir les limites de la zone couverte par le nœud, le nombre de nœuds enfants associés et leur position. Les informations de localisation semblent redondantes, mais les nœuds et enfants associées peuvent êtres découpes et donc décrits selon des dimensions différentes. Les feuilles, sont des nœuds contenant simplement des règles, ainsi aucune description de la zone couverte, ni autre indications ne sont présentes.

Afin d'illustrer la construction d'un arbre de décision, on considère l'espace de recherche contenant les règles présentées dans la figure 2.4, R_1 à R_6. Le rectangle rouge P correspond à un paquet P et sera util seulement pour la partie suivante, traitant du parcours de l'arbre. Chaque axe dans la figure est associé à un champ de l'entête du paquet. Les graduations utilisées pour chacun des axes, servent à mieux comprendre la génération de l'arbre de décision. Les zones couvertes par chacune des nœuds sont représentées sous la forme d'intervalles, $dim_1 \times dim_2$.

Pour cet exemple, on considère une valeur seuil fixée arbitrairement à 2. Le processus de construction débute en considérant l'ensemble de l'espace de recherche, un nœud racine est créé, couvrant l'ensemble de la zone délimitée par le rectangle noir, c'est-à-dire [0 ;40] selon le champ 1 et [0 ;60] selon le champ2. Ensuite, une première phase de découpe est réalisée selon le champ 1, ce dernier permettant de maximiser la séparation des règles entre elles. Les découpes réalisée sont de taille égale, et la première phase de découpe crée 4 sous-espace, chacun de taille 10 selon le champ 1. Ces découpes sont représentées par des lignes vertes hachurées dans la figure 2.4. On remarque que les trois première zones, c'est-à-dire dont les intervalles couverts s'échelonnent entre 0 et 30 selon le champ 1, contiennent chacun 2 règles, alors que la dernière zone contient 3 règles. Les informations spécifiant le nombre de découpes, ainsi que la dimension utilisée pour réaliser ces découpes sont ajoutées au noeud racine.

La valeur seuil déterminant si d'autres découpes doivent être effectuées ou non est ici égale à 2. Ainsi, chacun des noeuds (sous-espace) contenant plus de 2 règles se voit redécoupé. Par opposition, si un noeud contient moins de 2 règles, alors une feuille contenant ces règles et créé. Par conséquent, les trois premières zones remplissent les conditions pour qu'une feuille par zone soit créée. Trois feuilles sont donc crée (Feuilles 1 à 3) contenant chacune les règles couvertes par leur zone. Au contraire, la dernière zone contient un nombre de feuille supérieur à la valeur seuil, et à cet égard, une itération supplémentaire de découpe doit être effectuée. Un noeud associé à cette zone est donc crée, couvrant la zone [30 ;40] selon le champ 1 et [0 ;60] selon le champ 2. Découper cette zone selon le champ 1 serait peut pertinent, puisque



Paquet P	Champ 1	Champ 2
	36	12

Figure 2.4 Espace de recherche

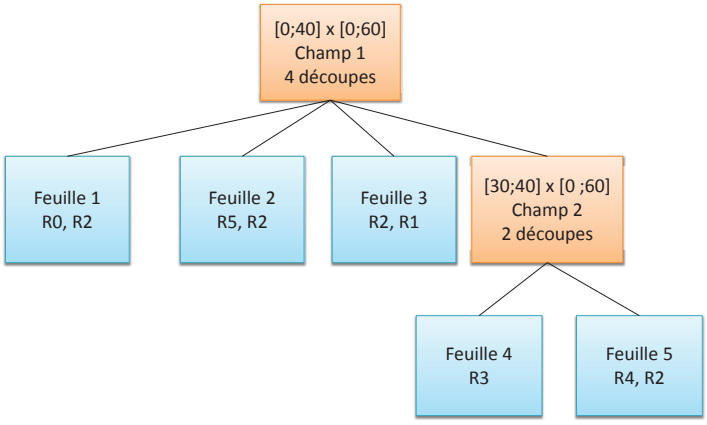


Figure 2.5 Arbre de décision associé à l'espace de recherche

les règles se chevauchent selon cette dimension. Par conséquent, le champ 2 est sélectionné, et une découpe suffit à séparer la zone initiale en deux zones contenant chacune 2 règles. Le trait noir représente ici la découpe ayant été réalisée. Chacune des nouvelle sous-zone crée contenant deux règles, les feuilles respectives sont créées (Feuilles 4 à 5). La construction de l'arbre est maintenant terminée, et l'étape suivante consiste en un parcours de l'arbre.

Parcours

Le parcours d'un arbre correspond d'un point de vue de la classification, à réduire de manière itérative l'espace de recherche jusqu'à identifier la ou les règles associées à un paquet. Cette étape consiste simplement en une traversée de l'arbre de décision généré. Il faut noter que dans le cas où plusieurs arbres de décision sont générés par l'algorithme, l'ensemble des arbres générés doit être parcouru.

Les algorithmes de classification présentés dans cette section construisent un arbre de décision constitué de nœuds et de feuilles. Comme présenté dans une section précédente, un nœud contient différentes informations : zone couverte, dimension(s) utilisée(s), nombre de découpes, ainsi que des informations permettant d'accéder aux nœuds enfants.

Afin de parcourir l'arbre, l'ensemble de ces informations est utilisé, et différents calculs sont effectués afin d'identifier le nœud enfant, et donc la branche inférieure de l'arbre à parcourir ensuite. Le processus de traversée de l'arbre commence au niveau du noeud nœud racine, et se poursuit à chacun des niveaux de l'arbre, tant qu'une feuille n'est pas atteinte.

Lors de la traversée, pour chacun des nœuds rencontrés, la première étape consiste à déterminer la ou les dimensions utilisées, ainsi que le nombre de découpes effectuées. Connaissant ces deux informations, la méthode de traversée consiste à déterminer les bornes de chacune des sous-zones découpée, et d'identifier dans laquelle le paquet se situe. À chaque sous-zone correspond un nœud enfant (qui peut être une feuille), ce qui signifie que l'étape suivante consiste simplement à accéder au nœud enfant approprié.

Lorsqu'une feuille est atteinte, chacune des règles contenues dans la feuille est parcourue et comparée à l'entête du paquet entrant. La ou les règles ayant entrainées une comparaison positive sont alors sélectionnées et utilisées pour les étapes ultérieures du traitement de paquet. Le processus de classification de paquet est alors terminé et un nouveau paquet peut être traité.

Dans l'exemple présenté dans la figure 2.6, on considère la même distribution de règles que celle présentée dans le schéma 2.4. On assume que l'algorithme a déjà réalisé les découpes illustrées dans le schéma et donc a abouti à la création de l'arbre de décision présenté ici. Il est à noter que les nœuds contiennent de l'information permettant un parcoure de l'arbre, mais que l'information présente ici peut différer selon les algorithmes considérés, et que certains

éléments d'information présents ici, le sont à des fins d'illustration seulement.

Au vu de l'espace considéré, le nœud racine couvre l'intégralité de l'espace de recherche, c'est-à-dire ici *0* à *40* selon la dimension *1*, et *0* à *60* selon la dimension *2*. Lors du parcours de l'arbre, après avoir récupéré les informations précisant le nombre de découpes et les dimensions utilisées, la taille des découpes doit être déterminée. Dans notre exemple de la figure 2.6, il y a 4 zones découpeés selon la dimension *1*. Les découpes étant de taille égales, chacune d'elle couvre une zone de *40/4* soit égale à *10* selon le champ 1. Le paquet entrant a la valeur *36* selon le *champ 1*, c'est-à-dire qu'il se situe dans la zone *36/10* ou quatrième zone. Le quatrième nœud enfant (qui peut être soit une feuille, soit un nœud) doit donc être lu. Dans notre exemple le noeud enfant associé est un nœud, dont les zones découpées le sont selon la *dimension 2*, couvrant la zone *[0 ;60]*, et 2 découpes sont effectuées. Le paquet a un *champ 2* ayant une valeur de *12* et, par conséquent, en adoptant un raisonnement identique à celui développé pour le nœud précédent, la zone de découpe identifiée correspond à la zone numéro 1. Cette zone est associée à la région $[30; 40] \times [0; 30]$.

Le nœud enfant associée correspond ici à une feuille, couvrant deux règles. Le processus de traversé d'une feuille est appliqué, et chacune des règle est parcourue. La comparaison s'avère positive pour une seule règle, en l'occurrence la règle R_2. Le processus de classification est maintenant terminé et un nouveau paquet peut désormais être traité.

Heuristique

Un pivot important dans la performance des différents algorithmes présentés ci-dessous concerne l'heuristique utilisée pour sélectionner les découpes à effectuer, d'une part et, d'autre part, l'heuristique déterminant le nombre de découpes à effectuer au niveau d'un nœud. Dans le flot de traitement de l'algorithme, la première heuristique utilisée correspond à la sélection de la ou les dimensions (dépendamment de l'algorithme considéré) selon laquelle effectuer des découpes. Cette dernière doit permettre de sélectionner la ou les dimensions selon laquelle les règles sont les plus disjointes afin de permettre de réaliser des découpes optimales, c'est à dire minimisant la réplication, la profondeur et la largeur de l'arbre. En effet, lorsqu'une dimension non adéquate est choisie, les découpes ensuite effectuées sont peu efficaces et rendent la séparation des règles longue. Si l'on regarde les figures 2.8 et 2.7, où deux cas de découpes sont représentés, la dimension utilisée correspond soit au champ 1 ou au champ 2. Dans la première figure 2.7, on constate ainsi que les découpes effectuées selon la dimension 1 permettent de créer 4 régions, contenant chacune au maximum 3 règles. Dans le cas d'une valeur seuil fixée à 2, cela signifie qu'une seule région nécessite une découpe supplémentaire. Par conséquent, l'arbre généré a une profondeur minimisée. Au contraire, dans le cas de la figure 2.8, les quatre découpes sont effectuées selon la dimension 2, mais

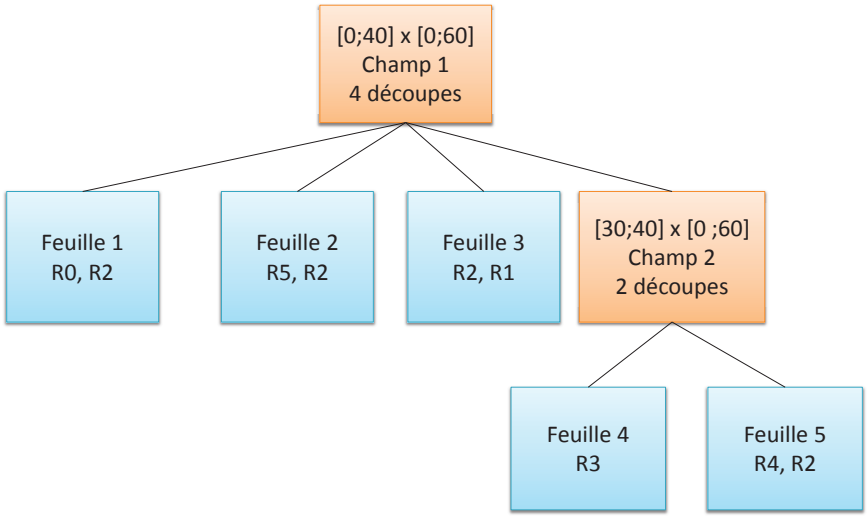


Figure 2.6 Arbre de décision associé à l'espace de recherche

génèrent plusieurs régions comportant quatre règles, c'est-à-dire que plusieurs régions vont devoir être redécoupées, ce qui va générer un arbre large et potentiellement profond. Au-delà de la taille de l'arbre, il est important de comprendre que de mauvaise découpes tendent à créer des feuilles ayant un nombre important de règles identiques, et donc augmenter de manière drastique le taux de réplication des règles. Une augmentation de la taille de l'arbre en profondeur va limiter la performance, que cela soit au niveau de la taille de la structure de donnée générée ou concernant le nombre d'accès mémoire. Notez qu'une augmentation de la taille en largeur nuit seulement à la taille finale de la structure de donnée générée.

Le principe partagé par les différentes heuristiques est d'identifier les dimensions selon lesquelles les règles sont les plus différentes et disjointes. A ce propos, une méthode usuelle adoptée pour l'heuristique consiste à identifier le nombre d'éléments uniques pour chacune des dimensions utilisées. Ce processus consiste à identifier le nombre de bornes différentes (ou composante d'une règle selon une dimension) par dimension. Une fois ce comptage effectué, les dimensions sont ordonnées par nombre d'éléments uniques qu'elles contiennent. Dépendamment de l'algorithme (découpe selon une ou plusieurs dimensions) considéré, soit la dimension contenant le plus grand nombre d'éléments unique est choisie, soit les dimensions contenant un nombre d'éléments uniques supérieur à la moyenne du nombre d'éléments unique.

On peut remarquer que cette méthode bien que couramment utilisée, a de sérieuse li-

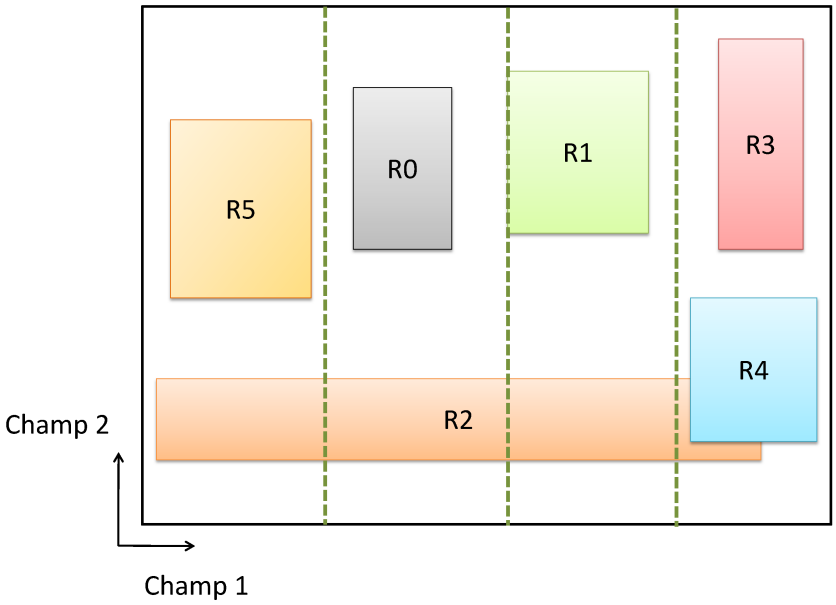


Figure 2.7 Illustration des choix possibles de découpe : Découpe selon la dimension 2

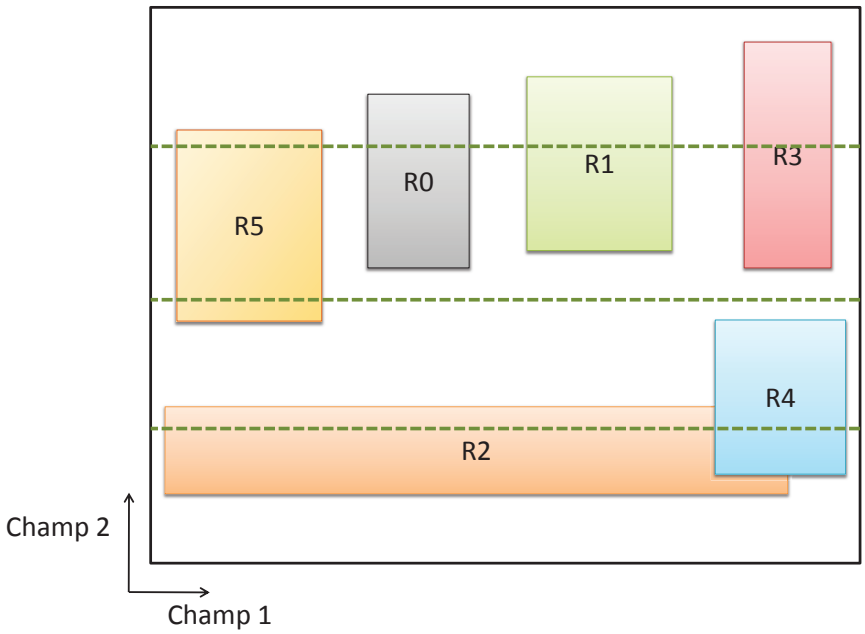


Figure 2.8 Illustration des choix possibles de découpe : Découpe selon la dimension 1

mites. En effet, une dimension peut être privilégiée par rapport à une autre, alors que le nombre d'éléments uniques est élevé mais centré autour d'une même valeur, ce qui risque de générer des découpes non optimales. Par conséquent, il serait pertinent de tenir compte aussi du nombre de règles incluses ou ayant des portions de couverture commune, et chercher les dimensions maximisant le nombre d'éléments uniques tout en minimisant les zones de couvertures superposées.

Comme mentionné précédemment, après avoir sélectionné la dimension selon laquelle effectuer des découpes, une seconde heuristique est chargée de déterminer le nombre optimal de découpes à réaliser au niveau d'un nœud. Cette opération est critique dans la mesure où un nombre trop élevé de découpes tend à créer beaucoup de nœuds et donc une grosse structure de donnée mais, par la même occasion, peut permettre de mieux séparer les règles entre elles et donc de réduire la profondeur de l'arbre. Par ailleurs, si l'arbre est large, le taux de réplication des règles a de forte chance d'être relativement élevé et donc de provoquer une surconsommation mémoire. On comprend donc que cette heuristique doit réaliser un compromis entre la taille de la structure de données générées et le nombre d'accès mémoire nécessaire à réaliser.

Considérons les deux figures présentées dans 2.9 et 2.10, présentant deux schémas de découpe. Le premier schéma illustré dans la figure 2.10, comprenant huit régions résultant du processus de découpe, le second illustré dans la figure 2.9 comportant le double, soit seize régions, pour les mêmes règles. On remarque dans la figure 2.9 que l'augmentation du nombre de découpes n'a pas permis de différencier davantage les règles, notamment pour la région contenant R_2 et R_4 et la région contenant R_5 et R_2. Pis, on constate la génération de zones ne contenant aucune règle. Par conséquent, dans le cas illustré, l'augmentation du nombre de découpes n'a pas permis de générer un arbre de plus petite taille, mais, en plus, la structure de donnée s'en trouve sensiblement alourdie. Après avoir illustré l'impact de l'heuristique utilisé pour calculer le nombre de découpes à effectuer au niveau d'un nœud, présentons en détail les méthodes couramment employées pour déterminer le nombre de découpes à effectuer.

Le nombre de découpes à effectuer au niveau d'un nœud est basé sur une métrique mesurant la quantité d'espace disponible pour la structure de recherche (des détails sont fournis dans la sous-section 4.5.2). C'est-à-dire que la métrique évalue par rapport à un budget donné, le nombre de découpes pouvant être effectuées à l'intérieur de ce budget. Il est à noter que l'augmentation du nombre de dimensions utilisées pour découper l'espace de recherche peu avoir un effet néfaste en augmentant la quantité d'information à stocker.

L'heuristique utilisée pour les découpes, est guidée par deux fonctions (la notation utilisée ici provient de Gupta et McKeown (2000)) : une fonction de mesure de l'espace, *Sm*, procédant à des découpes, guidée par une fonction *Spmf*, déterminant la valeur maximale que

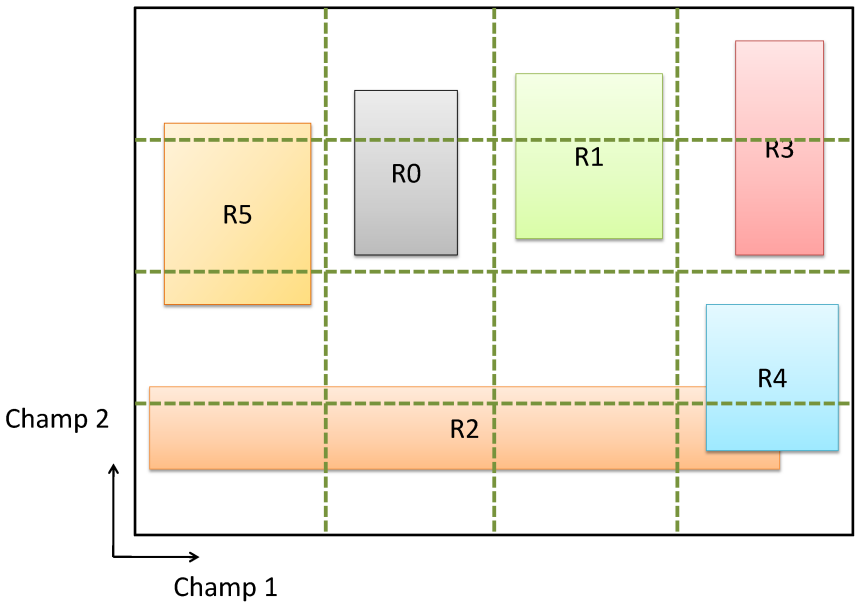


Figure 2.9 Illustration des choix possibles de découpes : Sur-découpage

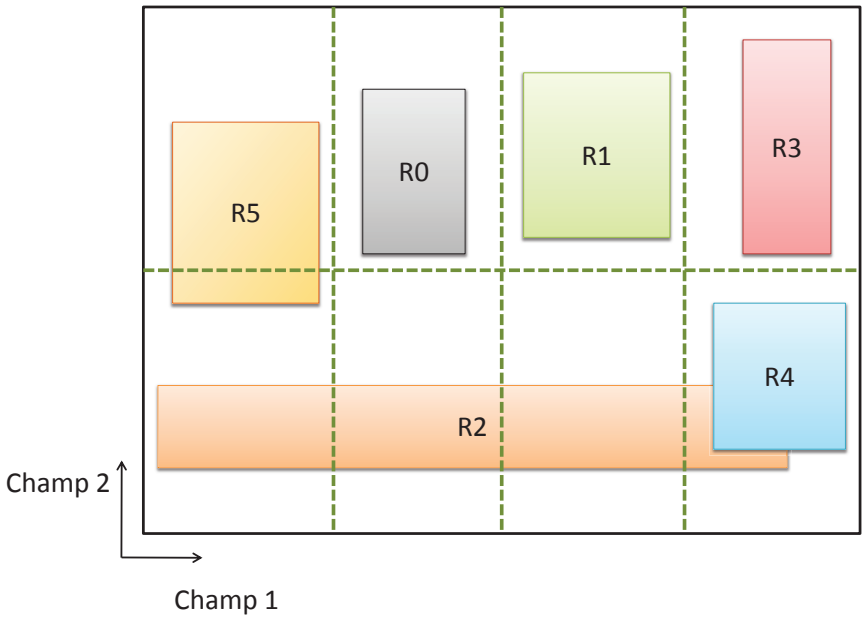


Figure 2.10 Illustration des choix possibles de découpes : Découpe optimale

peut prendre la fonction *Sm*. En effet, la fonction *Sm* est une fonction appelée de manière itérative, et cherchant à maximiser le nombre de découpes, tout en demeurant inférieure à la valeur fixée par la fonction *Spmf*. *Sm* est une fonction de mesure de l'espace, prenant en compte plusieurs paramètres : le nombre total de règles contenues pour chacune des sous-régions découpées, ainsi que le nombre de découpes effectuées. À chaque itération et tant que *Sm* demeure inférieur à la valeur de *Spmf*, le nombre de découpes est doublé. Lorsque la condition $Sm < Spmf$ est atteinte, les découpes calculées sont réalisées, et les différents nœuds enfants sont construits.

On verra plus tard, que l'implémentation de l'heuristique déterminant le nombre de découpes à réaliser au niveau d'un nœud, varie grandement en fonction de l'algorithme utilisé. Néanmoins, le fonctionnement détaillé ci-dessus demeure identique.

HiCuts

HiCuts (Gupta et McKeown, 2000) est le premier d'une longue lignée d'algorithmes, adoptant le point de vue géométrique, et utilisant des arbres de décision pour classifier des paquets. Cet algorithme procède donc à un traitement préliminaire, visant à construire un arbre de décision, pierre angulaire pour ensuite classifier des paquets. HiCuts présente ainsi deux algorithmes, le premier pour construire l'arbre et le second pour le parcourir. Cet algorithme reprend les concepts présentés précédemment et propose de découper l'espace de recherche selon une dimension à la fois. Afin de mieux comprendre les différences entre HiCuts et les algorithmes détaillés ensuite dans ce document, on présentera ici, en détails, la structure d'un nœud utilisé par HiCuts.

A chaque nœud de l'arbre, noté v, est associé :

- Une "boite" *B(v)*, décrit par un *k-uplet* d'intervalles, où k représente le nombre de dimensions (champs) considérées.
- Une découpe *C(v)*. Celle-ci est définie par une dimension d selon laquelle les découpes sont effectuées, et par *np*, le nombre de partitions de l'espace associée à *B(v)*. Chacune des partitions effectuées définie un nœud enfant du nœud v.
- Un ensemble de règles couvert, ou contenu par le nœud, *R(v)*. Le nœud racine de l'arbre couvrant l'ensemble des règles contenues dans la table de classification. Soit *NumRules(v)* le nombre de règles contenues par *R(v)*. Il est à noter que la règle n'a pas besoin d'être contenue intégralement dans l'espace couvert par le nœud. Dès qu'une partie de la règle couvre l'espace délimité par v, alors celle-ci est comprise dans *R(v)*.

Il faut remarquer que d'autres informations peuvent être requises lors de l'implémentation logicielle afin de pouvoir réaliser le parcours de l'arbre, notamment un pointeur permettant de récupérer la position du nœud enfant auquel accéder. Ce détail est pour l'instant ignoré

dans le cadre de cette analyse théorique.

Construction de l'arbre

Comme mentionné dans la section 2.3.3, lors de la découpe de l'espace couvert par un nœud, la première étape consiste à sélectionner la dimension selon laquelle les découpes vont être effectuées. L'heuristique associée à la sélection de la dimension, peut être guidée par plusieurs facteurs selon les auteurs de HiCuts :

- Minimiser le nombre de règles ou portions de règles contenues dans chacune des sous-régions potentiellement découpées.
- Considérer le *Nombre de règles contenues par sous-région/Sm(c)* comme une distribution de probabilité avec *Np(c)* éléments et maximiser l'entropie de la distribution. Intuitivement, cette technique revient à sélectionner la dimension qui amène la distribution de règles la plus uniforme pour les différents nœuds provenant du même nœud parent.
- Choisir la dimension minimisant *Sm(c)* sur l'ensemble des dimensions
- Découper la dimension ayant le plus grand nombre d'éléments distincts parmi l'ensemble des dimensions

Remarque : Il est à noter que les auteurs de HiCuts, bien que mettant en avant plusieurs heuristiques, ne comparent pas les méthodes proposées, ni ne préconisent d'heuristiques en particulier. On remarquera que l'ensemble des méthodes présentées, à l'exception de la dernière nécessitent un traitement lourd, dans la mesure où il faut appliquer l'heuristique de découpe pour ensuite savoir si la dimension choisie était pertinente ou non. De fait, de nombreuses combinaisons doivent être testées avant d'identifier la dimension optimale, ce qui rend le prétraitement, c'est-à-dire la construction de l'arbre, longue. Le point relatif à la problématique du temps de la construction de l'arbre est abordé plus en détails dans le chapitre 7. L'étape suivante consiste à utiliser une heuristique pour déterminer le nombre optimal de découpes à faires au niveau du nœud. Une présentation en détail est faite de cette heuristique dans (Stimpfling *et al.*, 2013). De même, une implémentation possible de cette heuristique est présentée dans (Gupta et McKeown, 2000).

Optimisation introduites par HiCuts

Les deux heuristiques présentées permettent de construire l'arbre de décision, mais elles ne permettent pas de garantir que l'arbre construit est optimal. En effet, il se peut, tout d'abord, que certaines règles se chevauchent, ce qui peut entrainer une réplication importante des dites règles en raison d'un nombre élevé de découpes pour les séparer, générant tout autant de

nœud enfants couvrant les même règles, augmentant la profondeur de l'arbre et générant un nombre élevé de feuilles partageant les mêmes règles. À cet égard, lorsque les portions de règles se superposent, celle ayant la priorité la plus importante est conservée, afin d'éviter au maximum la superposition de règles, source majeur de dégradation de la performance. Par ailleurs, lorsque des noeuds partagent le même ensemble de règles, afin de limiter la taille de la structure de donnée générée, un système de pointeur est utilisé afin de pointer vers un seul noeud plutôt que de générer un très grand nombre de noeuds.

Résultats

Afin d'évaluer la performance de l'algorithme, les auteurs testent HiCuts sur des tables de routage (table de classification utilisant deux dimensions), comportant 20 000 entrées. La structure de donnée générée est relativement petite et est d'environ 1 Mo pour un arbre peu profond. Néanmoins, la performance se dégrade rapidement lorsque quatre dimensions sont utilisées au niveau des règles. Pour ce schéma, différentes tables de classifications ont été testées, comportant de 100 à 1733 règles. La taille de la structure de donnée générée varie ainsi entre 80 Ko et 1 Mo pour une table de classification contenant environ 1000 règles. Les variations de performance s'expliquent principalement par les différentes tables de classification utilisées. En effet, dépendamment du contexte où ces dernières sont utilisées, la complexité des règles varie de manière importante. Or, la complexité peut se traduire notamment par des règles se superposant ou ayant un schéma difficile à découper. De même, la profondeur de l'arbre généré augmente drastiquement avec le nombre de champs supplémentaires considérés puisque la profondeur maximale atteinte est de 12, générant près de 20 accès mémoire pour une implémentation logicielle sur un Pentium II. A cet égard, il est important de rappeler que l'augmentation de la profondeur de l'arbre génère davantage d'accès mémoire, ainsi qu'une structure de donnée plus grosse. Les auteurs mettent en avant la faible taille de structure de donnée générée, qui peut dont être contenue dans la cache L2 du processeur et par la même, suivant leur raisonnement, offrir des performances décentes.

Néanmoins, afin de moduler leurs propos, il est important de rappeler que le nombre d'entrées considérées pour leurs tests est relativement restreint et n'est pas en phase avec les contraintes présentées dans la littérature, présentant des tables de classification sur 5 champs, pour plus de 100000 règles. Par ailleurs, la taille par règles étant inférieure à 20 octets, une structure de 1 Mo pour 1000 règles représente un poids proche de 1000 octets par règles dans le pire des scénarios, ce qui illustre à quel point la structure de donnée, menant aux règles, est lourde. En réalité, considérant que la profondeur de l'arbre est égale à 12 dans la majorité des cas, on peut davantage penser que le problème de la taille de la structure est lié à la réplication de règles et que les optimisations introduites sont mésadaptées à un contexte où les règles se

chevauchent. Par ailleurs, la profondeur des arbres est liée directement à la seule dimension utilisée pour réaliser des découpes. Il est simple de réaliser que des découpes sur plusieurs dimensions à la fois permettent d'accélérer la convergence vers les feuilles. Néanmoins, dans un tel cas, davantage d'information doit être stockée. Il sera donc intéressant d'évaluer l'impact de la prise en charge de davantage de dimensions sur la profondeur de l'arbre versus la taille de l'arbre. Néanmoins, une telle comparaison est relativement difficile à effectuer puisque les caractéristiques des tables de classifications (nombre d'entrées, typage) utilisées pour les tests ne sont pas constantes.

HyperCuts

Les auteurs de HyperCuts (Singh *et al.*, 2003) mettent en avant que les algorithmes de classifications présents dans la littérature, basés sur des arbres de décision ou encore RFC, ne permettent pas de passer outre le compromis habituel entre consommation mémoire et temps de recherche. De nouvelles idées sont proposées afin d'améliorer de manière drastique la performance de HyperCuts par rapport à ses concurrents. Ainsi, deux idées intuitives sont introduites :

- Découpage multidimensionnel de l'espace permettant, en une découpe, de réaliser l'équivalent de plusieurs découpes unidimensionnelles

  Cette méthode vise à accélérer la convergence vers les feuilles et permettre de réduire la taille de la structure de donnée ainsi que le nombre d'accès mémoire nécessaires pour classer une règle.
- Déplacement des règles vers le haut de l'arbre de décision

Suite à des observations faites par les auteurs de cet algorithme, la prise en compte de règles comportant des entrées génériques pour certains champs entraine des problèmes de réplications importants au niveau des règles ; ces dernières se trouvant sur-découpées et présentes dans un très grand nombre de feuilles. Par conséquent, afin de limiter ce problème, l'ensemble des règles communes, se propageant sur plusieurs niveaux, sont placées dans un sous-arbre et les recherches se font dorénavant verticalement et horizontalement

Afin d'illustrer la première idée, les figures 2.12 et 2.11 présentent respectivement les découpes effectuées par HyperCuts et HiCuts. Pour chacune de deux figures, le même ensemble de règles est considéré, contenant 6 règles, s'échelonnant entre R_0 et R_5. De même, pour chacune des figures, on considère que la phase de construction de l'arbre a déjà été effectuée, selon les découpes illustrées par des traits hachurées de couleur verte et rouge. Par analogie aux autres exemples présentés dans ce mémoire, la valeur seuil utilisée ici est égale à 2. Un

arbre de décision a été construit pour chacun de ces deux algorithmes. Les rectangles de couleur orange sont associés à des noeuds, et l'indication "Champ 1" ou "Champ 2" est utilisée pour préciser selon quelle dimension les découpes ont été réalisée. Les rectangles de couleur bleue représentent des feuilles, et chacun de ces rectangles contient une liste de règle qui lui y est associée.

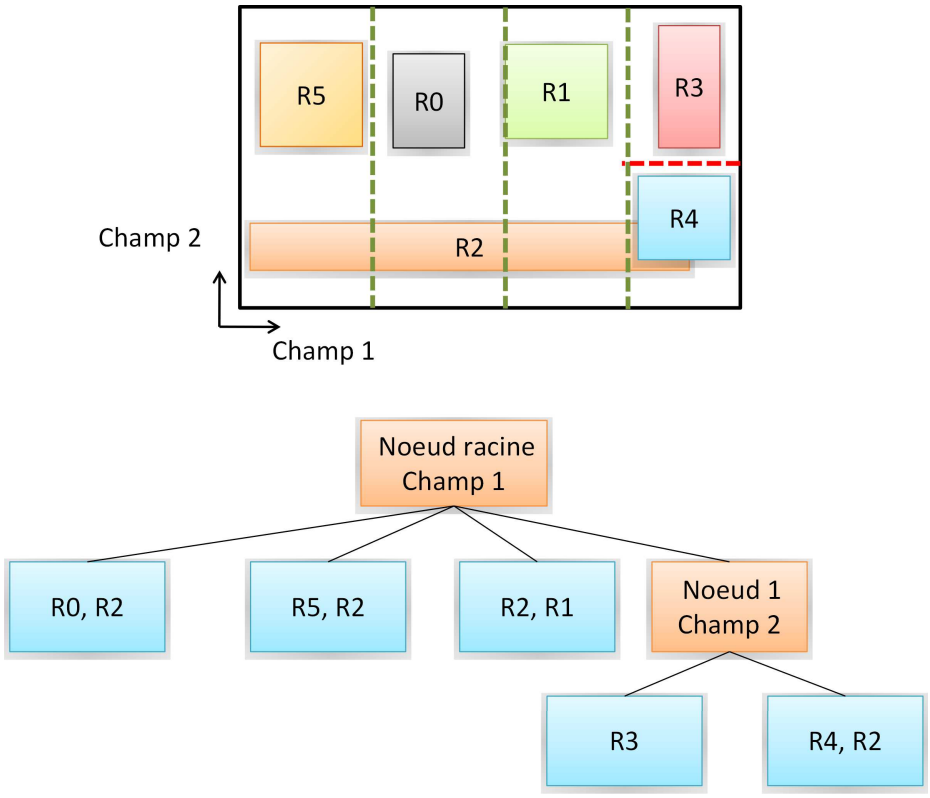


Figure 2.11 Découpes effectuées par HiCuts

On remarque aisément dans la figure 2.12 que les découpes multidimensionnelles introduites par HyperCuts -ici limitées à deux-, permettent de réduire la taille de l'arbre de manière importante et, dans le cas présenté, la taille des feuilles, par rapport à l'arbre construit par HiCuts représenté dans la figure 2.11.

Dans le cas de l'arbre créé par HyperCut, la diminution de la profondeur de l'arbre, ainsi que la réduction de la taille des feuilles permet d'accélérer la traversé de l'arbre, et donc le réduire le nombre d'accès mémoire à effectuer. Intuitivement, on peut prévoir une amélioration subséquente de la performance relative aux différentes métriques employées. Au-delà du nombre de dimensions pouvant être découpées simultanément, la structure du nœud demeure relativement proche de celle présentée dans HiCuts.

À chaque nœud contenu dans l'arbre de décision est associé

– Une région *R(v)* qu'il couvre et qui contient autant d'intervalles que de dimensions utilisées par les règles
– Un nombre de découpes *NC*, ainsi qu'une table contenant *NC* pointeurs
– Une liste de règles pouvant générer une comparaison positive avec le paquet entrant

Les nœuds utilisés par HyperCuts se distinguent de ceux utilisés par HiCuts au niveau du nombre de dimensions supporté et surtout par la possibilité pour un nœud de stocker une ou plusieurs règles, là où, dans HiCuts, seules les feuilles pouvaient contenir des règles.

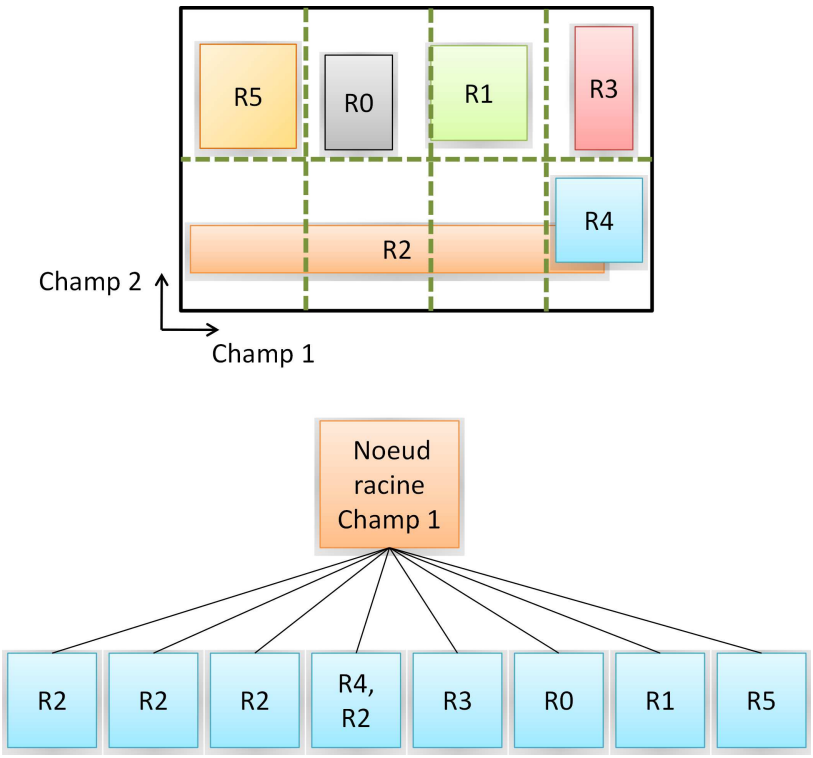


Figure 2.12 Découpes effectuées par HyperCuts

Construction de l'arbre

L'algorithme commence par séparer les règles en deux groupes : celles contenant des contraintes de type générique sur les champs adresses IP Source et Destination et toutes les autres règles restantes sont placées dans un autre ensemble. Pour chacun de ces deux ensembles, un arbre de décision est construit.

Sélection des dimensions La sélection des dimensions se base ici sur le nombre d'éléments uniques. Le principe est relativement simple et consiste à calculer, pour chacune des dimensions, le nombre d'éléments uniques contenus, puis effectuer la moyenne sur l'ensemble

des dimensions utilisées et de sélectionner les dimensions ayant un nombre d'éléments uniques supérieur à la moyenne.

Sélection du nombre de découpes Les découpes pouvant être effectuées sur plusieurs dimensions à la fois, il est dorénavant plus compliqué de trouver la meilleure combinaison de découpes globale. En effet, il s'agit d'un problème d'optimisation multi contraintes. Le nombre de découpes effectuées, *NC*, est défini comme le produit du nombre de découpes pour chacune des dimensions utilisées. *NC* est borné par $f(N) = spfac \cdot \sqrt[2]{N}$, avec N règles couvertes par le nœud et *spfac*, un facteur d'espace permettant de faire varier le compromis entre consommation de mémoire et temps de recherche. Considérant qu'il est impossible de tester l'ensemble des combinaisons de découpes, la stratégie adoptée consiste à déterminer séparément, pour chacune des dimensions, l'optimum local du nombre de découpes et, ensuite, à déterminer la meilleure combinaison centrée autour de ces valeurs. Afin d'identifier le nombre de découpes pour chacune des dimensions, la moyenne du nombre de règles des nœuds enfants, le nombre maximum de règles dans chacun des nœuds enfants ainsi que le nombre de nœuds enfants vides sont considérés. Là encore, le processus est itératif et, à chaque itération, le nombre de découpes est multiplié par deux. On remarquera que les précisions apportées par les auteurs restent relativement vagues et libres d'interprétation de telle sorte que l'implémentation d'HyperCuts basée sur les seuls détails fournis dans l'article est fortement liée à la personne l'implémentant. Une étape importante non précisée est : comment, à partir du résultat d'optimums locaux, le choix de la découpe globale est effectué ? De même, dans quelle proportion les différentes informations rassemblées sont-elles utilisées pour déterminer l'optimum local ? En effet, l'étape de découpage est l'étape critique qui conditionne la performance de l'algorithme.

Optimisations Lors de la construction d'un nœud, quatre différentes optimisations supplémentaires sont introduites par HyperCuts afin de réduire l'espace mémoire utilisé par la structure de donnée : fusion des nœuds partageant le même ensemble de règles, suppression des règles avec une faible priorité pour un cas particulier, réduction de la taille des régions, déplacement des ensembles communs de règles vers le haut. Il est à noter que les deux premières optimisations sont identiques à celles présentées dans (Gupta et McKeown, 2000) et, par conséquent ne sont pas re-détaillées et l'on réfère le lecteur à la Section 5.3.2 pour avoir un détail de ces deux techniques.

La réduction de la taille des régions, ou "Region Compaction", consiste à réduire la taille des régions de l'espace contenant des règles ne couvrant pas l'intégralité de l'espace. Dans un tel cas, les bornes du nœud sont réduites afin d'encadrer les règles contenues et éviter

d'inclure des zones vides, que cela soit proche des bornes inférieures ou supérieures du nœud. L'heuristique "Pushing Common Rule Subsets Upwards" est appliquée lorsque l'ensemble des nœuds enfants d'un même nœud partagent un sous-ensemble de règles couvert par le nœud parent. Dans un tel cas, plutôt que de générer de multiples nœud enfants découpant un même ensemble de règles, celles-ci sont associées au nœud parent et une feuille "horizontale" est ainsi construite. En effet, lorsque des règles sont partagées par l'ensemble des nœuds enfants, cela signifie qu'il s'agit de règles larges et donc complexes à découper. Par conséquent, en isolant ces règles au niveau du nœud, d'éventuels problèmes de réplication de règles sont évités au prix d'une traversée de l'arbre à la fois horizontale et verticale. A cet égard, la structure d'un nœud est désormais modifiée et elle permet de spécifier si le nœud contient des règles à parcourir horizontalement ou non. Il est à remarquer que, contrairement à HiCuts ou les optimisations introduites étaient effectuées après la construction de l'arbre, HyperCuts procèdes aux différentes optimisations directement pendant la construction de l'arbre, ce qui permet de faire levier sur les avantages amenées par les optimisations. Par exemple, "Pushing Common Rule Subset upward" permet d'éliminer du processus de découpe un ensemble de règles pouvant inutilement augmenter la profondeur de l'arbre. Le processus de parcours de l'arbre est identique à HiCuts, à la différence près que les nœuds peuvent contenir des règles et donc que le parcours (classification d'un paquet) se fait en largeur et en profondeur.

Performance

L'algorithme HyperCuts a été évalué à la fois sur des tables de classification utilisées dans un contexte de "coupe-feu", que sur des tables utilisées dans des routeurs de cœur de réseau. Les règles proviennent directement de certains fournisseurs d'accès à Internet et leur nombre varie entre 85 et 2800 et elles utilisent les cinq champs classiques (la composition est détaillée dans le chapitre 1 où l'on réfère au cinq tuplets). Les résultats présentés concernent le pire des scénarios, soit pour des paquets de taille minimale. HyperCuts permet de diminuer de manière considérable la taille de la structure de données par rapport à HiCuts, d'un facteur proche de 20 dans les meilleurs cas. Néanmoins, ces améliorations sont davantage prononcées pour des tables contenant peu de règles. Par ailleurs, la taille de la structure de donnée générée demeure importante et atteint plus de 16 Mo pour seulement 2799 règles, pour des règles utilisées dans un routeur de cœur de réseau. Dans le cas de règles utilisées dans un contexte de "coupe-feu", HyperCuts réduit la consommation de mémoire par rapport à HiCuts de manière importante, de l'ordre d'un facteur cinq. Là encore, la consommation de mémoire demeure importante et plus de 9 Mo sont consommés pour stocker la structure associée à 279 règles. Concernant, le nombre d'accès mémoire, l'amélioration est plus limitée, HyperCuts diminue en effet, dans le cas du "coupe-feu", le nombre d'accès mémoire par trois dans le

meilleur des cas, par rapport à HiCuts. Dans le cas de tables utilisées dans des routeurs de cœurs de réseau, l'amélioration est limitée à un facteur deux en moyenne. Ainsi, HyperCuts améliore sensiblement la performance par rapport à HiCuts en termes de taille de structure de donnée générée et de nombres d'accès mémoire nécessaires pour classifier un paquet. Néanmoins, la consommation mémoire demeure importante car HyperCuts utilise seulement deux arbres ; le premier contenant des règles ayant des contraintes génériques pour les deux premiers champs (IP Source et IP Destination) et l'autre arbre contenant toutes les autres règles. Dès lors, des règles aux caractéristiques différentes se trouvent mélangées ; des règles couvrant une large portion de l'espace peuvent côtoyer des règles délimitant un petit espace.

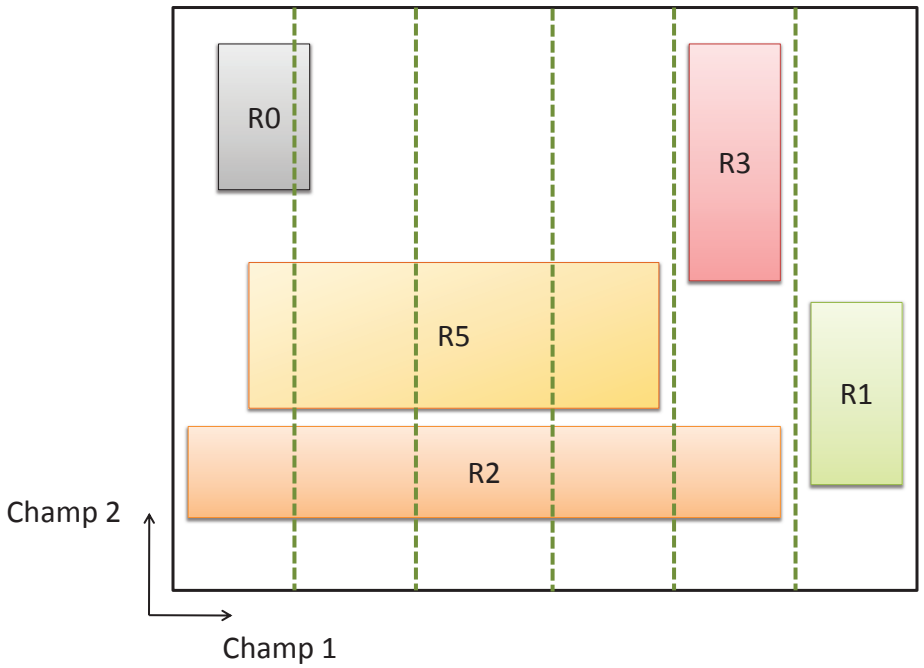


Figure 2.13 Sur-découpage lors d'un mélange de règles de tailles différentes

Or, de tels scénarios amènent un problème lors de la phase de découpe. En effet, afin de séparer les règles, comme illustré dans le schéma 2.13, l'heuristique calculant le nombre de découpes à effectuer va choisir de réaliser un nombre important de découpes afin de pouvoir séparer les "petites" règles. La fine granularité utilisée sur-découpe donc les grosses règles qui se retrouvent ainsi dans plusieurs nœuds enfants ou feuilles ce qui tend à augmenter sensiblement le facteur de réplication et donc la taille de la structure de donnée. Par ailleurs, dans la cette même figure, on constate que l'optimisation "Pushing Common Rule Subsets Upwards" n'a aucune utilité car les règles R_5 et R_2 ne sont pas partagées par l'ensemble des zones découpées. Ainsi, cette optimisation n'est pertinente que dans un nombre restreint de

situation, ce qui explique la consommation élevée de mémoire pour l'algorithme HyperCuts. Par ailleurs, l'ajout potentiel de règles au niveau d'une feuille permet, d'un côté, de diminuer, dans certains cas, la réplication de règles mais impacte négativement le nombre d'accès mémoire à effectuer pour classifier un paquet puisque la traversée de l'arbre se fait maintenant en profondeur et en largeur lorsqu'un nœud possède des règles.

EffiCuts

Conscient des problèmes majeurs de réplications intrinsèques à HiCuts et HyperCuts, l'algorithme EffiCuts a été proposé afin d'adresser l'ensemble de ces problèmes et de pouvoir offrir une solution permettant de supporter un nombre élevé de règles peu importe la complexité de celles-ci. L'algorithme proposé (Vamanan *et al.*, 2010) vise à réduire de manière drastique la consommation mémoire en éliminant l'impact du chevauchement de règles ayant des caractéristiques de tailles très variées et, d'autre part, en proposant une solution aux variations importantes de densité de règles par découpe. En effet, certaines découpes contiennent un nombre important de règles alors que d'autres contiennent très peu de règles, ce qui rend les découpes peu efficaces. Ainsi, EffiCuts introduit quatre optimisations permettant de compenser les différents problèmes mentionnés :

- **Separable Trees**
  Les règles sont triées dans différents arbres selon des caractéristiques de taille des intervalles couverts pour chacun des champs
- **Selective Tree Merging**
  Cette méthode vise à fusionner certains arbres entre eux afin de diminuer l'impact de l'augmentation du nombre d'accès mémoire lié à l'introduction de Separable Trees.
- **Equi-dense cut**
  Mécanisme visant à fusionner certains nœud enfants ensemble afin d'avoir moins de variation de densité de règles entre les nœuds enfants associés à un même nœud parent
- **Node colocation**
  Méthode visant à réduire le nombre d'accès mémoire en supprimant le recours à un pointeur pour identifier la position du prochain nœud enfant à traverser

Les quatre optimisations présentées ci-dessus sont décrites en détails dans (Stimpfling *et al.*, 2013). Ainsi, afin d'éviter un doublon dans les explications, dans cette partie, on fera abstraction d'une explication détaillée concernant ces quatres optimisations.

Structure d'un noeud

Contrairement à HyperCuts, où un nœud peut se voir associer des règles à la manière d'une feuille, les nœuds utilisés dans EffiCuts ne peuvent contenir de règles. La structure comprend donc un entête précisant le type de nœud : nœud avec des découpes de tailles égales, nœud avec des découpes à densité égale ou feuille. Ensuite, il est nécessaire d'avoir un champ permettant de décrire la zone de l'espace couverte par le nœud (tout comme pour les nœuds de HiCuts et HyperCuts), ainsi que l'information nécessaire pour décrire les découpes effectuées : dimensions utilisées et nombre. Enfin, l'introduction de la méthode de découpe à densité égale nécessite l'ajout d'un champ permettant de savoir quels nœuds ont été fusionnés entre eux. Par ailleurs, un pointeur de base est nécessaire afin de connaitre l'adresse du prochain nœud enfant à parcourir.

EffiCuts se distingue par rapport à HiCuts et HyperCuts en ne stockant pas de pointeur au niveau des feuilles mais directement la règle. La quantité d'information liée à une règle est sensiblement plus qu'un simple pointeur mais, contrairement à HiCuts et HyperCuts où le taux de réplication est très élevé, dans le cas d'EffiCuts, il est proche de 1. Par conséquent, il n'est plus nécessaire d'utiliser des pointeurs vers les règles, ce qui permet de gagner plusieurs accès mémoire lors du parcours de la structure de donnée.

Construction des arbres

Contrairement à HiCuts où seul un arbre de décision est construit et à HyperCuts où deux arbres sont construits, EffiCuts, avant toute construction d'arbres, utilise les deux heuristiques "Separable Trees" ainsi que "Selective Tree Merging" pour classer les règles en différentes catégories, fusionner certaines catégories entre elles et ensuite construire un arbre par catégorie crée.

La construction de l'arbre est identique à la construction utilisée dans l'ensemble des algorithmes utilisant des arbres de décision. Ainsi, une heuristique est utilisée pour déterminer la ou les dimensions utilisées, ainsi que le nombre de découpes à effectuer selon cette ou ces dimensions. Des optimisations locales sont effectuées au niveau de chacun des nœuds, notamment "Node Merging", ainsi que la réduction de la zone de l'espace couverte. Les conditions d'arrêt sont les même que pour les autres algorithmes.

Le parcours de l'arbre est assez semblable à celui employé par les autres algorithmes. Une différence majeure vient du fait que les découpes au niveau d'un nœud ne sont pas toutes de tailles équivalentes, suite à l'introduction du concept de "Equi-dense cut". Dans un tel cas, la première partie consiste à faire fi de la découpe inégale et de trouver le nœud enfant "normal" en déterminant la position du paquet par rapport aux différentes découpes. Ensuite,

la position est utilisée et comparée à une table contenant la position des nœuds fusionnés entre eux et elle permet donc de savoir à quel nœud enfant il faut accéder pour tenir compte de la fusion. Le restant du parcours se déroule de manière identique à HiCuts ou HyperCuts si l'on ne tient pas compte du mécanisme "Règles déplacées vers le haut de l'arbre de décision". Lorsqu'une feuille est atteinte, là où HiCuts et HyperCuts se basaient sur des pointeurs pour extraire la ou les règles à comparer, EffiCuts a des feuilles contenant directement les règles de telle sorte que la comparaison est immédiate après avoir lu les informations contenues dans la feuille. EffiCuts utilise plusieurs arbres et, à cet égard, il est nécessaire de parcourir chacun d'eux afin de pouvoir identifier les règles correspondantes au paquet entrant.

Évaluation

Pour évaluer les performances de cet algorithme, les auteurs utilisent un générateur synthétique de règles, ClassBench (Taylor et Turner, 2007), reproduisant des patrons issus de tables de classification réelles basées sur 5 dimensions. Trois types de tables de classification sont utilisés à cet égard : Firewall, IP Chain et Access List. Pour chacun de ces trois scénarios, trois sous-scénarios sont considérés, chacun utilisant respectivement 1000, 10 000 et enfin 100 000 règles.

Le premier constat à faire concerne le facteur de réplication. Ce dernier étant très faible, il permet d'expliquer la faible taille de la structure de donnée générée, qui ne dépasse pas 13 Mo pour 100 000 règles considérées. La réduction de la consommation mémoire atteint ainsi en moyenne un facteur de 57 fois par rapport à HyperCuts.

Néanmoins, la réduction drastique de la taille de la structure de donnée se fait au dépend d'une augmentation du nombre d'accès mémoire. En effet, l'augmentation du nombre d'arbres à parcourir pour classifier un paquet explique largement ce phénomène. On retrouve ainsi le compromis intrinsèque à cette famille d'algorithme, à savoir, consommation mémoire versus temps de recherche. L'augmentation est en moyenne de 50% et, dans le pire des cas, atteint un facteur 3.

Ainsi, EffiCuts rend possible l'utilisation de grandes tables de classification, comprenant 100 000 règles, ce qui est impossible avec HyperCuts, mais au détriment d'une augmentation du nombre d'accès mémoire. Une des causes majeures d'un grand nombre d'accès mémoire est la recherche linéaire exécutée au niveau d'une feuille, où l'ensemble des règles doit être parcourue, alors que seule une minorité correspond au paquet entrant. Un nombre substantiel d'accès mémoire est effectué pour accéder à de l'information non pertinente.

Il faut, par ailleurs, constater que l'ensemble des résultats associées aux algorithmes sont des résultats provenant de simulations logicielles et ne résultent en aucun cas d'une quelconque implémentation matérielle. Dès lors, les contraintes matérielles sont différentes des contraintes

logicielles et une architecture adaptée permet d'offrir une bande passante élevée malgré un nombre d'accès mémoire important à réaliser, alors que la taille de la structure de données est beaucoup plus difficile à bien supporter par du matériel.

2.4 Implémentations matérielles

Avant de présenter les différentes implémentations matérielles réalisées, il est important de remettre en contexte les travaux effectués ici ; le but est de concevoir un algorithme pour un contexte particulier : celui d'une perspective d'implémentation matérielle. A cet égard, l'algorithme doit pouvoir être implémenté en matériel et ce, en offrant une performance décente. Ainsi, l'algorithme AcceCuts, qui est l'objet de ce mémoire, a été optimisé selon des critères primaires de performance matérielle. L'algorithme doit ainsi permettre un traitement parallèle et pipelinable. Par ailleurs, la structure de données générée doit être compacte afin de minimiser la quantité de mémoire à intégrer à la plateforme sur laquelle l'algorithme sera exécuté. Également, afin de minimiser la latence du processus de classification de paquet, le nombre d'accès mémoire doit être minimisé. Le but de la section suivante est surtout de démontrer que les algorithmes basés sur des arbres de décision offrent, d'une part, des performances décentes sur différents types de plateformes matérielles et, d'autre part, que la structure de données utilisée par ces algorithmes permet un traitement parallèle, et pipelinable, ce qui permet de justifier la performance atteinte et renforce l'attrait que l'on porte à ces algorithmes. La revue de littérature concernant les implémentations matérielles de tels algorithme étant déjà développée dans le chapitre 5, l'analyse qui y développée n'est pas dédoublée ici et l'on réfère le lecteur à cette même section pour une étude des différentes implémentations réalisées.

2.5 Mises à jour

Les algorithmes héritant de l'approche introduite pas HiCuts souffrent d'un potentiel problème de mise à jour. Ainsi, jusqu'à présent aucune méthode de mise à jour sérieuse n'a été proposé dans la littérature (Gupta et McKeown, 2000; Singh *et al.*, 2003; Vamanan *et al.*, 2010; Jiang et Prasanna, 2012). Dépendamment, les mises à jour sont soient présentées comme supportées, mais aucun détail précis n'est fourni, soit il est convenu que les mises à jour ne sont pas possibles pour ce type d'algorithme sans reconstruire intégralement le ou les arbres de décision.

Si aucune méthode de mise à jour n'est viable, alors par défaut, il est nécessaire de reconstruire intégralement la structure de donnée. Par conséquent, l'insertion ou la surpression de règles demeure un problème ouvert, qui plus est, préoccupant dans le cadre du SDN. En

effet, la gestion du réseau se fait de manière dynamique, ce qui implique des modifications en temps réel des tables de classification, et donc des structures de données utilisées pour classifier les paquets entrant.

2.6 Conclusion partielle

Cette revue de littérature a permis d'établir qu'il existe des algorithmes basés sur des arbres de décision capables d'offrir des performances raisonnables dans un contexte de 5-uplets. Néanmoins, un des problèmes majeurs relatifs à la performance de tels algorithmes est lié à la non uniformisation des méthodes utilisées pour établir leur performance. Ainsi, chacun des algorithmes présentés ici utilise soit des tables de classification provenant de fournisseurs d'accès à Internet ou provenant de générateur synthétiques.

Par ailleurs, chacun des trois articles étudiés en détails ici présente d'importantes lacunes ; dans le premier, aucune mention n'est faite de la composition exacte des différents champs composant le nœud, concernant HyperCuts, un floue entoure l'heuristique utilisée pour sélectionner le nombre de découpes à réaliser au niveau d'un nœud (plusieurs méthodes proposées mais aucune n'est spécifiée comme étant utilisée), et enfin, concernant EffiCuts, les détails concernant la structure d'un nœud, -outre le fait d'être discutables-, ne sont tout simplement pas justifiés.

CHAPITRE 3

DÉMARCHE DE L'ENSEMBLE DU TRAVAIL DE RECHERCHE

Les algorithmes présentés dans la revue de littérature ont été conçus et évalués pour traiter des règles basées sur 5 champs. Néanmoins, comme présenté dans la chapitre 1, le contexte du SDN introduit une liberté et souplesse de traitement influençant à son tour le domaine de la classification de paquets. Ainsi, là où il était courant de classifier les paquets simplement sur 5 champs, dorénavant cette caractéristique devient variable et, selon les spécifications de protocoles utilisés pour le SDN, jusqu'à 12 champs - et plus - peuvent être utilisées simultanément pour classifier un paquet. Le but de ce mémoire est de concevoir un algorithme optimisé pour le contexte du SDN, permettant ensuite d'être implémenté en matériel. Néanmoins, avant toute conception d'algorithme, la démarche retenue ici consiste à évaluer la performance ainsi que les points forts et faibles d'un algorithme à l'état de l'art dans le contexte du SDN. La phase d'analyse permettra ensuite de déterminer s'il est nécessaire de concevoir un algorithme de zéro ou si certaines idées ou approches demeurent pertinentes.

Dans la littérature, il est important de souligner que certains travaux ont réalisés des évaluations sur un nombre de champs plus élevés, dépendamment, 11 ou 12 (Fong *et al.*, 2012; Jiang et Prasanna, 2012) champs étaient considérés. Néanmoins, aucun de ces travaux ne contient une analyse précise de la performance offerte par les algorithmes. Dès lors, il est difficile de comprendre d'où proviennent les défauts de ces algorithmes ; *s'agit-il d'un problème intrinsèque ne pouvant être résolue, ou au contraire l'algorithme souffre-t-il de problèmes liés à une mauvaise conception ?* Ainsi, la première réalisation ne peut s'appuyer sur les travaux déjà contenus dans la littérature, d'autant plus que ceux-ci portent sur des algorithmes antérieurs à l'état de l'art. Par conséquent, l'algorithme EffiCuts a été choisi en raison du degré de performance élevé offert dans le contexte du 5-tuplets et il sera à la base de l'analyse réalisée et présentée ci-dessous. Trois optimisations ont été aussi introduites, afin d'évaluer le potentiel d'optimisation de l'algorithme, et permettent de décider si cet algorithme peut ou non, servir de base à un algorithme conçu spécifiquement pour le contexte du SDN. Avant de présenter plus en détails l'analyse effectuée d'EffiCuts dans le contexte du SDN, présentons le travail réalisé afin d'observer et d'analyser la performance du dit algorithme dans le contexte du SDN.

L'implémentation d'EffiCuts utilisé a été obtenue auprès des auteurs de cet algorithme. Comme précisé dans la revue de littérature, cet algorithme a été conçu pour fonctionner avec

des règles utilisant 5 champs, ce qui le rend intrinsèquement incompatible avec le contexte du SDN. À cet égard, le premier travail consiste à adapter EffiCuts, afin qu'il puisse supporter des règles utilisant 12 champs. En effet, l'algorithme EffiCuts tel qu'il a été reçu, n'est pas codé de manière générique et l'ensemble du traitement ne permet pas, par des changements mineurs, de l'adapter au contexte de 12 champs. Des changements doivent être apportés au niveau du code pour :

- La séparation des règles en différentes catégories (*Separable Trees*)
- La fusion de certaines catégories entre elles
- La construction des arbres associés à chacune des catégories
- La récupération des statistiques associées aux arbres créés

Par ailleurs, afin de pouvoir évaluer la performance de l'algorithme EffiCuts dans plusieurs contextes, d'une part, un benchmark a été conçu (voir Section Methodology de 4.5.1), faisant levier à la fois sur ClassBench (Taylor et Turner, 2007) et sur FRuG (Ganegedara *et al.*, 2010). Des scripts de simulations ont été mis en place afin d'automatiser les nombreuses simulations à réaliser. Le travail décrits ci-dessous et présentés à la conférence HPPN 2013, et l'article est disponible sur `<http://dl.acm.org/citation.cfm?id=2465841>`.

Ce premier travail vise à établir certaines limitations d'EffiCuts dans le contexte de SDN, et de proposer en conséquent différentes optimisations. Néanmoins, le but de ce travail de recherche est de proposer un nouvel algorithme. À cet égard, un second travail présenté dans le chapitre 5, vise à procéder à une analyse en profondeur des causes limitant la performance de l'algorithme EffiCuts dans le contexte de SDN. Cette seconde phase d'analyse sert de base pour repenser la manière dont les arbres de décisions sont construits et parcourus. Plusieurs contributions sont ainsi effectuées et constituent le coeur de l'algorithme AcceCuts.

Afin de valider la performance de l'algorithme proposé, une évaluation théorique, ainsi qu'expérimentale est proposée.

L'algorithme AcceCuts est présenté dans le chapitre 5, et a été soumis dans le journal *IEEE/ACM Transactions on Networking*.

CHAPITRE 4

ARTICLE 1 : OPTIMAL PACKET CLASSIFICATION APPLICABLE TO THE OPENFLOW CONTEXT

Thibaut Stimpfling
École Polytechnique de Montréal
Montréal, Canada
thibaut.stimpfling@polymtl.ca

Yvon Savaria
École Polytechnique de Montréal
Montréal, Canada
yvon.savaria@polymtl.ca

Normand Bélanger
École Polytechnique de Montréal
Montréal, Canada
normand.belanger@polymtl.ca

André Béliveau
Ericsson Montréal
Montréal, Canada
andre.beliveau@ericsson.com

Omar Cherkaoui
Department of Computer Science
Université du Québec à Montréal
Montréal, Canada
cherkaoui.omar@uqam

Cet article a été soumis le 21 avril 2013 dans le cadre de la conférence *High Performance and Programmable Networking (HPPN)* 2013, et publié le 26 juin 2013.

4.1 Abstract

Packet Classification remains a hot research topic, as it is a fundamental function in telecommunication networks, which are now facing new challenges. Due to the emergence of new standards such as OpenFlow, packet classification algorithms have to be reconsidered to support effectively classification over more than 5 fields. In this paper, we analyze the performance offered by EffiCuts in the context of OpenFlow. We extended the EffiCuts algorithm according to OpenFlow's context by proposing three improvements : optimization of the leaf data set size, enhancements to the heuristic used to compute the number of cuts, and utilization of an adaptive grouping factor. These extensions provide gains in many contexts but they were tailored for the OpenFlow context. When used in this context, it is shown using suitable benchmarks that they allow reducing the number of memory accesses by a factor of 2 on average, while decreasing the size of the data structure by about 35

4.2 Introduction

Packet classification is used in a wide range of network equipment and needs. Examples of needs include quality of service (QoS), load balancing, security, monitoring, and traffic analysis. With ever-increasing link speed, it is necessary to continue improving classification algorithms in order to increase performance to match requirements. Packet classification consists of comparing fields from the packets to rules, and to find the rules that apply to any given packet. Several previous contributions focused on rules relating to only five fields, and proposed algorithms and methods that apply to that context (Gupta et McKeown, 2000, 2001; Singh *et al.*, 2003; Taylor, 2005; Dharmapurikar *et al.*, 2006; Vamanan *et al.*, 2010; Yeim-Kuan et Chao-Yen, 2012). Several classes of algorithms and approaches have been explored, from which two perform well : decomposition based algorithms and decision-tree based algorithms. Currently, tree-based algorithms perform better. Several tree-based algorithms have been proposed, like HiCuts (Gupta et McKeown, 2000), HyperCuts (Singh *et al.*, 2003), and EffiCuts (Vamanan *et al.*, 2010). All these algorithms divide the search tree into sub-spaces iteratively until each sub-space contains a number of rules that is below a given threshold.

HiCuts was the first algorithm to use a decision tree in which each node represents a sub-space and the leaves contain the rules pertaining to each of the sub-sets. Existing tree-based algorithms have been proposed and validated for the 5-tuple context (IP source address, IP destination address, protocol, source port, and destination port). Considering that the requirements evolve in terms of granularity of control (Vaish *et al.*, 2011), it is now important to adapt these algorithms to new types of network protocols such as OpenFlow (Open Networking Foundation, 2009), which have much more stringent classification requirements.

The OpenFlow specification requires that the classification must be done according to at least twelve fields from version 1.0.0 (Open Networking Foundation, 2009), which drastically increases the complexity of classification requirements. To the best of the authors' knowledge, no research has been published about optimizing classification algorithms for more than five fields, such as in the OpenFlow context.

Although EffiCuts reduces significantly rule replication when compared to the other known algorithms (Vamanan *et al.*, 2010), it has its drawbacks. First, it increases the average number of memory accesses needed to identify the relevant rules. Also, it does not take into account the table specifics in order to optimize performance. Finally, EffiCuts was designed and optimized for the 5-tuple scenario. In the OpenFlow context, rules are based on at least twelve fields (v1.0.0 and later) (Open Networking Foundation, 2009), which implies that rules

be represented using at least 41 bytes of data (see Section Methodology 4.5.1), while the 5-tuple scenario requires only 13 bytes per rule (Vamanan *et al.*, 2010). This three-fold increase can severely limit performance, and it must be addressed.

The goal of this paper is to assess the suitability of EffiCuts to the OpenFlow context and to modify and extend it to improve its performance in terms of memory space used and average number of memory access per packet identification. Note that this work is a first (optimization) step toward an efficient hardware implementation of a classification module. Toward that goal, a set of solutions are proposed. First, an adaptive compression factor is used to limit the number of categories created during the initial sort in order to limit the number of memory accesses required. Also, adopting the heuristic used to perform cuts in HiCuts compensates the negative impact of an overly large number of cuts performed in EffiCuts (which increases considerably the number of nodes, thus the amount of memory used). We also propose to tune the size of tree leaves to limit the number of memory accesses (at a small cost in memory usage).

This paper is organized as follows. A description of the weaknesses of known algorithms is presented in Section Related work along with the constraints of the current context. Section Extensions describes the proposed modifications to EffiCuts. Section Results presents results of the comparison between the optimized algorithm and EffiCuts. Finally, Section Conclusion draws conclusions from this work.

4.3 Related Work

4.3.1 Existing Algorithms

The first proposed tree-based packet-classification algorithm, HiCuts, generates a lot of rule replications as it creates a single decision tree, thus mixing together rules with very significant differences in size (which causes a lot of superposition). On the other hand, it was devised to handle fairly small tables, so it is not surprising that nothing was done to improve this aspect.

HyperCuts was proposed as an evolution to HiCuts, which aims at improving the convergence rate of the classification (thus, minimizing tree depth) while limiting the data structure size. To achieve this, the algorithm is based on multi-dimensional cuts and it includes techniques to minimize replication. These techniques produce better performance in terms of number of memory accesses, but scalability is poor (Singh *et al.*, 2003).

EffiCuts aims mainly at striking the best compromise between the average number of memory accesses and the data set size for the 5-tuple context. The authors (Vamanan *et al.*, 2010) designed this algorithm based on two key observations. The first one is that there

is a lot of overlap between rules in a classification table. Also, the variation in size of rule overlap is high, leading to a high degree of rule replication caused by thin cuts. EffiCuts addresses this issue by partitioning the rule set and binning rules with different size patterns in different subsets. Each of these subsets is then associated with a dedicated decision tree. This method is called separable trees. More details about this method are given in Section Adaptive grouping factor. However, the introduction of multiple trees adds extra memory accesses which decrease throughput. This problem is solved in EffiCuts with selective tree merging. This method aims at merging selectively separable trees, mixing rules that are either small or large in at most one dimension.

The second observation made in (Vamanan *et al.*, 2010) concerns the variation of the classifier's rule space density. HiCuts (Gupta et McKeown, 2000) and HyperCuts (Singh *et al.*, 2003) cut the space covered by a node equally between each child node. This process is named equi-sized cuts and a node that uses equi-sized cuts is therefore called an equi-sized node. Using equi-sized cuts for separating dense parts and empty zones result in ineffective nodes containing only a few rules. Thereby, it will unnecessarily increase the size of the decision tree. In order to address this problem, EffiCuts merges such nodes together. This method introduces a new type of nodes called equi-dense nodes. Therefore, EffiCuts uses two kinds of nodes : equi-dense nodes when dealing with fused nodes, and equi-sized nodes when the rule space size can advantageously be divided equally.

Additionally, EffiCuts introduces different optimization techniques like separable trees, selective tree merging and node collocation. Separable trees and selective tree merging were introduced to address the issue of rule size variation. Node collocation was proposed in order to reduce the overall number of memory accesses. It thus reduces considerably the memory usage compared to HyperCuts while having a low replication factor. On the other hand, these optimizations tend to increase the average number of memory accesses. A packet is classified by traversing each decision tree. For each node of a tree, the position of the packet relative to the sub space region associated with the node must be computed. The next step is to select the next child node to visit. For this, the relative position of the packet is converted into an offset. Equi-sized nodes equally cut the space, so the offset value is used to access directly the next child node. In the case where an equi-dense node is reached, the index value computed has to be next compared with an interval array. This array stores the boundary of each child region.

This process is repeated until a leaf is reached. Then, the packet is compared with every rule held in the leaf. In (Vaish *et al.*, 2011), the authors focus on an algorithmic evaluation, i.e. without considering any hardware implementation.

However, it has been shown in (Vaish *et al.*, 2011) that EffiCuts is implementable in

hardware and offers a high degree of performance. This implementation has been made on the PLUG architecture and it achieves a throughput of 142 million packets per second.

It was also observed that EffiCuts offers a good scalability (Vamanan *et al.*, 2010), and it seems to offer the best potential to perform well when a high number of fields is considered, while remaining implementable in hardware.

4.3.2 Context

The algorithms described above were devised for the 5-tuple context. The context explored here requires a 12-field classification. Later versions of OpenFlow take into account even more fields, but we consider that bridging the large gap between the classical 5-tuple fields and the 12-tuple fields used in version v1.0.0 is the most important aspect of the research that needs to be done. Thus, the algorithms proposed in the literature should be revisited to evaluate and improve their performance in this context. In this paper, we focus on characterizing and improving EffiCuts.

The first method that EffiCuts uses that requires attention is the separable trees method (details are given in Section Adaptive Grouping Factor). The goal is to minimize overlap between rules by grouping them into different trees according to their sizes. In the 5-tuple case, up to 32 categories (i.e. trees) can be created. On the other hand, the 12-tuple case can generate theoretically up to 4096 categories. Knowing that each tree must be visited to find the right rules, this is obviously very costly for the 12-tuple case due to the excessively large number of memory accesses required. Using the selective tree merging implemented in EffiCuts and extending it to 12 fields did not reduce experimentally the resulting number of memory accesses to an acceptable level, as in some cases, up to 800 memory accesses were needed 4.2. Indeed, in considering the perspective of a future high performance hardware implementation, each tree traversal would be done independently to reach the highest rate of packet processing rate, and therefore could ultimately require as many tree traversal engines as the number of trees generated. (Reusing the engines is of course possible but would increase processing time). This can be very costly. Contrary to the number of memory accesses, the data set size is fairly constant with respect to the number of packet fields used in the classification, because it is mainly a function of the level of overlap between rules with varying size. This is governed by the environment and thus cannot be optimized.

Recall that our goal is to optimize EffiCuts in a perspective of implementing it in hardware while considering a high number of fields processed, such as in the OpenFlow context. Considering this context, the main challenge is to allow a high degree of parallelization of the data structure while minimizing the average number of memory accesses in order to provide an acceptable performance level. This should preferably not be done at the expense of a

significant increase in data set size.

4.4 Extensions

Three extensions to EffiCuts are proposed in this paper. When applied together, they provide gains in every considered context, including the 5-tuple context. Nevertheless these extensions are proposed to give the highest observed performance gains when used in the OpenFlow context.

In order to clearly identify which aspects needed to be improved, we did an experimental analysis based on simulations. Due to space constraints, it is not possible to give all details of this analysis. However, this analysis was done accordingly to the methodology presented in Section Methodology.

4.4.1 Cutting heuristic

Upon doing a first experimental analysis, we noticed that EffiCuts generates a relatively large number of nodes (e.g. when compared to HiCuts). Analyzing the code allowed us to see that it was using a different heuristic to decide on the number of cuts to do on a given tree node. In each iteration of the heuristic used by EffiCuts to determine the number of cuts to be performed, a variable *Sm* (space measurement) is computed using (4.1) as the number of partitions done on the previous iteration, Np_i, plus the number of rules of the sibling child nodes under consideration. Those children nodes are virtual until inequality (4.3) becomes false. Then, the node considered is cut according to the number of cuts computed in Equation (4.2) at the last iteration that fulfills Equation (4.3).

$$Sm(i) = NumRules(children_i) + Np_i \tag{4.1}$$

$$Np_i = 2^i \tag{4.2}$$

$$Sm(i) \leq Smpf(NumRules) \tag{4.3}$$

where *Sm* is a measure of space requirement as defined in (Gupta et McKeown, 2000), i is the iteration number, *NumRules* is the number of rules contained in the node under analysis at iteration i, and *Smpf* a pre-determined space measure function (Gupta et McKeown, 2000). In our experimentations, $Smpf = NumRules \cdot 8$.

This algorithm is applied to every node until it contains a number of rules less or equal to a threshold value that was set to 16 in our experimentations. Thus, according to Equation (4.3),

$Smpf \geq 17 \cdot 8 = 136$. Also, the number of rules within the node being processed decreases as the number of iterations increases and, at the same time the (Np_i) term becomes more and more dominant. In our simulations, we observed many nodes being cut thousands of times, which creates, in the worst case, as many child nodes. With each cut, a node or a leaf is created. Thus, it is obvious that this heuristic presents a potential to generate an oversized data structure. The heuristic implemented in HiCuts is based on the following formula :

$$Sm(C(v)) = \sum_{i=1}^{Np(C(v))} NumRules(child_i) + Np(C(v)) \tag{4.4}$$

Where *Np(C(v))* represents the number of times the node v is partitioned (i.e. it is also computed according to Equation (4.2)), and *C(v)* represents a cutting iteration on the node *v*. The only difference between EffiCuts and HiCuts is that Equation (4.1) is replaced by Equation (4.4.1), and every other equations or inequality remain true. Here, the heuristic adds the result of all previous iterations along with the number of rules in the temporary nodes used, consequently *Sm* converges faster to *Smpf*. Thus, for HiCuts, the number of iterations is reduced, which implies that the number of cuts is also reduced and so is the data set size. Considering this observation, the heuristic presented in (Gupta et McKeown, 2000) was adopted in this paper.

4.4.2 Leaf size modulation

It is stated in (Vamanan *et al.*, 2010) that, in their experimentations, the tree depth had no influence on the number of memory accesses and that only redundancy was increased when the leaf size parameter was decreased. This is clearly not true in the OpenFlow context. The size of the leaves directly impacts on the tree depth ; when the size of the leaf is reduced, it tends to create more levels in the trees. Reducing the size of the leaves presents an advantage only if the reduction of the number of memory accesses in a leaf is higher than the overhead added due to a deeper tree. On the other hand, because twelve fields are being processed in our case, the size of the rules is much larger than in the 5-tuple case. In this case, a rule is about twice the size of a node (see 4.1 below), which increases the number of memory accesses. Thus, we observed in our experiments that lowering the size of the leaves may increase the tree depth by adding a few more levels, (causing more reads to memory) but this is compensated by the number of operations saved when reading a rule.

4.4.3 Adaptive grouping factor

EffiCuts uses *separable trees* in order to reduce the overlap between large and small rules. For every field of a rule, if the range covered by the rule is larger than 50% of the range of the considered field, this field is considered large. Otherwise, it is considered small. This process is performed on every field of a rule, and then rules are binned into different categories based on the combinations of large and small fields. The value of 50% introduced in (Vamanan *et al.*, 2010) was set based on experimental analysis of EffiCuts's authors. Even though, our conclusions are also based on experimental analysis, they are quite different. We call the percentage that defines the difference between large and small rules the *grouping factor D*. As previously stressed, the 12-tuple context creates a very large number of possible combinations. Considering that a tree is created for each combination and that all trees must be traversed, this implies that a large number of memory accesses must be performed for each rule look-up (even though the trees are small). Thus, our goal is to find a way to reduce the number of categories. To this end, we decided to make this parameter variable unlike the similar parameter in EffiCuts. A value of 50% for D sometimes introduces a large number of trees (see Section Extensions 4.5.2) which results in a larger number of memory accesses. We observed that this is mainly due to some fields that contain a small number of large rules thus creating a large number of small trees. The proposed solution simply consists of adapting D to the data at hand. We first experimentally analyze the percentage of large rules for every field, starting with a value of 50% for D. If we detect fields that contain a percentage of rules that is less than or equal to 10%, we increase the value of D by 10%. This procedure is repeated until those fields contain a percentage of rules close to 0%, thereby eliminating a significant number of trees. Reducing the number of trees improves performance in two ways. Firstly, from a hardware implementation perspective, reducing the number of trees reduces the amount of hardware resources needed to access a data structure that can be traversed more efficiently. Secondly, a smaller number of trees decreases the number of memory accesses by eliminating the need to traverse many small trees, and thereby many leaves. This method has been validated in simulation. Nevertheless, the introduction of the *adaptive grouping factor* tends to re-introduce overlap between rules, when considering a value for D higher than 50%, which tends to increase the size of data sets but, as we will show below, the resulting overhead is reasonably small.

4.5 Results

4.5.1 Methodology

Classification Table Generation

Because we did not have access to real classification tables for the 12-tuple context, we instead relied on table generators. The only two such generators available to us were ClassBench (Taylor et Turner, 2007) and FRuG (Ganegedara *et al.*, 2010). These generators do not meet our needs directly because ClassBench generates only 5-tuple rules and FRuG does not include any form of scenario so the configuration is left to the user. Thus, our strategy is to create sets of rules that can give us lower and upper bounds on the performance reached by EffiCuts. The strategy that we finally adopted consists of creating 5-tuple rules with ClassBench and 7-tuple rules with FRuG and to fuse them. As mentioned previously, FRuG does not use seeds as ClassBench does to generate classification tables. However, considering that EffiCuts produces very different results when the size of the rules and the overlap between them vary, we implemented two scenarios in FRuG : a "small" scenario with little overlap and mostly small rules (best case) and a "large" scenario with a lot of overlap and a wide variety of rule sizes, including wildcards (worst case). Regarding ClassBench, three different seeds were used, representative of : Firewall (FW), Access List (ACL) and IP Chain (IPC) such as those used in (Vaish *et al.*, 2011) and (Jiang et Prasanna, 2012). We generated classification tables with 10 000 rules and 100 000 rules.

Node organization

Before explaining the data structure of the tree nodes, some explanations are necessary. First, EffiCuts almost always cuts rules according to only two dimensions. The reason is simple : the algorithm used only splits dimensions that have a number of unique elements higher than the average. Considering that the number of unique elements is very high for the IP source and IP destination fields, then EffiCuts seldom splits a third dimension and it never splits more than three in the experiments that we performed. Considering this observation and the fact that it causes deeper trees to be generated (which is not a problem considering that trees are fairly small, thus the added depth is small), we made the assumption that only two dimensions would be split. In order to store the boundaries of two dimensions, we need $2 \cdot 5$ bytes (i.e. the size of the IP Addresses including the prefix, which are the largest relevant fields). Note that we voluntarily excluded the MAC field from the cutting process (because the number of unique elements is much higher in other fields) so we do not take the size of this field into account when computing the maximum size for the boundaries. The size of

the header is increased because we consider more fields and a maximal number of cuts of 256 for each dimension. Thus, the header size now equals 4 bytes. A rule contains as many boundaries as the number of fields used. For the Rule field, we need to store either upper and lower bounds, or just a value and its prefix depending on the packet field being processed. When considering 12 fields such as in (Open Networking Foundation, 2009), we need 41 bytes to describe a rule. Each node contains a header, boundaries (in order to know the edge of the covered region) and a child pointer. Equi-dense nodes carry one more field, due to the fusion of some child nodes (Vamanan *et al.*, 2010). We limited the maximal number of child nodes to 256, and thereby need only one byte to store an index value used by equi-dense nodes (Vamanan *et al.*, 2010). The details concerning the size of the node data structure are given in 4.1.

Measurements

The sizes in bytes of the various components of the node data structure are listed in 4.1 and (Vamanan *et al.*, 2010). The replication factor is computed as the total number of rules in the leaves of the trees divided by the total number of rules in the classification table. The memory bus width is assumed to be 18 bytes because this equals the size of an equi-size node and because this type of node is the most frequent one by far (Vamanan *et al.*, 2010). We assumed that we need 2 memory accesses for an equi-size node (including accessing the child node), and two accesses for an equi-dense node, as the size of these node remains higher than the bus width.

Tableau 4.1 Node data structure

Filed name	Filed size (bytes)	Field content
Header	4	node type, dimensions to cut, number of cuts per dimension
Rule	41	12 fields
Boundaries	10	two dimensions
Cut index	1	per unequal cut index ($max_{cuts} = 7$) (equi-dense)
Child pointer	4	Pointer value

4.5.2 Extensions

For our tests, we used the implementation provided directly by the authors of EffiCuts. We added our extensions on top of the EffiCuts algorithm ; separable tree, selective tree merging,

and node merging are also enabled. We begin by comparing EffiCuts with and without extensions in terms of memory requirements. In order to highlight the size of the node-structure created by EffiCuts, we plotted, in 4.1, the memory used per rule. Considering that the size of a rule is 41 bytes, we observe that EffiCuts adds a large amount of data to the minimal size that can be achieved. This is mainly due to the high number of nodes generated by the heuristic used (Vamanan *et al.*, 2010). The extensions added on top of EffiCuts decrease the memory used per rule by 35% on average (for both 10K and 100 K simulations). In our case, the amount of memory used per rule remains close to the size of a rule (41 Bytes). Our extensions tend to create trees with a smaller number of nodes, compared to the implementation of EffiCuts, which results in a low memory utilization per rule. Furthermore, this reduction is amplified by the small number of trees created by *Separable trees* combined with our *adaptive grouping factor D*. We found in our experimentations that a value of D close to 0.9 (90%)

Tableau 4.2 Replication factor comparison

	ACL		FW		IPC	
	Small	*Large*	*Small*	*Large*	*Small*	*Large*
10 K						
EffiCuts	1.01	1.02	1.03	1.06	1	1.01
EffiCuts Extended	1.04	1.05	1.38	1.44	1.1	1.18
100 K						
EffiCuts	1.1	1.11	1.08	1.07	1	1.1
EffiCuts Extended	1.39	1.32	1.07	1.28	1	1.13

achieves the best results. In 4.3, we report the number of trees used in EffiCuts before and after adding our extensions. As explained in the previous section, D affects only the number of trees created. In 4.3, the number of trees generated is given for all cases. The introduction of the *grouping factor* cuts down the number of trees by roughly 50%.

It was observed that limiting the number of categories when binning rules impacts the replication factor. When selecting only very large rules (D of 0.9 or higher), we re-introduce overlap between rules, a consequence of mixing different size of rules into the same category. Still, as shown in 4.2, the replication factor remains under 1.44 even in the worst case. It seems to contradict the previous results (i.e. that we decreased the memory utilization). This apparent contradiction is due to the new heuristic used that cuts down the size of the data structure, thus the overhead caused by increasing the replication factor is more than compensated. In the experiment we conducted, we often observed the large number of memory accesses required by EffiCuts for tree traversal. This is mainly due to the size of rules used in OpenFlow. In the worst case, EffiCuts needs up to 800 memory accesses to classify a packet.

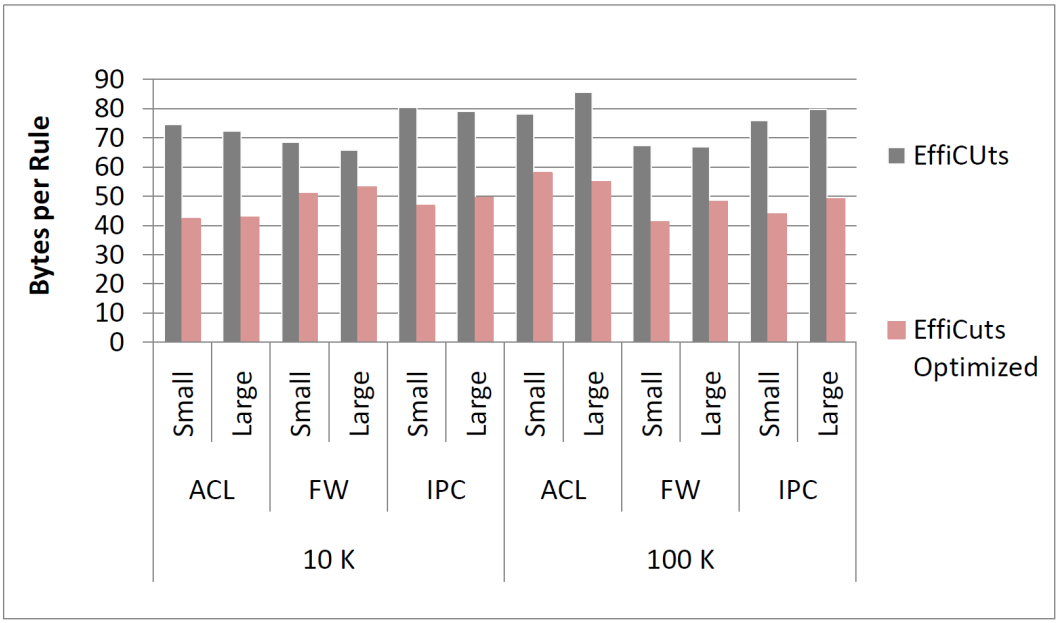


Figure 4.1 Bytes per rule for EffiCuts with and without extensions

This appears to be impractical and unacceptable. The adaptive grouping factor D reduces the number of trees to be traversed and thereby the number of memory accesses. However, the size of the leaves directly influences the number of memory accesses, including the depth of the trees, which means the size of the data structure is also influenced. We found that the leaf size that achieves the best tradeoff between the number of memory accesses and the size of the data structure is 8. To visualize this tradeoff, we plotted in 4.2 the largest observed number of memory accesses needed to complete the traversal of the entire forest of trees. It means that, for every tree, we assumed that the biggest leaf at the deepest level was selected. Note that, in the case of a hardware implementation, every tree would be traversed independently, thus the number of total memory access has to be compared with the number of trees used. Our extensions reduce this worst case by a factor as large as 2.8. For IPC classification tables, the improvement is about 20%. Nevertheless, on average, we cut down the number of memory accesses by 52% (i.e. a factor of about 2).

For every scenario simulated, we can observe a performance variation between cases with small and large rules for every criteria evaluated in 4.1, 4.2, 4.2, 4.3. The large rules have a higher degree of variation in size and overlap between rules than the smaller rules. It is also obvious that large rules create more categories, and thereby generate a larger number of trees than the small rules. Furthermore, D is higher in most scenarios when dealing with large rules when compared with the situation observed with small rules, but the difference remains under 15% in the worst case. The performance variation is relatively limited for the size per

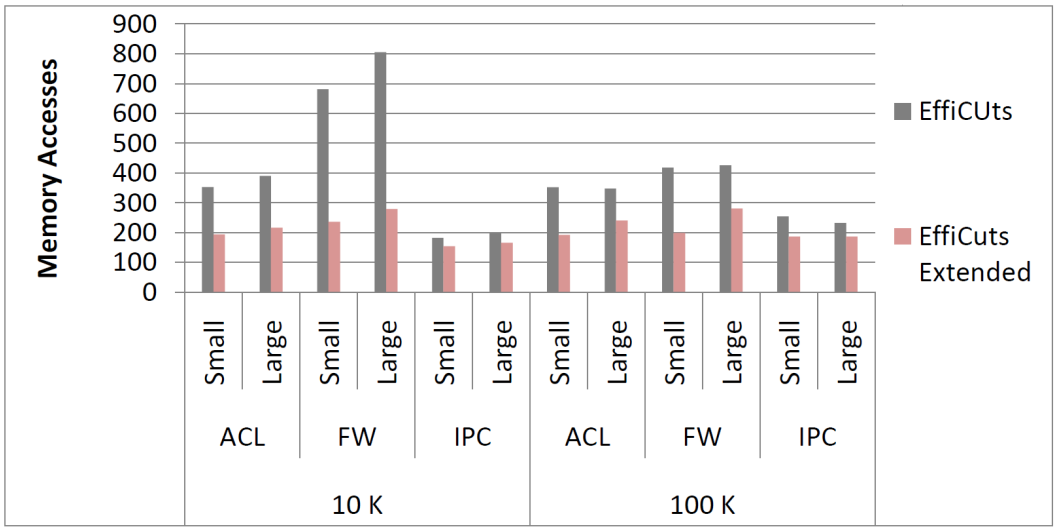


Figure 4.2 Total number of memory accesses for EffiCuts with and without extensions

Tableau 4.3 Number of trees generated

	ACL		FW		IPC	
	Small	*Large*	*Small*	*Large*	*Small*	*Large*
10 K						
EffiCuts	23	28	40	47	19	20
EffiCuts Extended	12	15	17	21	10	10
100 K						
EffiCuts	25	29	27	33	19	20
EffiCuts Extended	12	14	14	18	10	10

rule and is below 15%, while the difference for the memory accesses is 14% on average.

4.6 Conclusion

In this paper, it was argued that EffiCuts appears to be best available option to implement high performance classifiers. It was thus studied and some of its shortcomings uncovered when considered in the OpenFlow context. The main issue is the size of the rules for OpenFlow. In order to address this bottleneck, three extensions were proposed. They have a strong influence on two opposing parameters ; on one hand, the number of memory accesses and, on the other hand, the size of the data structure. Lowering the number of memory accesses in the perspective of a hardware implementation allows increasing throughput. Also, reducing the size of the data structure allows storing a higher number of rules using the same amount of memory. Until now, previous contributions were only optimizing one of these parameters in

isolation. We proposed changing the heuristic used to compute the number of cuts to process a node, of which the main impact is a reduction of the size of the data structure. In order to decrease the number of memory accesses, we introduced an adaptive grouping factor, which limits the number of categories (created when binning rules) and consequently the number of trees. Additionally, we discovered that the modulation of the leaf size can cut down the number of memory accesses while adding a relatively small overhead in size. These extensions cause a decrease in the number of memory accesses by a factor of 2, while decreasing the size of the data structure by 35%. These extensions also allow EffiCuts to be used with larger classification tables over a larger number of fields without excessive performance degradation, and it seems promising for the design of a high-speed hardware classification unit.

4.7 Acknowledgments

We thank the authors of EffiCuts for graciously providing us with their source code. We also thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, and Ericsson Canada for financial support to this research.

CHAPITRE 5

ARTICLE 2 : ACCECUTS : A PACKET CLASSIFICATION ALGORITHM TO ADDRESS NEW CLASSIFICATION PARADIGMS

Thibaut Stimpfling Normand Bélanger André Béliveau Ludovic Béliveau

Omar Cherkaoui Yvon Savaria

Cet article a été soumis le 17 janvier 2014 dans le journal *IEEE/ACM Transactions on Networking*.

5.1 Abstract

Packet Classification remains a hot research topic, as this telecommunication networks are now facing new challenges. Classification has to be more flexible and is based on many more packet fields than it used to be. Thus classification solutions are facing performance and scalability issues. Also, new applications and networking paradigms are emerging that current packet classification algorithms are not designed to handle. This paper proposes a new algorithm, AcceCuts, designed for better performance compared to other decision tree based algorithms. It cuts down the number of memory accesses and reduces the data structure size. It uses an Adaptive Grouping Factor, that allows parsing the rule set, in an optimized manner. AcceCuts adopts a heuristic for generating a smaller data structure without impacting on its depth. Finally, we propose a modification to leaf processing, in order to cut down the number of memory accesses. These extensions provide gains in many contexts, but they were tailored for handling complex rule sets which can be used with Software Defined Networking (SDN) context. It is shown, using suitable benchmarks, that they allow reducing the number of memory accesses by a factor of 3 on average, while decreasing the size of the data structure by about 45% over EffiCuts, a state-of-the-art decision tree based algorithm.

5.2 Introduction

Packet Classification is a functionality required by networking devices in a wide range of contexts like Quality of Service (QoS), load balancing, security, monitoring, and network traffic analysis. This functionality becomes more challenging as the average link speed is constantly increasing, while means of classifying improve at a slower pace than the bandwidth

on physical links. The challenge is compounded by a demand for classifiers to process many more fields and rules. Thereby, classification remains a hot research topic.

Packet classification aims at matching incoming packets with one or multiple rules, contained in a rule set. Even if packet classification is widely covered in the literature Taylor (2005); Gupta et McKeown (2001), most previous studies are linked to the classical 5-field context. Nonetheless, due to the significance of data centers, access and aggregation networks, and resources management, a global view of the system is required, from network equipment to servers Google (2012). Software Defined Networking (SDN) has been proposed to provide a framework for such global view.

SDN allows a small processing granularity and brings new possibilities, such as optimizing the link utilization rate, getting a unified view of the network fabric, improving failure handling, etc. Such improvements introduce drastic changes to networking. Hence, packet classification is deeply changed and has to handle much more complex rules over a larger number of fields. Thus, the classical 5-tuple context, which is largely covered in the literature, no longer matches the trends and evolution in the networking field Stimpfling *et al.* (2013); Fong *et al.* (2012). SDN and, more specifically, the OpenFlow protocol take more and more importance in the literature, mainly due to its high flexibility.

By contrast to the classical 5-tuple context, SDN rule sets, with large flow entries, are much more likely to be complex, due to the higher number of fields that can be used to classify a packet, and the ability to use masks, or wildcards on more fields. For instance, with version v1.0.0 of OpenFlow Open Networking Foundation (2009), up to 12 fields of a packet header can be used for classification.

In order to better understand the impact of complex rules on performance, we will consider, in this paper, SDN rule-sets defined on 12-tuples. It is of interest that, when using IPv6 or MAC addresses as in the latest evolution of an SDN protocolOpen Networking Foundation (April 2, 2013), the rule size increases substantially. For instance, real rules sets defined as 11 tuples are used for experimentation in Fong *et al.* (2012). Therefore, it is clear that SDN can lead to, on one hand, larger fields and, on the other hand, a larger number of fields. So, rules tend to be much more complex than what was considered in the classical 5-tuple context. This evolution will strongly impact packet classification performance and optimized algorithms are clearly needed.

Although more functionalities are offered to end-users, limited progress has been achieved at the algorithmic level. From an industrial point of view, TCAM-based solutions are widely used, although they have many drawbacks, such as limited flexibility and high power consumption Jiang et Prasanna (2009). According to the Open Networking Foundation, latest versions of protocols such as OpenFlow, used in the case of SDN, require support from

powerful TCAM-like tables, but with more capability than those offered by available and announced hardware implementations Beckmann *et al.* (April 4, 2013). We are clearly facing a bottleneck by offering much more flexibility to end users without any optimized hardware (matching these requirements) being available. Before attempting to design new hardware, the first step is to design suitable algorithms.

The algorithm proposed in this paper, AcceCuts, is designed to support a large number of complex rules, while reaching high performance. Also, AcceCuts offers good scalability regarding two main criteria : the number of memory accesses and the size of the data structure. The main contributions of this paper include :

- An adaptive grouping factor that sorts the rules before tree building, in order to obtain a better tradeoff between the number of trees built and rule separability. This factor is used in order to decrease the number of memory accesses.
- A heuristic that reduces the data structure size in order to handle a large number of rules.
- A new leaf structure to fetch only the information needed, eliminating many irrelevant memory accesses. This new structure is proposed to tackle one of the main sources of useless memory access in decision tree based algorithms presented in the literature.

Furthermore, to the best of our knowledge, AcceCuts is the first decision tree based algorithm that modifies the tree structure and tackle the issue of linear search at the leaves. Also, AcceCuts improves performance over two opposing performance criteria.

The paper is organized as follows. In Section 5.3, a review of packet classification algorithms is presented including the context considered for this paper. In Section 5.4, we identify weaknesses in a state of-the-art algorithm, EffiCuts, for the context of complex rules. Identifying limitations of existing algorithms is the first step before designing a new algorithm. We present our algorithm AcceCuts in Section 5.5 along with the impact of each of our contributions. The methodology adopted to characterize AcceCuts is discussed in Section 5.6. Then, a detailed performance evaluation of AcceCuts is made in Section 5.7. Finally, a conclusion to this work is given in Section 5.8.

5.3 Related work

Many approaches have been considered in the literature to tackle the problem of packet classification but existing algorithms appear to under-perform or are not tailored for handling complex rules.

5.3.1 Packet Classification Algorithms

We can categorize packet classification techniques in three main types : Decomposition Based, Decision Tree based, and pure hardware solutions, typically based on TCAMs.

Hardware TCAMs

Ternary Content Addressable Memories (TCAMs) are a powerful type of memory that offers O(1) time packet classification. To achieve such a high performance, TCAMs match, in parallel, each rule against the incoming packet header. TCAMs offer high performance but suffer from severals drawbacks. In particular, parallel match is extremely power consuming and TCAM chips are very costly. Many contributions have been made on this topic, targeting a high memory efficiency Huan (2002). Nonetheless, the power consumption issue is still inherent to the TCAM technology, while supporting range based rules remains an open issue Chao et Liu (2007). Such bottlenecks tend to limit the use of TCAMs in current and future networking contexts.

Finally, the Open Networking Foundation warns about the current evolution and latest version of the OpenFlow protocol as the switch model presented requires a very flexible series of powerful "TCAM-like" tables, with much more capabilities than any existing or announced hardware platforms Beckmann *et al.* (April 4, 2013).

Decomposition Based

One approach adopted in the literature to classify packets, named Decomposition Based, aims at separating the lookup process into multiple parallel reduced lookups, and then combining the results together. For instance, RFC Gupta et McKeown (1999) and HSM Bo *et al.* (2005) are both using this technique. Those algorithms can achieve good performance but suffer from a large memory requirement Singh *et al.* (2003); **?**. Decomposition Based algorithms are not scalable, due mainly to the memory requirements and consequently. Thus, they cannot handle large classification tables, which is an essential requirement in the context of this study.

Decision Tree Based

Decision tree based algorithms are another avenue proposed in the literature to address packet classification issues. Several algorithms in the literature use this approach, such as HiCuts Gupta et McKeown (2000), HyperCuts Singh *et al.* (2003), and EffiCuts Vamanan *et al.* (2010) (a state-of-the-art algorithm). These algorithms are adopting a geometric point of

view, as each packet header field can be viewed as a defining a volume in a multi-dimensional space. Therefore, in this paper, a dimension refers to a packet header field.

These algorithms operates in two steps ; firstly, one or multiple decision trees are built according to a rule set and, secondly, a tree traversal process is conducted on each tree, based on the incoming packet header. The second step represents the actual packet classification process.

We first give a high level description of the tree building process. These algorithms divide the rule space (i.e. the rule-set) into subsets in an iterative fashion, until each subset contains fewer rules than a given threshold. A tree building example is shown in Fig. 5.1, using the HiCuts algorithm. The first step is to cut the rule space along the dimension that maximizes the differentiation between rules. Accordingly, a first cutting sequence, represented with the green lines, is done along the *Field 1* direction, which generates four nodes. Three of those nodes contain fewer rules than the threshold value (set here to 2) so those nodes correspond to leaves (leaves 1 to 3). Node 1 stores three rules, so another cutting sequence has to be completed and is represented with a red line. This process creates two more leaves (Leaves 4 and 5). At that point, the decision tree is completely refined. The classification process consists in a simple tree traversal, from root node to leaves. The incoming packet header is compared with the rule space covered by each node, and then the position of the next child node to visit is computed based on informations contained in each node. When a leaf is reached, each rule is matched against the packet header, and the matching rules are then selected. The process of packet classification is completed, and a new packet can be processed.

The first proposed tree-based packet-classification algorithm, HiCuts (in Fig. 5.1), generates a lot of rules replication as it creates a single decision tree, thus mixing together rules with very significant differences in size (which causes a lot of superposition). On the other hand, it was devised to handle fairly small tables, so it is not surprising that nothing was done to improve this aspect.

HyperCuts was proposed as an evolution to HiCuts, which aims at improving the convergence rate of the classification (thus, minimizing tree depth) while limiting the data structure size. To achieve this, the algorithm is based on multi-dimensional cuts and it includes techniques to minimize replication. These techniques produce better performance in terms of number of memory accesses, but their scalability is poor Singh *et al.* (2003).

EffiCuts aims mainly at striking the best compromise between the average number of memory accesses and the data set size for the 5-tuple context. The authors Vamanan *et al.* (2010) designed this algorithm based on two key observations. Firstly, there is a lot of overlap between rules in a classification table. Secondly, the variation in size of rule overlap is high, leading to a high degree of rule replication caused by thin cuts. Thus, they proposed EffiCuts

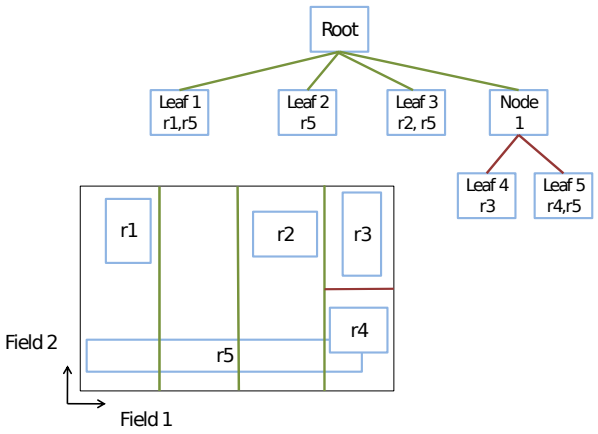


Figure 5.1 Example of building a decision tree

to address these issues by partitioning the rule set and binning rules with different size patterns in different subsets. Each of these subsets is then associated with a dedicated decision tree. This method is called separable trees. However, the introduction of multiple trees adds extra memory accesses, which decrease throughput. This problem is partially solved in EffiCuts with selective tree merging. This method aims at merging selectively separable trees, mixing rules that are either small or large in at most one dimension.

HiCuts Gupta et McKeown (2000) and HyperCuts Singh *et al.* (2003) cut the space covered by a node equally between each child node. By contrast, EffiCuts introduces equi-dense cuts, in order to tackle the problem of ineffective nodes containing only a few rules. This situation occurs when separating dense parts and empty zones.

Additionally, EffiCuts introduces other optimization techniques like node collocation. Node collocation was proposed in order to reduce the overall number of memory accesses.

It thus reduces considerably the memory usage compared to HyperCuts while causing a low replication factor. The replication factor, a synonym for rule replication, is defined as the number of rules used in the data structure, divided by the number of rules used as inputs for the data structure. On the other hand, separable trees, even with selective tree merging, tend to increase slightly the average number of memory accesses over HyperCuts.

5.3.2 Hardware Implementation

In the previous sections, different algorithms have been described. It is of interest to determine whether these algorithms can serve as a base for effective hardware implementations. An ideal algorithm could be one that is implementable in hardware and that offers good performance and scalability with problem size. This section thus reviews hardware implementations

of related algorithms.

In Jiang et Prasanna (2012), an FPGA implementation of the HyperCuts algorithm, including optimizations, is presented. While HyperCuts suffers from a high replication factor, optimizations are included to tackle this issue and to address hardware tree traversal issues. The implementation presented can process data at up to 80 Gbps for minimal packet size (40 bytes) while using 5-tuple classification tables. However, a study evaluating its scalability was conducted on OpenFlow-like rules (V1.0.0). In this case, the architecture that was implemented handles 40 Gbps of data for minimal packet size, with as many as 1000 rules. For this implementation, the larger number of fields (OpenFlow-like rules) cuts down the bandwidth by a factor of 2. Finally, one algorithmic optimization brought by this work (called the "Decision Forest") imposes a tradeoff between performance and data structure size. The performance over HyperCuts is increased but the classical tradeoff of memory versus performance is still not addressed.

Other FPGA implementations are presented in the literature, such as in Fong *et al.* (2012), where the algorithm used is based on HyperSplit Yaxuan *et al.* (2009), but, in order to increase the performance offered by ParaSplit over HyperSplit, two combined methods are used that sort the rule set and split it into different groups, prior to tree building. This algorithm improves the performance over HyperSplit and EffiCuts : the hardware implementation reaches 123 Gbps for one classification engine with minimal packet size (64 bytes). The main strength of this work resides in the small amount of resources used for an engine, allowing to implement multiple engines on a single FPGA, and, thereby, reaching a bandwidth higher than a terabit per second.

An alternative to FPGA implementation is to use an array of processors as shown in Vaish *et al.* (2011), where the authors implemented EffiCuts on the PLUG platform (Pipelined LookUp Grid). PLUG is a flexible lookup module platform designed to easily deploy new protocols in high-speed routers.

Multiple modifications to implement EffiCuts have been added to the PLUG platform on both the hardware and the software side. Even then, this implementation can only support 33 Gbps of data bandwidth for minimal packet size. The performance issue is mainly due to the limitation or weakness of this generic platform, rather than the implementation itself. However, this paper does not explore the performance of EffiCuts in the context of 12-tuple rules.

The decision tree based algorithms described here mainly focus on decreasing the replication factor and accelerating the convergence to leaves. Optimizations have been proposed on multiples fronts : before tree building and when generating tree nodes. Nevertheless, no means was presented for optimizing leaf traversal and for improving on linear-time rule matching.

To summarize, in this section, we have reviewed prior work that shows that decision tree based algorithms are implementable in hardware and that they offer respectable performance in the classical 5-tuple context. In some cases, as shown above, some exploration was conducted with OpenFlow-like rule, on 11 tuples (in Fong *et al.* (2012)) or 12 tuples (Jiang et Prasanna (2012)), but no optimization was proposed, no recommendation was made, and no in-depth analysis was performed. Due to the lack of analysis for the case of complex rules, it is hard to understand whether performance issues are linked to a lack of optimization or to the algorithms themselves and, therefore, this cannot be addressed based on prior work.

5.3.3 Our approach

Our approach aims at developing a new algorithm optimized to handle complex rules, in the perspective of a future hardware implementation. Thereby, multiple goals have to be achieved. First, our AcceCuts algorithm, proposed later in this paper, has to perform well from a theoretical point of view, while remaining implementable in hardware. Therefore, as a first step, our work aims at analyzing in details the performance achieved by EffiCuts (that appears to be the best prior solution in the literature) and identifying its limiting factors when handling complex rules over a large number of fields. AcceCuts has been designed to tackle issues highlighted in the previous analysis, and optimized to perform well when implemented in hardware.

As a general principle, we must avoid reading unused data, which leads to wasted clock cycles. Thereby, we want to minimize the number of memory accesses and fetch from memory only useful information.

However, we want our algorithm to use as little memory as possible, so AcceCuts is designed to build a relatively compact data structure. Limiting the data structure size allows to store either larger rules or larger classification tables on the same memory chip.

To summarize, the design of AcceCuts was guided by two main criteria : on one hand, reducing the number of memory accesses by reading only the required information and, on the other hand, reducing the size of the data structure.

5.4 Analysis of EffiCuts

5.4.1 Memory utilization

In this section, we look at the size of the data structure generated by EffiCuts. The total amount of memory used is difficult to appreciate ; it is easier to consider the memory size per rule, defined as the total size of the data structure divided by the number of rules of

the classifier. Also, this "bytes per rule" measure helps to differentiate the size of the tree overhead from the rules.

For readability, our experimental results characterizing the performance of EffiCuts are first reported in this section, but details of the methodology to obtain them are best presented later in Section 5.6.1 and Section 5.6.2. In the following simulations, the word Small is associated to a best case scenario, whereas Large is associated to a worst case scenario. Three types of classification tables are used : Access List (ACL), Firewall (FW) and IP Chain (IPC), holding 10 000 rules (10 K) and 100 000 rules (100 K). The source code of EffiCuts used for this analysis comes directly from EffiCuts authors, and has been extended to support 12-tuples.

Tableau 5.1 Size of the EffiCuts data structure : bytes per rule

Bytes per rule	ACL		FW		IPC	
	Small	*Large*	*Small*	*Large*	*Small*	*Large*
10 K	74.6	72.3	68.5	65.8	80.3	79.1
100 K	78	85.6	67.4	66.9	75.8	79.6

In Table 5.1, the size of the data structure generated by EffiCuts is measured as the total size divided by the number of rules, which leads to a performance measure expressed in bytes per rule. As observed in Table 5.1, the size of the data structure generated by EffiCuts is quite higher than the rule size (80% on average), as a rule weighs here 41 bytes Stimpfling *et al.* (2013). On the other hand, EffiCuts is a state-of-the-art algorithm as it generates a very small replication factor. Thus, the high memory consumption observed in Table 5.1 is not a consequence of a high replication factor, and is only linked to the large number of nodes in the trees built by EffiCuts. However, experimentally, we observed the maximal depth of trees generated by EffiCuts, which is pretty low with an average of 3. Under these circumstances, it is obvious that the convergence is really fast from root node to leaves. This is also reflected by measures of the number of cuts done at each node level, which is a direct observation of the number of nodes created. We discovered that, in one of our experiments, a node had 131072 child nodes. This creates trees with too many nodes, and consequently creates an over-weighted data structure.

5.4.2 Memory access overhead

The second part of this study aims at highlighting which parts of the trees generate the highest number of memory accesses. In Stimpfling *et al.* (2013), it is shown that SDN-related rules increase drastically the number of memory accesses, but no explanations of this

behaviour is provided. Different causes can lead to a high number of memory accesses in our case : deep trees, leaf traversal, or both.

The number of memory accesses depends directly on the tree depth t_{depth}, on the number of memory accesses to fetch a node Ma_{node} and a rule Ma_{rule}, and, finally, on the number of rules held in a leaf $nr(L_j)$. Equation (5.1) is used as a criteria to decide whether tree depth or leaf traversal requires more memory accesses in tree traversal.

$$nr(L_j) \cdot Ma_{rule} \geq Ma_{node} \cdot t_{depth} \tag{5.1}$$

With our hypotheses (stated in Section 5.5), $Ma_{rule} = 3$, and $Ma_{node} = 2$. Thus, Equation 5.1 can be simplified as Equation (5.2).

$$nr(L_j) \cdot 1.5 \geq t_{depth} \tag{5.2}$$

Experimentally, we observed, for all scenarios, a maximum tree depth t_{depth} equals to 4. Nonetheless, the maximum number of rules carried by a leaf was 16. Thus, Equation (5.2) remains true in the worst case. Nevertheless, this equation should remain true even on average for every tree generated by EffiCuts. Thus, the average number of rules per leaf has been observed in simulations.

The minimum average number of rules per leaf observed in our simulation is **3.1**. Consequently, Equation (5.2) is fulfilled, which demonstrates that leaf traversal requires a higher number of memory accesses than node structure traversal. Experimentally, we obtained the same conclusion even with smaller leaves size (down to 8). It is thereby necessary to address the memory accesses issue, first, at the leaf level.

One other issue to address in the SDN context is linked to the explosion of the number of trees generated by EffiCuts, as stated in Stimpfling *et al.* (2013). Since the classification process involves every tree generated, this explosion leads to large leaf traversal time and number of memory accesses. As a consequence, we focused our attention on decreasing the number of trees generated, while decreasing the total number of memory accesses.

In this analysis, we pointed out some undesirable characteristics of EffiCuts in a SDN context. To summarize, we have to address two issues : high memory usage, and large number of memory accesses.

5.5 AcceCuts algorithm

AcceCuts is an algorithm that introduces several methods, in order to build fewer trees, cut down the data structure size, and address the issue of large number of memory accesses

observed with EffiCuts. Even though EffiCuts suffers from multiple drawbacks in an SDN context, some ideas introduced by this algorithm remain highly relevant. As exposed in Section 5.3.1, EffiCuts firstly bins rules into different categories using two methods : Separable Trees and Selective Tree Merging. In Efficuts, a tree is built for each category. Tree building involves several heuristics to decide which dimensions to cut, the number of cuts, and what what type of node to use. Finally, when the tree building process is completed, the packet classification process relies on a straightforward tree traversal algorithm of each tree generated by EffiCuts.

AcceCuts reuses some methods brought by EffiCuts for both tree generation and tree traversal. Regarding tree generation, AcceCuts reuses Selective Tree Merging but introduces an Adaptive Grouping Factor rather than using Separable Trees. During tree building, AcceCuts reuses both heuristics used to determine which dimensions to cut along, and to select a type of node. Nevertheless, AcceCuts leverages a new heuristic to compute the number of cuts to do, and it introduces a new leaf structure. As a consequence, AcceCuts tree traversal is based on EffiCuts tree traversal but with a new leaf traversal method. In the following sections, we present our three main contributions.

5.5.1 Adaptive Grouping Factor

EffiCuts uses separable trees in order to reduce the overlap between large and small rules. In EffiCuts, for every field of a rule, if the range it covers is larger than 50% of the range of the considered field, this field is deemed large. Otherwise, it is small. This process is performed on every field of a rule, then rules are binned into different categories based on the combinations of large and small fields. The value of 50% introduced in EffiCuts was set based on the author's experimental analysis. We observed experimentally a quite different behavior when modulating this value.

This threshold, used to classify a field as a large or a small one, is called the grouping factor D. As previously stressed, the 12-tuple context creates a very large number of possible combinations. Considering that a tree is created for each combination and that all trees must be traversed, this implies that a large number of memory accesses must be performed for each rule look-up (even though the trees are small). Thus, one of our goals is to find a way to reduce the number of categories. To this end, we decided to make D variable, unlike the similar parameter in EffiCuts. Indeed, a value of 50% for D introduces a large number of trees (see Stimpfling *et al.* (2013)), which results in a larger number of memory accesses. We observed that this is mainly due to some fields that contain a small number of large rules, thus creating a large number of small trees. Adjusting D dynamically to the analyzed part of the data set can mitigate that. To make D adaptive, the data set is first analyzed with

D set to 50% for all fields. If we detect fields that contain no more than 10% of the range, we increase the value of D by 10%. This procedure is repeated until those fields contain a percentage of rules close to 0%, thereby eliminating a significant number of trees. Reducing the number of trees improves performance in two ways. Firstly, from a hardware implementation perspective, reducing the number of trees reduces the amount of hardware resources needed to access a data structure that can be traversed more efficiently. Secondly, a smaller number of trees decreases the number of memory accesses by eliminating the need to traverse many small trees, and thereby many leaves. This method has been validated in simulation and works pretty well for rules with wildcards on some fields. The means through which grouping factor was made adaptive in this paper is rather rudimentary, however, improving Adaptive Grouping Factor efficiency is out of scope for this paper. The introduction of the Adaptive Grouping Factor tends to re-introduce overlap between rules, when considering a value for D higher than 50%, which tends to increase the size of data sets, the resulting overhead was found to be reasonably small as characterized later.

5.5.2 Modifying the Cutting Heuristic

Based on an experimental characterization and analysis, we noticed that EffiCuts generates a large number of nodes (e.g. when compared to HiCuts). Analyzing the source code allowed us to see that it was using a different heuristic to decide on the number of cuts to do on a given tree node. In each iteration of the heuristic used by EffiCuts to determine the number of cuts to be performed, a variable Sm (space measurement) is computed using (5.3) as the number of partitions done on the previous iteration, Np_i, plus the number of rules of the sibling child nodes under consideration. Those children nodes are virtual until Inequality (5.5) becomes false. Then, the node considered is cut according to the number of cuts computed in Equation (5.4) at the last iteration that fulfills Equation (5.5).

$$Sm(i) = NumRules(children_i) + Np_i \tag{5.3}$$

$$Np_i = 2^i \tag{5.4}$$

$$Sm(i) \leq Smpf(NumRules) \tag{5.5}$$

where Sm is a measure of the space as defined in Gupta et McKeown (2000), i is the iteration number, *NumRules* is the number of rules contained in the node under analysis at iteration i, and *Smpf* is a pre-determined space measure function Gupta et McKeown (2000).

In our experimentations, $Smpf = NumRules \cdot 8$. This algorithm is applied to every node until it contains a number of rules smaller or equal to a threshold value that was set to 16 in our experimentations. Thus, according to Equation (5.5), $Smpf \geq 17 \cdot 8 = 136$. Also, the number of rules within the node being processed decreases as the number of iterations increases and, at the same time, the Np_i term becomes more and more dominant. In our simulations, we observed many nodes being cut thousands of times, which creates, in the worst case, as many child nodes. With each cut, a node or a leaf is created. Thus, it is obvious that this heuristic presents a potential to generate an oversized data structure. The heuristic implemented in HiCuts is based on the following formula :

$$Sm(C(v)) = \sum_{i=1}^{Np(C(v))} NumRules(child_i) + Np(C(v)) \tag{5.6}$$

Where $Np(C(v))$ represents the number of times the node v is partitioned (i.e. it is also computed according to Equation 5.4). Here, the heuristic adds the result of all previous iterations along with the number of rules in the temporary nodes used, consequently Sm converges faster to $Smpf$. Thus, for HiCuts, the number of iterations is reduced, which implies that the number of cuts is also reduced and so is the data set size. Considering this observation, the heuristic presented in Gupta et McKeown (2000) was adopted in this work.

5.5.3 Leaf Structure Modification

This section first presents the proposed modified leaf structure. By changing the leaf data structure, we propose a new leaf traversal procedure tailored to access only relevant information, thereby avoiding many useless memory accesses. As shown previously, the main issue for leaf traversal resides in the large number of rules held in a leaf. To confirm a match, each field of a rule has to be matched, against the packet header. Conversely, when a packet field does not match a sub-rule (a rule constraint on a single field), the entire rule can be discarded. Matching a sub-rule is a necessary but not sufficient condition to match the entire rule. Therefore, we can proceed to partial matching with a rule, by doing a match only on a field of this rule, rather than matching a rule entirely.

Recall that one of our goals is to minimize the number of memory accesses, so we want to maximize the differentiation between rules, by using as few as possible fields to discard rules that do not match. On the other hand, if we use a single field for every rule, then, due to a high probability of rule overlap along a single field, many partial matches may be positive, and then will require to fetch the entire rule. In this case, it is harder to cut down the number of memory accesses. Consequently, we propose to create multiple sets of sub-rules, each one

containing sub-rules disjoint two by two, along one field, so that, at least one positive match can occur per sub-rule set. For each leaf level, AcceCuts sorts rules, and differentiates them into multiple sub-rule groups. A pre-processing is now required before completing the leaf traversal, as multiple partial matches may occur. Leveraging the result of this pre-processing, a regular match is only performed on a reduced number of rules. This process is applied on each tree generated by AcceCuts.

To help understand the modified leaf structure, we adopted the notation detailed in Table 5.2.

Tableau 5.2 Notation used for AcceCuts

Notation	**Detail**
P	Incoming packet
P_i	Value of the i-th field of the incoming packet
F_{il}	A specification of a rule on the i-th header field, for the l-th rule
R_i	i-th rule of the rule set
$s(R)$	Size of a rule contained in the rule set, in bytes
d	Number of fields used in the considered classification table
w	Memory bus width
L_j	The j-th leaf of the tree
$nr(L_j)$	The number of rules covered in leaf L_j
S_{ij}	A rules subset of L_j on the i-th header field
pm_k	Number of positive matches for a packet k
c	Cost of a comparison between an F_{il} and a packet.
$\left\|S_{ij \in L_j}\right\|$	Number of rules subsets S_{ij} created by AcceCuts for the leaf L_j
s(F_{il})	Size of the F_{il} considered (in bytes)
s(H_j)	Size of the header of L_j
H_j	Header of L_j
ovlp(L_j)	Number of overlapping rules in leaf L_j
Ma_{node}	Number of memory accesses for a node
Ma_{rule}	Number of memory accesses for a rule
Ma_j	Number of memory accesses for a leaf j, in EffiCuts
$Ma_{j,k}$	Number of memory accesses for a leaf j and a packet k, in AcceCuts
$C_{j,k}$	Comparison cost for a leaf j and a packet k, using AcceCuts
C_j	Comparison cost for a leaf j using EffiCuts

Modifying leaf traversal requires a new leaf header as shown in Fig. 5.2. The first leaf header field only specifies that the current node is a leaf. The following leaf header field contains the leaf header length. The third leaf header field, called AcceCuts header, is used

to store all information to complete the pre-processing step introduced in AcceCuts.

Leaf code	Size of AcceCuts header	AcceCuts header

Figure 5.2 Header of a leaf in AcceCuts

Packet header field index used	Number of entries $\|S_i1\|$	Sub-rule bound 1 F_{i1}	Sub-rule bound 2 F_{i2}

Figure 5.3 Structure of an AcceCuts header

Details about AcceCuts header fields are presented in Fig. 5.3. AcceCuts header is used to describe a sub-rule group. Each of these sub-rule groups is described by three leaf header fields. Two of them have a fixed length and the last one has a variable size. The first two leaf-header fields hold respectively information regarding the packet-header field index used and the number of sub-rules in this group. Then follows the variable length size field, which contains bounds of each sub-rule contained in the sub-rules group.

In the example shown in Fig. 5.4, we represented a sub-rule group associated with a rule where the packet header field index is 3, holding two sub-rules and the bounds for each two sub-rules.

3	2	[28 : 31]	[12 : 25]

Figure 5.4 AcceCuts header : an example of a set S_{ij} holding 2 sub-rules and where the packet header field index is 3

For each leaf field presented, the leaf-field size is exposed in Table 5.3. The first leaf field used is associated with the dimension (or packet header field index) used for the sub-rule group. With our hypothesis, up to 12 packet-header fields can be used to classify packets, and thereby, we need 4 bits to store a value going from 0 to 11. Each sub-rule group can carry at most as many sub-rules as the number of rules in the leaf. In our case, we limited the maximal number of rules per leaf to 12. Consequently, we need only 4 bits to store the number of sub-rules held by each group. Regarding the last leaf field used, it stores lower and upper bounds of each sub-rule and ranges from 1 (small field and mask) to 7 bytes (MAC field and mask value) depending on the field considered.

5.5.4 AcceCuts Leaf Creation Procedure

As mentioned at the beginning of Section 5.5, AcceCuts derives from EffiCuts, but it introduces multiple extensions. A significant contribution of this paper is the proposed mo-

Tableau 5.3 Size of the fields used in the leaf header

Field	Size (bit)
Field used	4
Number of rule used	4
Sub rule	1 – 7 bytes

dified leaf structure. In order to understand why the proposed modified leaf structure allows significant performance improvements, the leaf building process and leaf traversal procedures of AcceCuts are presented in details.

AcceCuts Leaf Buidling Procedure

The pseudo-code of the procedure used to build an AcceCuts leaf is presented in Fig. 5.5. This procedure is best explained through an example. Let us consider a leaf L_j holding rules as presented in Table 5.4.

When building a leaf, AcceCuts uses two stacks, the first one, called *drs*, is used to store the rules associated with a sub-rule group, and the second one, *urs*, is used to store undifferentiated and unprocessed rules. Note that rules are considered undifferentiated when they are not disjoint two by two along at least one field.

First, each of the 6 rules held by leaf L_j are copied to the *urs* stack (**line 1**). Then, the first rule in leaf L_j, R_1 is selected and is used as a reference rule (**line 4**). Then, the procedure starts to differentiate rules into sub-rules along each packet header field used (**lines 2 to 7**). AcceCuts first compares, along packet header field A, sub-rule F_{A1} with the others sub-rules F_{Al}, $l \in \{2, 6\}$. The procedure then selects the first sub-rule F_{Al} not overlapping F_{A1}. However, sub-rules have to be disjoint two by two and consequently, before adding a sub-rule to the *drs* stack, the pending sub-rule is compared to all rules already included in the *drs* stack. The *drs* stack content is used to create the leaf header associated to the sub-rules (**lines 8 to 12**) only after each rule has been processed along the same packet header field.

Tableau 5.4 Example of rules contained in a leaf

Rule	Field A	Field B	Field C
1	20 : 51	0 : 31	100 : 248
2	1023 : 1050	2 : 46	39 : 41
3	36 : 512	4 : 10	28 : 31
4	61 : 92	12 : 19	11 : 28
5	102 : 257	22 : 54	1 : 10
6	150 : 189	1 : 38	12 : 25

```
\\Construction of S_ij groups
1. Copy every rule R_i contained in L_j in urs
2. For each field f used, f ∈ {1 , d}
3.      while (urs ≠ {∅}){
4.              Use R_1 as a reference rule
5.              For (R_i in {L_j}R_1){
6.                      If (F_fi ∩ F_f1 = { ∅ } and ∀ R_k ∈ drs
                        F_fi ∩ F_fk = { ∅ }) {
7.                          Add R_i to drs
                        }
                }
8.              If ( drs ≠ { ∅ } ) {
9.                  Add R_1 to drs
10.                 Create the header associated to the
                    group of rules in drs
11.                 Remove every rules contained in drs
                    from urs
12.                 Flush drs}
        }
\\Construction of S_ij groups dealing with two by two
\\non-disjoint rules after first processing
13. If ( urs ≠ { ∅ }) {
14.     Select a field which has not been chosen yet
        and with a minimum size
15.     Copy urs in drs
16.     Create the header associated with rules in drs
17.     Flush drs }
```

Figure 5.5 Building an AcceCuts leaf

In our case, sub-rules F_{A2}, F_{A4}, and F_{A5} are added to *Group 1*, associated to packet header field A. The processing along packet header field A is now completed. The associated leaf header for *Group 1* is now created, according to the format presented in Fig. 5.3.

We can now start the same process along the next packet header field with the remaining rules (**lines 3 to 12**). This only concerns rules R_3 and R_6. Here, R_3 is used as the reference rule. Nevertheless, along the B packet header field, those sub-rules are overlapping, thus we cannot differentiate sub-rules. The process is now done along packet header field C, from which we can differentiate the remaining sub-rules. The differentiation process is now completed and the last step is to create the associated header. Two sub-rules groups are now created and represented in Table 5.5.

This example represents a favorable case, as some rules may share common characteristics,

Tableau 5.5 The two sets created

Group 1		**Group 2**	
Rule	*Field A*	*Rule*	*Field C*
1	20 : 51	3	28 : 31
2	1023 : 1050	6	12 : 25
4	61 : 92		
5	102 : 257		

therefore leading to a situation where it is impossible to create a S_{ij} group with two by two disjoint rules. For this purpose, a second processing is applied on all non-disjoint rules as exposed in **lines 13 to 17**. In this case, all non disjoint rules are simply put together, and the packet header field associated to the S_{ij} group is chosen so that it minimizes the leaf header size if it has not already been selected for another S_{ij} group during the first processing.

Once each tree (and therefore leaf) is entirely built, the tree traversal processing can be executed. Since AcceCuts reuses some methods of EffiCuts, the tree traversal sequence differs from EffiCuts only for the leaf traversal.

Leaf Traversal

In this section, we present the procedure used to complete traversal of leaf L_j. We assume that the entire leaf header has been read, before starting the leaf traversal process. The explanations of the leaf traversal procedure will be given based on an incoming packet header presented in Table 5.6, on the leaf built in Table 5.5 and on the pseudo-code exposed in Fig. 5.6.

Tableau 5.6 Example of incoming packet

Field A	Field B	Field C
412	12	29

Each sub-rule F_{il} of each group is matched against the appropriate packet header field P_i (**lines 2 to 10**). In case of a positive match, the associated rule is read and matched entirely against the other packet fields (**lines 4 to 7**). If we consider the incoming packet presented in Table 5.6, our procedure selects each sub-rule of every sub-rule group, begins with *Group 1*, and matches them against the packet header field A. If the procedure finds a F_{Al} so that $P_A \subset F_{Al}$, a positive match occurs and a complete match must be performed on the rule R_l associated to F_{Al} (**lines 4 to 7**).

In the example shown, we cannot find a sub-rule F_{Al} so that $P_A \subset F_{Al}$, no sub-rule contains the value 412 in *Group 1*. Therefore, the algorithm executes the same processing

on the next sub-rule group, that is to say *Group 2*, and will match sub-rules against packet field *C*. $P_C = 29 \subset [28;31]$ and, thereby, a partial positive match occurs with sub-rule F_{C3}. Following that partial positive match, a standard match is performed with R_3. Nevertheless, in our example, the match remains negative and the packet shown in the example ends up matching no rule.

When multiple positive matches occur, rule numbers and their respective priorities are transmitted to the processing unit, which treats the packet according to the related processing associated to each rule (**line 11**).

1. Decode the leaf header
2. For ($\forall$ S_{ij} in L_j) {
3. For ($\forall$ F_{ip} in S_{ij}){
4. If (P_i matches F_{ip}) {
5. If (P_i matches R_p) {
6. Record the rule priority of R_p
7. }
8. }
9. }
10. }
11. Treat matched rules according to their related processing.

Figure 5.6 AcceCuts leaf traversal

5.5.5 Complexity Study

The modification of the leaf data structure requires a pre-processing prior to rule matching. The aim of this section is to analyze the overhead introduced by this new leaf data structure, regarding two criteria : number of memory accesses, and cost comparison.

Memory Accesses

Before matching sub-rules and rules, the entire header of the leaf has to be fetched from memory, generating a few memory accesses. In this section, we will propose an analytic model that predicts the number of memory accesses required to match rules in a leaf, in the worst case and in the best case, with AcceCuts and then with EffiCuts. This model will allow characterizing the improvement introduced by AcceCuts.

– AcceCuts

Let us introduce $Ma_{j,k}$ as the number of memory accesses for a leaf L_j and a packet k, using AcceCuts. $Ma_{j,k}$, depends simply on the amount of data to read divided by the bus width w. In our case, several parameters impact on the amount of data read : the leaf header size $s(H_j)$, the number of positive matches of sub-rules pm_k (for a packet k) and the size of a rule $s(R)$ (that multiplies pm_k). Equation 5.7 derives from this observation. The parameter $s(H_j)$ is linked to the size of each sub-rule group header, and the amount of data required to store each sub-rule bound for each sub-rule group. Since the total size of the two fixed-size leaf header equals 1 byte, $s(H_j)$ depends directly on the number of sub-rules categories $\left|S_{ij\in L_j}\right|$ held by the leaf L_j and the sum of each sub-rule bound size $s(F_{il})$, as expressed in Equation 5.8.

$$Ma_{j,k} = \frac{s(H_j) + pm_k \cdot s(R)}{w} \tag{5.7}$$

$$s(H_j) = \left|S_{ij\in L_j}\right| + \sum_{S_{ij}\in L_j} s(F_{il}) \tag{5.8}$$

As a first step to evaluate the overhead introduced by AcceCuts, equation 5.7 is evaluated in the worst and best case scenarios.

The worst case occurs when each sub-rule held in the leaf header generates a positive match, and forces to read every rule. This situation occurs when a leaf holds a single sub-rule group, with sub-rules not differentiated two by two. This give us a maximum value for pm_k, and obviously, the minimum value is equal to 0.

Moreover, in order to differentiate rules between them, a sub-rule group carries at least 2 rules. Nevertheless, we can easily imagine a leaf in which each sub-rule groups contains 2 rules, with an odd total number of rules, and thereby the last sub-rule group holds only one sub-rule. Therefore, the highest number of sub-rule group $\left|S_{ij\in L_j}\right|$, is $\frac{L_j}{2} + 1$. The minimum value is equal to 1.

From those observations derives the set on inequalities 5.9.

$$\begin{cases} nr(L_j) & \geq pm_k \geq 0 \\ \frac{L_j}{2} + 1 & \geq \left|S_{ij\in L_j}\right| \geq 1 \\ 7 & \geq s(F_{il}) \geq 1 \end{cases} \tag{5.9}$$

Regarding the size required to store upper and lower bounds, we made the same assumptions as presented in the previous paragraph, that is to say ranging from 1 to 7 bytes.

– EffiCuts

Similarly, we introduce Ma_j as the number of memory accesses for a leaf L_j using EffiCuts. Ma_j depends on three parameters : the number of rules held the leaf L_j, the size of a rule

$s(R)$, and the bus width w as shown in Equation 5.10. EffiCuts is much simpler as every rule held in the leaf has to be read, since multiple matches are possible. As discussed before, the complexity of exploring a leaf can be expressed by Equation (5.10).

$$Ma_j = \frac{nr(L_j) \cdot s(R)}{w} \tag{5.10}$$

– Comparison Between the Complexity of Accecuts and Efficuts

In this paragraph, we compare the performance in the worst and best cases between the two algorithms.

Regarding AcceCuts, recall that each sub-rule group S_{ij} carries sub-rules disjoint two by two, except in cases where the last sub-rule group can hold undifferentiated rules. Therefore, every sub-rule held in a leaf generates a positive match if and only if they are not differentiated two by two in the same sub-rule group. This case appears either when only one sub-rule is held in S_{ij}, or when the last sub-rule handles all undifferentiated sub-rules. However, we considered, for the worst case scenario, that $pm_k = nr(L_j)$ and $\left|S_{ij \in L_j}\right| = 1$. Independently of the considered scenario, we assume $s(R) = 41$ bytes (See Stimpfling *et al.* (2013)). Finally, the upper bound of $s(F_{il})$ is, based on Inequality (5.9). Based on those assumptions, we can write (5.11), (5.12) and (5.13).

$$s(H_j) = 1 + 7 \cdot nr(L_j) \tag{5.11}$$

$$1 + 7 \cdot nr(L_j) \ll 41 \cdot nr(L_j) \tag{5.12}$$

$$1 + \min(s(F_{il})) \cdot nr(L_j) + s(R) \approx \min(s(F_{il})) \cdot nr(L_j) + s(R) \tag{5.13}$$

For the best case, we assume that $pm_k = 1$ and $\left|S_{ij \in L_j}\right| = 1$. This assumption means that only one positive match occurs with a sub-rule, leading to reading and matching only one rule. However, in the best case, only one positive partial match occurs ; this scenario occurs when only one sub-rule group S_{ij} holding differentiated rules exists. Based on those assumptions, we derive (5.13). Table 5.7 summarizes the main conclusions of this analytic comparative study.

Through the analytic model just presented, the order of complexity of AcceCuts in the worst case is the same as the order of complexity of EffiCuts. Note that complexity polynomials can be somewhat misleading, and, as will be shown later through experimental results, the actual complexity of Accecuts was observed to be systematically lower than that of Efficuts in worst case benchmarks. Moreover, when considering the best case, our prediction is that AcceCuts is a way ahead of EffiCuts.

Tableau 5.7 Worst case and Best Case memory accesses comparison with a leaf using AcceCuts and EffiCuts

	EffiCuts	**AcceCuts**
Worst case	O $(\frac{nr(L_j)\cdot s(R)}{w})$	O$(\frac{nr(L_j)\cdot s(R)}{w})$
Best Case	O $(\frac{nr(L_j)\cdot s(R)}{w})$	O $(\frac{\min s(F_{il})\cdot nr(L_j)+s(R)}{w})$

Evaluation of the Cost of Comparing a Packet Header with a Leaf Using Accecuts and Efficuts

The leaf traversal process involves a tentative match of every sub-rule F_{il} held in each sub-rule group S_{ij} against the packet header. Then, for every positive partial match, a complete match has to be done along each field used in the rule. This process requires many matches from a comparison point of view. By contrast, EffiCuts does not require any pre-processing, and needs only to match linearly each rule held in a leaf. Nevertheless, AcceCuts is designed to retrieve only relevant information, leading to sub-rules and rules to match. This paragraph aims at evaluating the potential comparison overhead introduced in AcceCuts by the new leaf data structure compared to EffiCuts. For this paragraph, we introduce c, the cost of a sub-rule comparison, independently of the field considered.

– AcceCuts

When reaching an AcceCuts leaf, a comparison against all sub-rule group entries has to be performed. Since AcceCuts header holds as many sub-rules as the number of rules covered by the leaf, AcceCuts pre-processing requires a number of comparisons that is equal to the $nr(L_j)$. Accecuts must also perform as many comparisons as the number of partial matches pm_k in each dimension. From this observation derives Equation 5.14.

$$C_{j,k} = c \cdot (nr(L_j) + pm_k \cdot d) \tag{5.14}$$

– EffiCuts

Regarding EffiCuts, no matter which scenario is considered, each field of each rule must be matched against the packet header. So, the matching process of a single rule costs c multiplied by the number of fields considered.

$$C_j = c \cdot nr(L_j) \cdot d \tag{5.15}$$

– Comparison between the cost of comparing one packet with one leaf of AcceCuts and Efficuts

The assumption regarding the worst and best cases remains identical to the one presented in the previous paragraph in Inequation 5.9 Adopting a similar reasoning and similar simplifications, we derive a summary of the various rule-comparison cost expressions, provided in Table 5.8.

The cost of matching one packet header with a leaf for AcceCuts remains pretty close to the cost for EffiCuts in the worst case. Nevertheless, in the best case, EffiCuts needs to match a single rule, and thereby requires fewer comparison than AcceCuts, which requires some extra comparisons due to the pre-processing step.

Tableau 5.8 Worst case and Best Case complexity comparison between AcceCuts and EffiCuts

	EffiCuts	**AcceCuts**
Worst case	O $(nr(L_j) \cdot d \cdot c)$	O$(nr(L_j) \cdot d \cdot c)$
Best Case	O $(nr(L_j) \cdot d \cdot c)$	O $((nr(L_j) + d) \cdot c)$

5.6 Experimental methodology

5.6.1 Algorithm and Parameters used

In this section, we evaluate, using multiple benchmarks, the performance offered by AcceCuts, and measure the improvement over EffiCuts. Several performance metrics are measured such as the memory consumption, the number of memory accesses, and the number of comparisons.

The performance of AcceCuts depends on the incoming packets. As a consequence, measuring AcceCuts performance with packet injection shows the performance of the algorithm for a specific input packet distribution and the type of traffic carried in a particular part of a network. Thereby, to present a wider range of results, we measured parameters for the worst and best cases. Therefore, we can bound the performance reached by AcceCuts.

The version of EffiCuts used as reference is the original code created by EffiCuts authors. When evaluating performance, the parameters that were used are identical to those provided in Vamanan *et al.* (2010), except for the leaf size, as summarized in Table 5.9. Regarding leaf size, we found that AcceCuts reaches its optimal performance when considering a leaf size

different from the one used in EffiCuts. Nevertheless, in order to present a fair comparison, we choose to run simulations for AcceCuts and EffiCuts with the same leaf size.

Tableau 5.9 Parameters used for simulations

Value of parameter	**AcceCuts**	**EffiCuts**
Size of leaf	12	12
Binning *Factor D*	*Adaptive Grouping Factor*	0.5
Binning *Factor D* for IP field	0.05	0.05
Space Factor	8	8
Maximum number of intervals	7	7

5.6.2 Classification Table Generation

Because we did not have access to real classification tables for the 12-tuple context such as Fong *et al.* (2012), we instead relied on table generators. The only two such generators available to us were ClassBench Taylor et Turner (2007) and FRuG Ganegedara *et al.* (2010). These generators do not meet our needs directly because ClassBench generates only 5-tuple rules and FRuG does not include any form of scenario so the configuration is left to the user. Thus, our strategy was to create sets of rules that can give us lower and upper bounds on the performance reached by EffiCuts. The strategy that we adopted consists of creating 5-tuple rules with ClassBench and 7-tuple rules with FRuG and to fuse them randomly. As mentioned previously, FRuG does not include any seeds, such as ClassBench, to generate classification tables. However, considering that EffiCuts produces very different results when the size of the rules and the overlap between them vary, we implemented two sub-scenarios in FRuG : a “small” sub-scenario with little overlap and mostly small rules (best case) and a “large” sub-scenario with a lot of overlap and a wide variety of rule sizes, including wildcards (worst case). None of the generated rules is an exact match rule, thereby, we really focused on complex rule sets. Regarding ClassBench, three different seeds were used, representative of three scenarios : Firewall (FW), Access List (ACL), and IP Chain (IPC), such as those used in Fong *et al.* (2012) and Vamanan *et al.* (2010). However, the SDN context allows a fine-grained processing, and, in order to illustrate complex processing, we created a fourth category, as a mix of IPC, FW, and ACL rules. We generated classification tables with 10 000 rules and 100 000 rules. This configuration led us to the four scenarios, with each two sub-scenarios, as benchmarks for performance measurement.

5.6.3 Measurements

In this section, we expose the other hypotheses made. First, we use a bus width w of 18 bytes, so that each node requires two memory accesses, as exposed in Stimpfling *et al.* (2013). As mentioned in the previous paragraph, our measurement process is focused on two scenarios : worst and best case. Regarding EffiCuts, it is pretty simple to define our best and worst cases. For the worst case, we picked-up the deepest leaf which held as many rules for the worst case, and the fewer rules in the best case, for each tree generated. The best and worst case measure of AcceCuts performance are obtained trough two metrics S_{min} and S_{max}. The first metric, S_{min}, identifies, at the deepest level of the tree, which leaf will lead to the lowest number of memory accesses, by selecting the leaf minimizing the number of sub-rule groups, S_{ij}, while having the fewest number of undifferentiated sub-rules. The S_{max} metric acts to identify which leaf will result in the highest number of memory accesses. For this purpose, S_{max} selects the deepest leaf in the tree while maximizing the number of undifferentiated sub-rules, and the number of sub-rule groups S_{ij}.

In order to evaluate AcceCuts and EffiCuts scalability, we measure the data structure size generated. However, the leaf structure modification introduced in AcceCuts generates memory overhead over the regular leaf structure used by EffiCuts, as a leaf now holds, at the same time, the rules and a large header used for preprocessing. Therefore, we want to evaluate the extra memory used to store the new header. Thus, we observe the memory consumption for EffiCuts, for AcceCuts without leaf processing (EffiCuts leaf), and, finally, AcceCuts.

Another critical parameter is linked to the number of comparisons needed to classify a packet. Therefore, after the previous theoretical study, we evaluate the number of comparisons with our benchmarks in the best and worst cases. AcceCuts without leaf structure modification shares the same leaf structure as EffiCuts, and the same number of comparisons. This is why only AcceCuts and EffiCuts will be taken into account in our practical evaluation.

The performance reached by AcceCuts reflects mainly the ability to differentiate rules into sub-rule groups holding only sub-rules disjoint two by two. Therefore, to better understand memory accesses results, it is important to evaluate the maximum number of sub-rule groups generated, or the maximum number of undifferentiated sub-rules. Indeed, memory access measurement is based on the worst case scenario, and is impacted by those two parameters.

5.7 Experimental results

5.7.1 Data structure size

For the first part, our experimentation is focused on the size of the data structure generated. Like in EffiCuts, we adopt the same metric, bytes per rule, to characterize memory

requirements. Bytes-per-rule measurement are obtained by dividing the total data structure size by the number of rules considered, which is a straightforward approach to isolate how much memory is used to store just the data structure (nodes and leaf header), as the rule size is fixed to 41 bytes.

In Fig. 5.7, we observe that the number of bytes per rule for EffiCuts is ranging from 77 to 90, independently of the scenario considered (IPC, ACL, FW), for classification tables with 100 000 rules. Considering that a rule weights 41 bytes, just the node structure represents roughly 50% of the memory used. By contrast, AcceCuts without leaf structure modification generates a data structure where that bytes per rule value is bounded between 42 and 53 bytes. Therefore, our algorithm is close to the minimal memory consumption we can reach, which is the rule size here. Now, if we take a closer look at the overhead introduced by changing the leaf structure, we observe that the memory consumption is increased by 14% over AcceCuts without leaf structure modification. Even though extra memory is consumed by the proposed leaf structure of Accecuts leaves, its overall memory utilization is still 33% lower than EffiCuts.

When considering a table with 10 000 rules, we observe a similar distribution, leading to the same conclusions. AcceCuts generates a data structure smaller by 33% over EffiCuts, even with the new leaf structure, and the new leaf introduces an overhead below 15% over a regular EffiCuts leaf structure.

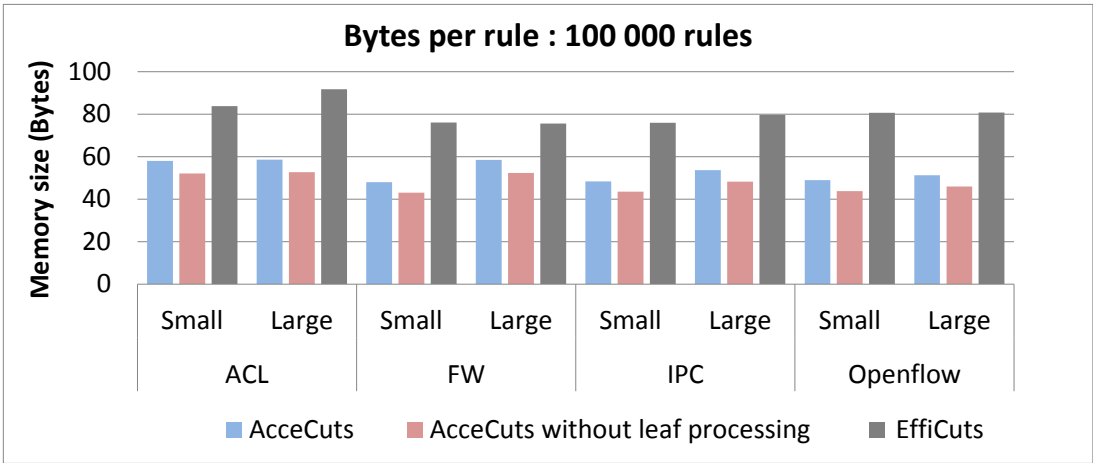


Figure 5.7 Bytes per rule for 100K rules

The two following sections characterize the number of memory accesses and the number of comparisons. As mentioned in the theoretical analysis section, the number of memory accesses and also the number of comparisons are directly linked to the number of sub-rule groups and undifferentiated rules. For the following two sub-sections, our explanations will be

based on the maximum number of undifferentiated sub-rules, and on the maximum number of sub-rules groups generated by AcceCuts. For this purpose, we will use Fig. 5.9 and 5.10.

5.7.2 Number of memory accesses

In Fig. 5.8, we report results obtained with AcceCuts, with and without leaf structure processing, and with EffiCuts for the 100 000-rule benchmarks.

We find that AcceCuts is less sensitive to table types compared to EffiCuts as the variation range for AcceCuts is more limited compared to EffiCuts. In the worst case, AcceCuts allows to classify a packet in a maximum of 252 memory access for every context presented here, independently of the number of rules taken into account. Moreover, even though our analytic model predicted similar performance in the worst case, and improved performance in the best case, we observed that AcceCuts offers a major improvement over EffiCuts for both cases.

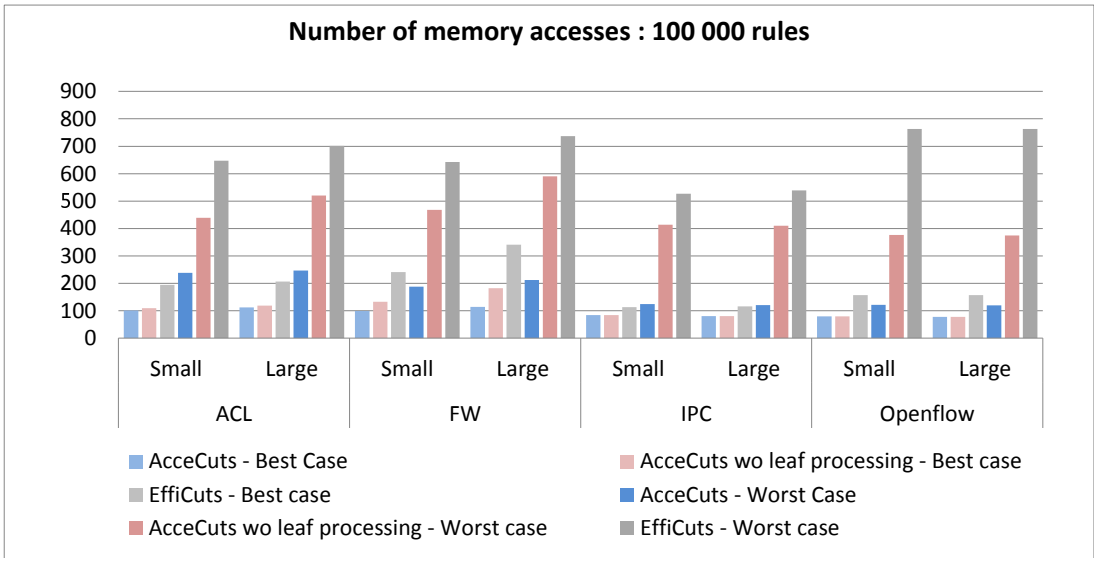


Figure 5.8 Memory Accesses for 100K rules

In Fig. 5.8, we observe the effectiveness of leaf processing introduced in our algorithm; AcceCuts with leaf structure modification cuts down the number of memory accesses over AcceCuts without leaf structure modification by 62,5% in the worst case, and 15% in the best case considered. On average, AcceCuts decreases the number of memory accesses by a factor of 3.2 in the worst case considered, and by 2.3 in the best case, over EffiCuts.

However, AcceCuts performs more efficiently when dealing with IPC and OpenFlow-like classification tables. For those two scenarios we observed in 5.10 that no rule was undifferentiated, and the maximum number of categories (see Fig. 5.9) was either very low for IPC,

as only one sub-rule group was created, or low for OpenFlow-like table with a number of sub-rule groups of 3.

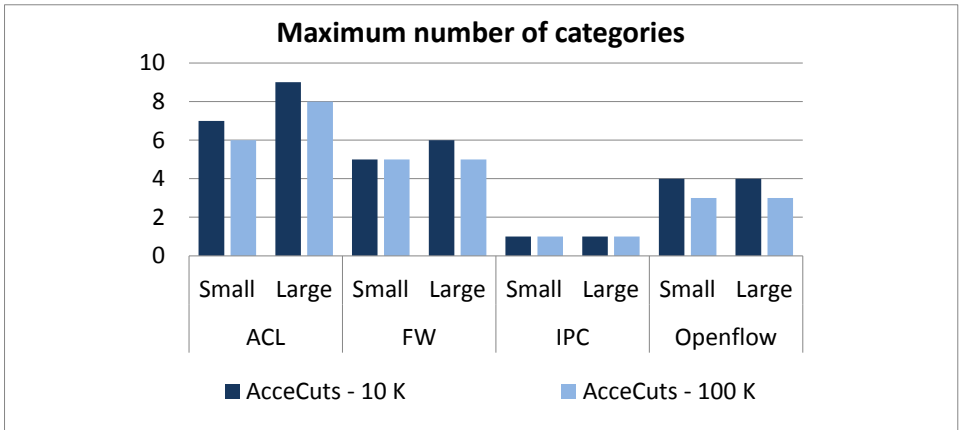


Figure 5.9 Measurement of maximum number of undifferentiated rules in all S_{ij} groups

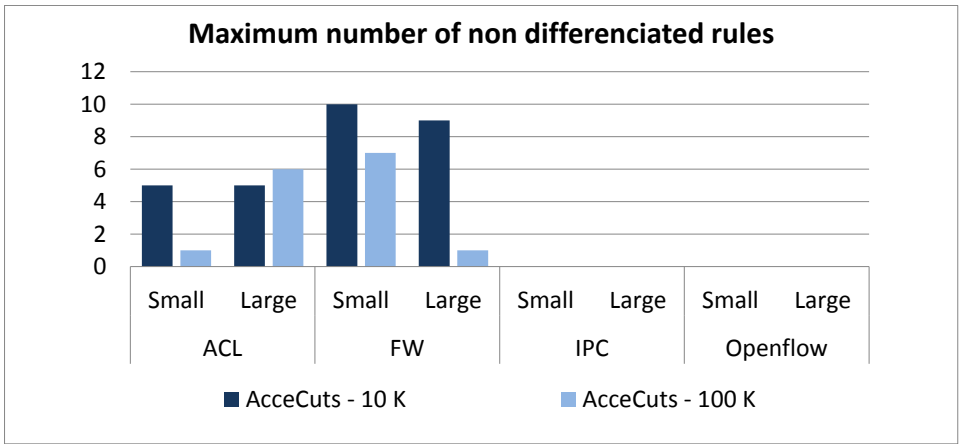


Figure 5.10 Measurement of maximum number of overlapping sub-rules

By contrast, we observe in Fig. 5.8 a larger number of memory accesses for the ACL and FW scenarios, for both worst case and best case. This performance variation is a consequence of the larger number or undifferentiated rules (up to 7 rules for the FW scenario, and up to 6 for ACL), combined with a large number of sub-rule groups. Indeed, as shown in Fig. 5.9, FW scenarios generates 5 sub-rule groups, for the Small and Large sub-scenarios, whereas Small and Large sub-scenarios for ACL are generating respectively 6 and 8 sub-rule groups.

When considering smaller classification tables, with 10 000 rules, we arrived at the same conclusions. Since the results and the conclusions are similar, we only present a summary of our observations for the 10K-rule case.

AcceCuts with leaf structure modification reduces the number of memory accesses over AcceCuts without leaf structure modification by 65% in the worst scenario, and 35% in the best scenario considered. However, even without leaf modification, AcceCuts requires fewer memory accesses than EffiCuts, by a factor of 2 in the best case, but also in the worst case. On average, AcceCuts decreases the number of memory accesses by a factor of 3.9 in the worst case considered, and by 3.7 in the best case, over EffiCuts.

Again, the results obtained with 10 000 rules experiments are very similar to those obtained with 100 000 rules as reported in Fig. 5.9 and 5.10.

As a conclusion, regarding the number of memory accesses, we have shown that AcceCuts performs better over EffiCuts in both best and worst cases. Nevertheless, the best improvement is made for the worst case, with a reduction factor of 3 over EffiCuts. Improving the worst case is a main concern regarding memory accesses and AcceCuts fulfills one critical requirement for hardware implementation. Even though we put emphasis on the worst case, AcceCuts, still uses around half of the number of memory accesses required by EffiCuts in best case scenarios.

5.7.3 Number of comparisons

Through the two simulations presented before, we have presented how AcceCuts compensates for EffiCuts issues. To achieve this performance, we leveraged three contributions. In this paragraph, we argue that AcceCuts, even with leaf structure modification, introduces the same number or fewer comparisons as needed in EffiCuts to complete a leaf traversal.

Independently of the number of rules considered, we can first see in Fig. 5.11 that the leaf structure introduced in AcceCuts does not create an explosion of the number of comparisons, for both best cases and worst cases, as predicted by our analytic model. Moreover, the number of comparisons for AcceCuts in the worst case remains equal to those expected with EffiCuts. In fact, the worst case is represented for EffiCuts by a number of 12 rules held in a leaf, which generates a number of comparisons equals to the number of rules multiplied by the number of fields considered. By contrast, in the best case of EffiCuts, a leaf holds only one rule, which lead to a number of comparisons equal to the number of fields considered.

Now, if we consider classification tables with 100 000 entries, as reported in Fig. 5.11, we observe, in the best case, that EffiCuts requires one comparison less than AcceCuts, whereas, for the worst case, our algorithm generates only half the number of comparisons needed in EffiCuts. Only the Firewall scenario leads to a number of comparisons close to the one observed in EffiCuts, but it remains lower. For others scenarios, AcceCuts clearly out-performs EffiCuts, and reaches the best results for IPC and OpenFlow scenarios.

The results observed in Fig. 5.11 reflect the observation made in Fig. 5.9 and 5.10, as

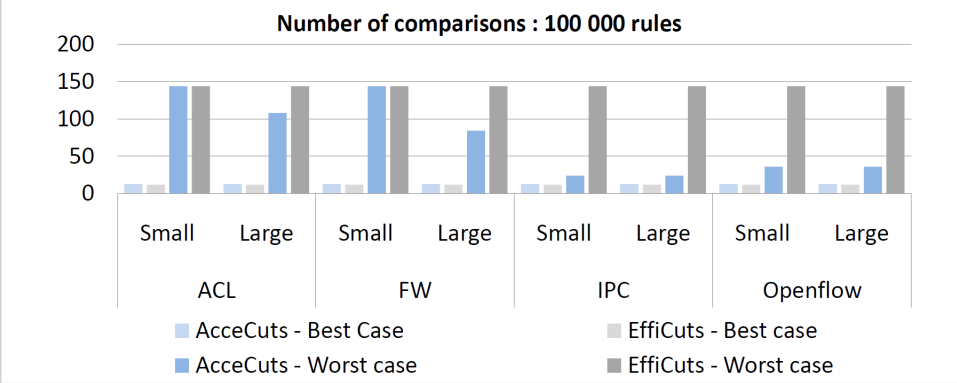


Figure 5.11 Number of comparisons needed for 100K rules

the number of comparisons is directly impacted by two parameters : the maximum number of sub-rule groups and the maximum number of undifferentiated sub-rules. Since, for IPC and OpenFlow scenarios, the value of both parameters is pretty low (no undifferentiated sub-rules, and a small number of sub-rule groups), the number of comparisons in the worst case is very limited. In the best case, AcceCuts needs an extra comparison, as it first compares the sub-rule in the header, and then only retrieves the associated rule.

We then simulated the number of comparisons made when AcceCuts handles 10 000 rules. In this case, the same conclusions can be made as what has been observed with the 100 000 rules. Thus, in the best case, AcceCuts remains behind EffiCuts by only one comparison but, in the worst case, AcceCuts cuts down on the average number of comparisons required, but, in this case, only the FW scenario is close to EffiCuts performance. Nevertheless, IPC, ACL, and OpenFlow scenarios generate much fewer comparisons in AcceCuts, as opposed to EffiCuts. The observations made here support our conclusions made in the theoretical study. Our results are supported with Fig. 5.9 and 5.10, as explanations given in the previous paragraph remains valid.

5.7.4 Versatility

Even if our algorithm has been optimized for the SDN context, with a large number of fields, it is interesting to evaluate the performance for the 5-tuple context. In this case, we make the assumption that a rule size is 19 bytes Yeim-Kuan et Chao-Yen (2012), that the bus width remains unchanged, and that classification tables are generated using the same methodology as exposed in Vamanan *et al.* (2010).

In simulation, we observed, regardless of scenarios and number of rules considered, that the number of memory accesses reached by AcceCuts in the worst case is slightly better than in EffiCuts. On average, AcceCuts cuts down the number of memory accesses over EffiCuts

by 18%. Conversely, AcceCuts generates an increase of 5% in the best case over EffiCuts. Here, the rule size is half the size of what was reported in the 12-tuple context, leading to a smaller number of memory accesses to complete leaf traversal, which reduces the impact of numerous irrelevant memory access introduced by EffiCuts and older decision tree based algorithms.

Regarding the size of the data structure generated per rule, AcceCuts reduces this parameter for every scenario simulated, on average, by 48% over EffiCuts. We observe a variation between the scenarios simulated, as the number of bytes per rule ranges from 25 to 38, depending on the complexity of the scenario used, whereas EffiCuts ranges between 50 and 61 bytes per rule. Improvements obtained with AcceCuts are mainly due to the new heuristic employed to compute the number of cuts in AcceCuts.

5.8 Conclusion

In this paper, some shortcomings of EffiCuts were highlighted in the SDN context ; EffiCuts does not scale well with the number of fields considered, and achieves poor performance when considering complex rules, such as with 12-tuple rules. The performance degradation with EffiCuts is mostly due the large number of memory accesses rather than the size of the data structure. Our algorithm, AcceCuts, is designed to reach a higher performance over EffiCuts, and to reduce the number of memory accesses and the data structure size. First, we propose the Adaptive Grouping Factor, a method which parses the rule-set properties and bins rules accordingly, in an optimized manner. Secondly, we adopted a heuristic generating a small sized data structure without impacting on the data structure depth. Finally, we proposed a leaf processing modification, in order to cut down the number of memory accesses over EffiCuts. These extensions provide gains in many contexts but they were tailored for the 12-tuple rule context.

AcceCuts gives a better performance compared to EffiCuts when the size of the rule becomes larger than a node size. It is shown, using suitable benchmarks, that they allow reducing the number of memory accesses by a factor of 3 on average, while decreasing the size of the data structure by about 45%. AcceCuts is an algorithm that improves two partly opposing parameters at the same time, while remaining versatile. The three innovations proposed in AcceCuts make it more suitable for larger classification tables applied over a larger number of fields without excessive performance degradation. It seems promising for the design of a high-speed hardware classification unit.

CHAPITRE 6

DISCUSSION GÉNÉRALE

L'article présenté dans le chapitre, 4, a permis d'affirmer que l'algorithme EffiCuts, transposé dans le contexte de SDN ne permet pas d'atteindre des performances décentes. En effet, malgré une consommation de mémoire raisonnable, le nombre d'accès mémoire a tout simplement explosé. Ce problème est majoritairement du au nombre élevé d'arbres à parcourir, conséquence directe de l'augmentation du nombre de champs considérés.

Par ailleurs, il est aussi important de remarquer que, comparé à (Vamanan *et al.*, 2010), où seuls 5 champs sont considérés, la consommation mémoire de l'algorithme présenté dans le chapitre 4 demeure dans le même ordre de grandeur. En effet, l'algorithme EffiCuts fait levier sur de multiples méthodes pour minimiser la valeur du facteur de réplication et ainsi minimiser l'espace occupé par chaque règle, de telle sorte que l'augmentation de la taille des règles a un impact limité au niveau de la taille de la structure générée. De plus, les nombreux arbres construits dû à la prise en compte de 12 champs, sont majoritairement de petits arbres, contenant un nombre faible voir très faible de règles. Par conséquent, ce type d'arbre a une influence très restreinte au niveau de la taille de la structure de donnée générée. Ces deux justifications permettent d'expliquer les performances rencontrées par l'algorithme EffiCuts, dans sa version non optimisée, dans un contexte de SDN. Bien que l'extensibilité soit bonne en termes d'occupation mémoire, l'extensibilité de cet algorithme est très mauvaise concernant le nombre d'accès mémoire.

Les trois optimisations présentées dans (Stimpfling *et al.*, 2013) permettent de réduire de manière importante la consommation mémoire ainsi que le nombre d'accès mémoire et d'atteindre une performance acceptable pour envisager une implémentation matérielle ou logicielle. Ainsi le nombre d'accès mémoire moyen a été réduit par un facteur 2, alors que la consommation mémoire est réduite de 35% en moyenne par rapport à EffiCuts.

Néanmoins, on remarquera que l'optimisation "Leaf size modulation" est une optimisation partielle, dans la mesure où, dans le contexte présenté, on constate que ce paramètre a une influence sur la performance et notamment le nombre d'accès mémoire mais que la performance est sujette à variation. Dès lors, le problème sous-jacent, à savoir le parcours linéaire des feuilles, n'est pas traité. Ce point amène deux autres questions ; *Quelle partie des arbres "comprimer" pour diminuer le nombre d'accès mémoire ?*et *Peux-t-on proposer une alternative au parcours linéaire des feuilles ?*. Le chapitre 5 vise à apporter un éclairsissement à ces questions.

Le second article présenté dans le chapitre 5, permet de pousser la reflexion plus loin, via une analyse détaillées de la performance et des problèmes rencontrés par EffiCuts dans le contexte de SDN. La solution proposée, AcceCuts va beaucoup plus en loin en termes de performance par rapport à ce qui a été proposé dans le chapitre 4. En effet, la réduction moyenne du nombre d'accès mémoire atteind un facteur 3, et une diminution de 45% de l'occupation mémoire est obtenue par rapport à EffiCuts. La grande différence entre ces deux chapitre repose sur la modification de la structure des feuilles introduite par AcceCuts, et qui impact drastiquement sur le nombre d'accès mémoire en impactement de manière très limitée sur la taille de la structure de donnée générée. Il est par ailleurs intéressant de noter que jusqu'à présent, dans la littérature, aucune contribution n'a été faite concernant la structure des feuilles.

Par ailleurs, l'article (Stimpfling *et al.*, 2013) mentionne qu'en raison de l'absence de générateur de règles synthétiques reprenant des caractéristiques de tables de classification réelles, une méthode de génération de règles a été proposée combinant deux générateurs de règles, l'un, ClassBench permettant de créer des règles avec des typages et l'autre, FRuG, dont les paramètres de génération sont laissés à la discrétion de l'utilisateur. Dans un tel cas, il parait impossible de configurer FRuG pour recréer des patrons de règles utilisées réellement dans des tables de classification. Certains sites offrent, à des fins de recherche, des traces de paquets (CAIDA : The Cooperative Association for Internet Data Analysis) mais ces traces ne permettent pas de déterminer les caractéristiques des règles associées qui ont filtré ces paquets. Par conséquent, la solution retenue a été de configurer FRuG afin d'encadrer la performance de l'algorithme testé, en créant deux scénarios permettant de simuler le meilleur ainsi que le pire des cas. Néanmoins, une telle approche est discutable, non pas sur le principe, mais bien sur les paramètres qui ont été associés au meilleur et pire cas. Autant, il est peu important d'avoir un "meilleur cas" éloigné (en dessous) du "meilleur cas" réel, dans la mesure où le seul risque est de ne pas obtenir des résultats potentiellement meilleurs. Autant un scénario illustrant le pire des cas et configuré de manière trop avantageuse peut être lourd de conséquence. En effet, dans la littérature la performance des algorithmes est souvent étudiée par rapport au pire des scénarios, afin de garantir des performances, et ce sont ces mesures qui permettent de déterminer si oui ou non l'algorithme est digne d'intérêt ou non.

De même, la situation inverse est aussi envisageable, à savoir que les paramètres entrés pour configurer le générateur de règles représentent certes un scénario engendrant la pire performance d'un point de vue théorique, mais rien ne dit que cette situation va être rencontrée dans un contexte de production. De fait, le pire des scénarios simulés peut se situer très loin du pire des scénarios réels et, donc, les performances réelles peuvent s'avérer sensiblement meilleures que celles simulées. Il est important de considérer ces faits et de rappeler que la

seule simulation valable demeure celle effectuée avec des règles utilisées dans un environnement de production.

Néanmoins, ces deux remarques demeurent valables dans le cas de l'algorithme AcceCuts présenté dans le chapitre 5, puisque ce dernier se base sur les mêmes benchmark.

CHAPITRE 7

CONCLUSION

7.1 Synthèse des travaux

L'algorithme AcceCuts proposé ici fait suite à un travail préliminaire présenté dans la première partie de ce mémoire, dont le but visait à adapter un algorithme de classification de paquet au contexte de SDN, et de visualiser la performance obtenue avec des règles de type OpenFlow v1.0.0.

À cet égard, il a été présenté que des algorithmes basés sur des arbres de décision peuvent offrir une performance acceptable dans le contexte de SDN, mais que plusieurs modifications sont nécessaires. En effet, le niveau de performance de l'algorithme EffiCuts observées dans le chapitre 4 n'est pas acceptables. Les trois optimisations présentées dans chapitre 4 permettent de réduire en moyenne la consommation mémoire de 35%, tout en réduisant le nombre moyen d'accès mémoire par un facteur 2. Cette amélioration des performances est importante, mais aucune remise en question de la structure de donnée utilisée n'est effectuée.

La seconde partie du travail, exposé dans le chapitre 5, vise à identifier les causes profondes de la sous-performance relatif au nombre d'accès mémoire. Par conséquent, une seconde phase d'analyse est ici effectuée, et sert de base pour comprendre certains défauts non traités dans le chapitre 4. Le problème principale identifié, est lié au parcours des feuilles, et plus particulièrement à la lecture de règles qui dans de nombreux cas, ne correspondent pas aux caractéristiques du paquet d'entrée.

De ces observations ont découlé trois modifications, dont la plus importante consiste en une modification de la structure des feuilles utilisées. Les modifications introduites au niveau de la feuille dans AcceCuts permettent ainsi de réaliser un prétraitement lors du parcours de la feuille et de réaliser des comparaisons sur un nombre restreint de règles, ce qui diminue drastiquement le nombre d'accès mémoire à effectuer, tout en ayant un surcout (stockage, nombre de comparaisons à effectuer) très limité.

Comme mentioné dans le chapitre 5, AcceCuts au travers des trois optimisations, a permis de réduire par rapport à EffiCuts, le nombre moyen d'accès mémoire par un facteur 3, tout en réduisant la taille de la structure de donnée par 45% en moyenne, dans un contexte de type SDN. De plus, AcceCuts, malgré une différence de performance réduite avec EffiCuts dans un contexte traditionnel, conserve néanmoins un gain par rapport à ce dernier. L'algorithme AcceCuts est donc utilisable dans un nombre varié de contextes, et permet d'atteindre une

performance élevée lorsque le nombre de champs considérés augmente.

Au-delà de la performance brute atteinte, il est intéressant de constater d'une part l'amélioration de la performance selon deux métriques jusque-là relevant d'un compromis, à savoir la consommation mémoire et le nombre d'accès mémoire. D'autre part, les modifications apportées au niveau des feuilles permettent de briser l'influence de la taille des feuilles sur le nombre d'accès mémoire, de telle sorte qu'il est dorénavant possible d'avoir la taille d'une structure de donnée générée avec 16 règles maximum par feuille, tout en ayant une performance proche de ce qui aurait été offerte en considérant un maximum de 4 règles par feuille.

Afin d'évaluer la performance des différents algorithmes, avec et sans optimisations, un benchmark est proposé, dont le but n'est pas tant de recréer des règles proches de situations réelles, mais davantage de générer des règles dont les caractéristiques permettent d'encadrer la performance des algorithmes dans le meilleur ainsi que dans le pire des cas.

7.2 Limitations de la solution proposée

Une des limitations majeures du travail réalisé ici concerne le benchmark utilisé. En effet, les règles utilisées sont des règles synthétiques, et bien que la configuration adoptée ai été effectuée pour générer des règles représentant le pire et le meilleur des scénario, ces règles demeurent synthètiques, et ne représentent pas une situation réelle. Par conséquent il est difficile d'apprécier la différence de performance lié à l'utilisation des règles réelles par rapport aux règles utilisées dans le cadre de ce travail.

De même, le domaine SDN n'est pas encore assez mature pour identifier clairement des nouveaux scénarios, applications qui impacteront sur l'étape de classification de paquet. Par conséquent, bien que des tendances se dégagent pour certains paramètres, la complexité, la taille, ainsi que le nombre de champs utilisés réellement demeurent des inconnues.

Enfin AcceCuts a été développé selon le contexte d'étude précisé dans la revue de littérature. Ainsi, la capacité de l'algorithme à effectuer des mises à jour a été écartée.

Or, ce problème est particulièrement préoccupant dans le cadre du SDN, puisque la gestion du réseau se fait de manière dynamique, ce qui implique des modifications en temps réel des tables de classification, et donc des structures de données utilisées pour classifier les paquets entrant.

Ainsi, le support des mises à jour pour l'algorithme AcceCuts devrait être l'objet d'un travail futur.

7.3 Améliorations futures

Comme mentionné dans le chapitre 2, les algorithmes héritant de l'approche introduite par (Gupta et McKeown, 2000) souffrent d'un potentiel problème de mise à jour. Ainsi, jusqu'à présent aucune méthode de mise à jour sérieuse n'a été proposé dans la littérature (Voir l'analyse présentée dans le chapitre 2).

Si aucune méthode de mise à jour n'est viable, par défaut, il est nécessaire de reconstruire intégralement la structure de donnée. Par conséquent, l'insertion ou la surpression de règles demeure un problème ouvert.

La où EffiCuts nécessitait plusieurs heures pour créer une structure de donnée associée à une table de classification contenant 100000 règles, AcceCuts réduit le temps de construction à une dizaine de minutes. Il faut néanmoins préciser que le code utilisé sert uniquement à effectuer des mesures des différents paramètres présentés dans (Stimpfling *et al.*, 2013). Or, le temps de construction des arbres n'a pas été évalué comme métrique, en partie parce que l'ensemble du code aurait dû être repensé afin d'atteindre des performances décentes.

En effet, le code actuel utilise des conteneurs stl : :list, introduisant une lourdeur dans les traitements associés à la création des arbres. Par ailleurs, bien que les traitements employés dans la construction des arbres soient facilement parallélisables, aucune méthode de calcul parallèle n'y est implémentée.

Il est donc nécessaire d'attaquer le problème des mises à jour en considérant deux dimensions à la fois ; la parallélisassion du traitement, pour faire levier sur la puissance de calcul et repenser des méthodes de mises à jour minimisant la reconstruction de la structure de donnée.

Par ailleurs, au-delà de la vitesse de construction des arbres, la plus grande incertitude demeure les caractéristiques des mises à jour : vitesse, nombre, répartition ; autant d'éléments qui peuvent lourdement impacter la performance et nécessite des réponses adaptées. Par conséquent, les travaux futurs devront tenir compte de la diversité des contextes pouvant être rencontrés. Par ailleurs, la question des mises à jour devra aussi être prise en compte au niveau de l'architecture matérielle, afin de minimiser les transferts de données.

Le contexte entourant le SDN est peu mature pour comprendre en intégralité les défis amenés. Peu d'implémentation à large échelle ont été faites jusqu'à présent (au délà de grand nom de l'informatique) ; de fait peu d'information au-delà des publications universitaires sont disponibles. Au niveau global (voir la figure 1.1), de nombreuses contributions ont été faites au niveau logiciel et de nombreux codes sont disponible sous licence GPL, néanmoins au niveau ‘’infrastructure” peu de matériel spécifique au contexte SDN n'a encore atteint un volume de production élevé.

Par conséquent, de nombreux problèmes demeurent ouverts, tel que le support de la virtualisation au niveau matériel, le type de plateforme matérielle à privilégier pour atteindre le meilleur compromis entre flexibilité et performance.

RÉFÉRENCES

BABOESCU, F. et VARGHESE, G. (2001). Aggregated bit vector search algorithms for packet filter lookups. Rapport technique, La Jolla, CA, USA.

BECKMANN, C., TONSING, J. et MACK-CRANE, B. (April 4, 2013). Charter : Forwarding abstractions working group. Rapport technique.

BO, X., DONGYI, J. et JUN, L. (2005). Hsm : a fast packet classification algorithm. *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on.* vol. 1, 987–992 vol.1.

CHAO, H. J. et LIU, B. (2007). *High Performance Switches and Routers.* Wiley.

DHARMAPURIKAR, S., HAOYU, S., TURNER, J. et LOCKWOOD, J. (2006). Fast packet classification using bloom filters. *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on.* 61–70.

FELDMAN, A. et MUTHUKRISHNAN, S. (2000). Tradeoffs for packet classification. *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.* vol. 3, 1193–1202 vol.3.

FONG, J., XIANG, W., YAXUAN, Q., JUN, L. et WEIRONG, J. (2012). Parasplit : A scalable architecture on fpga for terabit packet classification. *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on.* 1–8.

GANEGEDARA, T., WEIRONG, J. et PRASANNA, V. (2010). Frug : A benchmark for packet forwarding in future networks. *Performance Computing and Communications Conference (IPCCC), 2010 IEEE 29th International.* 231–238.

GOOGLE (2012). Inter-datacenter wan with centralized te using sdn and openflow. Rapport technique.

GUPTA, P. et MCKEOWN, N. (1999). Packet classification on multiple fields. *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication.* ACM, New York, NY, USA, SIGCOMM '99, 147–160.

GUPTA, P. et MCKEOWN, N. (2000). Classifying packets with hierarchical intelligent cuttings. *Ieee Micro*, 20, 34–41.

GUPTA, P. et MCKEOWN, N. (2001). Algorithms for packet classification. *IEEE Network*, 15, 24–32.

HUAN, L. (2002). Efficient mapping of range classifier into ternary-cam. *High Performance Interconnects, 2002. Proceedings. 10th Symposium on.* 95–100.

JIANG, W. et PRASANNA, V. K. (2009). Large-scale wire-speed packet classification on fpgas. *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, New York, NY, USA, FPGA '09, 219–228.

JIANG, W. R. et PRASANNA, V. K. (2012). Scalable packet classification on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20, 1668–1680.

LAKSHMAN, T. V. et STILIADIS, D. (1998). High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28, 203–214.

NARAYAN, H., GOVINDAN, R. et VARGHESE, G. (2003). The impact of address allocation and routing on the structure and implementation of routing tables. *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, New York, NY, USA, SIGCOMM '03, 125–136.

OPEN NETWORKING FOUNDATION (2009). *The OpenFlow Switch Specification v1.0.0*.

OPEN NETWORKING FOUNDATION (April 13, 2012). *Software-Defined Networking : The New Norm for Networks*.

OPEN NETWORKING FOUNDATION (April 2, 2013). *The OpenFlow Switch Specification Version 1.3.2*.

OVERMARS, M. H. et VAN DER STAPPEN, A. F. (1994). Range searching and point location among fat objects. *Journal of Algorithms*, 21, 629–656.

SINGH, S., BABOESCU, F., VARGHESE, G. et WANG, J. (2003). Packet classification using multidimensional cutting.

SONG, H. (2006). Test on tuple space search.

SONG, H. et TURNER, J. S. (2013). Abc : adaptive binary cuttings for multidimensional packet classification. *IEEE/ACM Trans. Netw.*, 21, 98–109.

SOURDIS, I. et SOURDIS, I. (2007). Designs & algorithms for packet and content inspection.

SRINIVASAN, V., SURI, S. et VARGHESE, G. (1999). Packet classification using tuple space search. *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, New York, NY, USA, SIGCOMM '99, 135–146.

SRINIVASAN, V., VARGHESE, G., SURI, S. et WALDVOGEL, M. (1998). Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28, 191–202.

STEPHEN, L. (2010). Facebook sees need for terabit ethernet.

STIMPFLING, T., SAVARIA, Y., BÉLIVEAU, A., BÉLANGER, N. et CHERKAOUI, O. (2013). Optimal packet classification applicable to the openflow context. *Proceedings of the first edition workshop on High performance and programmable networking*. ACM, New York, NY, USA, HPPN '13, 9–14.

TAYLOR, D. E. (2005). Survey and taxonomy of packet classification techniques. *Acm Computing Surveys*, 37, 238–275.

TAYLOR, D. E. et TURNER, J. S. (2004). Scalable packet classification using distributed crossproducting of field labels.

TAYLOR, D. E. et TURNER, J. S. (2007). Classbench : A packet classification benchmark. *Ieee-Acm Transactions on Networking*, 15, 499–511.

VAISH, N., KOOBURAT, T., DE CARLI, L., SANKARALINGAM, K. et ESTAN, C. (2011). Experiences in co-designing a packet classification algorithm and a flexible hardware platform. *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM IEEE Symposium on*. 189–199.

VAMANAN, B., VOSKUILEN, G. et VIJAYKUMAR, T. N. (2010). Efficuts : Optimizing packet classification for memory and throughput. *Computer Communication Review*, 40, 207–218.

WANG, Z., CHE, H., KUMAR, M. et DAS, S. K. (2004). Coptua : Consistent policy table update algorithm for tcam without locking. *IEEE Trans. Comput.*, 53, 1602–1614.

YAXUAN, Q., LIANGHONG, X., BAOHUA, Y., YIBO, X. et JUN, L. (2009). Packet classification algorithms : From theory to practice. *INFOCOM 2009, IEEE*. 648–656.

YEIM-KUAN, C. et CHAO-YEN, C. (2012). Layer partitioned search tree for packet classification. *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*. 276–282.