Conception et évaluation des systèmes logiciels de classifications de paquets haute-performance

Peng He

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

préparée dans le cadre d'une cotutelle entre l'*Université Grenoble Alpes* et L'académie des Sciences de Chine

Spécialité : **Informatique**

Arrêté ministériel : le 6 janvier 2005 - 7 août 2006

Présentée par

Peng HE

Thèse dirigée par **Kavé Salamatian** et **Gaogang Xie**

préparée au sein des **Laboratoires LISTIC et Institut des technologies Informatique de l'académie des sciences de Chine**
dans **l'École Doctorale SISEO**

Conception et évaluation des systèmes logiciels de classifications de paquets haute-performance

Thèse soutenue publiquement le **02 MAI 2015**,
devant le jury composé de :

Mr Thierry TURLETTI
Directeur de Recherche à l'INRIA Sophia-Antipolis, Président et Rapporteur
Mr Serge FDIDA
Professeur à l'université Pierre et Marie Curie, Rapporteur
Mr Laurent Mathy
Professeur à l'université de Liège, Membre
Mr Steve Uhlig
Professeur à l'université Queen Mary de Londres, Membre
Mr Kavé Salamatian
Professeur à l'université de Savoie , directeur de thèse
Mr Gaogang Xie
Professeur à l'académie des sciences de Chine, directeur de thèse

1. Motivations

La classification de paquets est à la base de nombreux services avancées dans les réseaux tels que les pare-feux, la détection et la prévention d'intrusion, l'équilibrage de charge, la gestion de la Qualité de service, *etc.* Cette opération consiste généralement en la comparison de champs d'entête des paquets par rapport à un ensemble de règles prédéfinies et à l'application des actions associées à ces règles aux paquets validant celles-ci. Au vu de son importance, la classification haute performance de paquets a été extensivement étudié durant les dix dernières années. Les approches classiques de classification sont généralement fondées sur des solutions matérielles dédiées. Mais, récemment les systèmes de classification logiciels ont attiré un intérêt croissant [1,2,3]. Dans la suite nous décrirons les raisons de cet intérêt.

L'émergence des SDNs

Les réseaux logiciels, défini par le sigle SDN (*Software defined Network*) sont des réseaux où les plans de contrôle et de données sont physiquement séparés. Des switches installés dans le cœur du réseau transfèrent les paquets en fonction de règles définies dans le plan de contrôle qui peut être distant. Cette séparation simplifie la gestion de réseaux étendus et complexes, et la définition de politiques de traitement dans ces réseaux. Néanmoins, pour que de telle solutions soient possible, il est nécessaire d'avoir un plan de données suffisamment flexible afin de transférer le plus rapidement possible ses différentes taches. L'architecture OpenFlow [4] est aujourd'hui le standard *de facto* qui donne les abstractions des fonctionnalités du plan de données. En suivant la spécification OpenFlow, un switch doit pouvoir appliquer une classification sur au moins 10 champs. Mais les matériels informatiques actuels ne peuvent pas gérer plus de quelques milliers de telles règles, *e.g.*, la taille de la mémoire TCAM du commutateur HP ProCurve 5406zl ne permet pas de stocker plus de 1500 règles. Cette limitation du nombre de règles, réduit sérieusement le déploiement des réseaux SDN. De plus la version 1.3 de l'architecture OpenFlow recommande jusqu'à 30 champs optionnels pour la classification. Cette multitude de champs impose un challenge technique important sur la conception des matériels pour SDN.

Dans [5] Martin Casado et ces co-auteurs proposent une solution pragmatique pour l'évolution des SDNs qui suggère de traiter le cœur du réseau et de l'accès de façon séparés. Le réseau d'accès se positionne à l'interface entre le réseau et les serveurs et clients terminaux et fourni des services réseaux comme la virtualisation, l'ingénierie de trafic, la qualité de service, *etc.* Le cœur de réseau, pour sa part ne transmet que des paquets à haute vitesse. Les auteurs de [5] proposent que le cœur et l'accès du réseau soient gérer par des contrôleurs indépendants. Similairement à MPLS, les routeurs du réseau d'accès ajoutent des étiquettes (*tags*) aux paquets qui sont transmis au cœur de réseau. Ces étiquettes sont utilisées dans le cœur de réseau pour simplifier la classification des paquets. Cette architecture est présentée dans la Figure 1.

Le volume de trafic dans le réseau d'accès est relativement plus faible, mais les besoins en classification de paquets y sont aussi les plus importants. C'est donc à ce niveau qu'il faut concentrer le déploiement de solutions flexibles de classification logicielle qui se fondent sur plusieurs champs d'entête. L'exemple typique de ceci est donnée par les « Open vSwitch » [6] qui échangent les paquets entrants entre plusieurs machines virtuelles en utilisant une classification fondée sur des règles OpenFlow. Ainsi la classification de paquets est un des éléments principaux des SDN et la performance des ces algorithmes conditionne la

performance de ces réseaux.

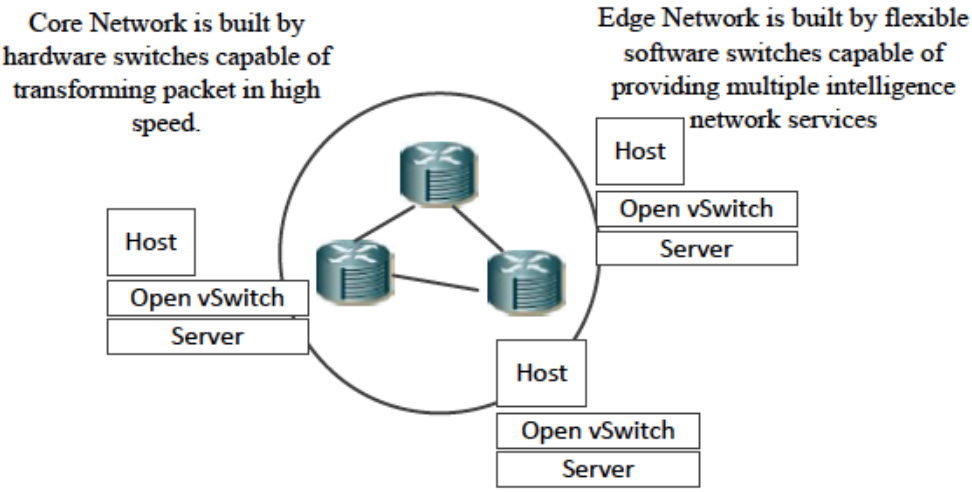


Figure 1- architecture de réseaux SDN

L'externalisation des services réseaux.

L'informatique dans le nuage (*Cloud computing*) a permis la séparation de la propriété de l'infrastructure matérielle et des applications s'exécutant sur ceux-ci. Les services réseaux sont parmi les services de bases fournis par l'infrastructure informatique. Il est donc pertinent d'explorer la possibilité d'externaliser les services réseaux et les fonctionnalités ci-rattachant dans le nuage. Ceci permettrait de faire bénéficier aux services réseaux les bienfait de la virtualisation dans le nuage. Il y'a déjà plusieurs entreprises pionnière, comme AT&T, qui proposent déjà pare-feux dans le nuage, réduisant de cette façon le CAPEX/OPEX des petites entreprises. Cisco a aussi produit une image de machine virtuelle implantant un pare-feu filtrant.

La classification de paquet est une composante essentielle de ces systèmes virtualisés, et elle doit être implanté de façon logicielle afin de pouvoir être déployé sur des matériels quelconque.

Le besoin de la virtualisation réseau

De plus en plus dans la pratique, ont observe des centres de données partagés entre plusieurs acteurs ayant chacun leur propre plan d'adressage. Dans le cadre du nuage et la virtualisation, le matériel exécutant un service n'est pas déterminé et fixé à l'avance et il peut changer d'emplacement. On peut donc dans ce genre de scénario avoir une architecture réseau par acteur dans le centre de données et il convient de virtualiser le réseau afin de pouvoir exécuter sur un seul support matérielle plusieurs routeurs relatifs aux divers acteurs du réseau. Dans [7], les auteurs montrent que dans ces scénarios, l'opérateur réseau a besoin de règles avec une granularité très fine afin de séparer le trafic réseaux entre les différentes propriétaires. Ces règles à grains fin consomment beaucoup de ressources matérielles et aboutissent même parfois à des pannes réseaux de grande ampleur [8]. Le débit d'addition de nouvelle règle peut atteindre 3000 changement par seconde et le nombre de règle à traiter dans ces scénarios peut facilement atteindre 60 K règles. Ceci aboutit à une consommation des ressources importantes et un besoin de mise à jour fréquentes de règles de classification qui n'est compatible qu'avec une implantation logicielle de la classification de paquets.

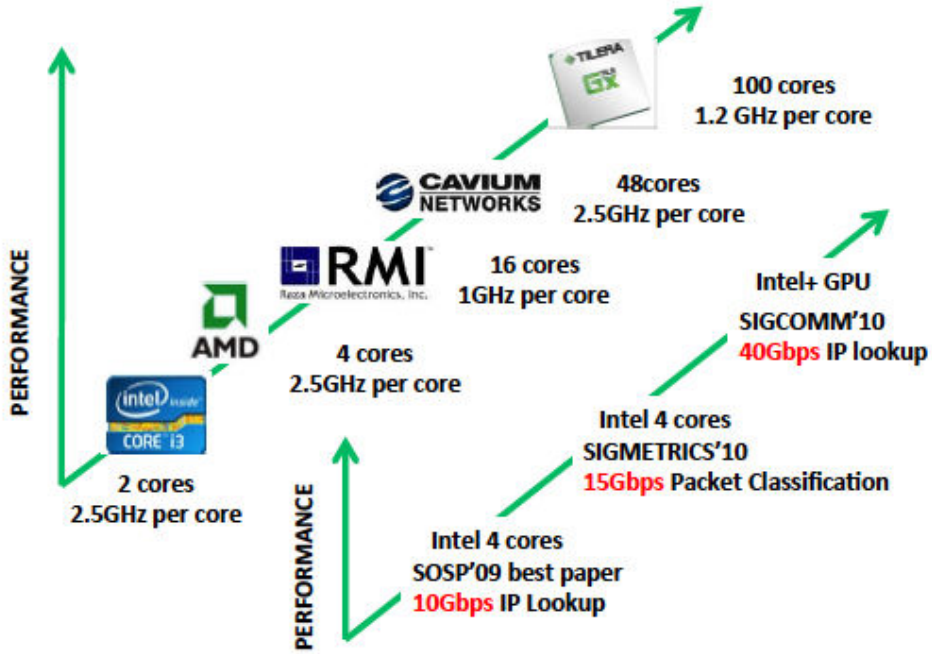


Figure 2 : Architecture Multi-cœurs et performances des routeurs logiciel

Les avancées dans les architectures multi-cœurs et dans le parallélisme à grain fin

Ainsi que je le montre dans la figure 2, le développement des processeurs multi-cœurs permet aujourd'hui d'atteindre des performances comparables aux routeurs commerciaux, mais avec un coût largement inférieur. Ceci s'est traduit depuis quelques années par plusieurs travaux de recherche visant à l'implantation de routeurs sur des architectures de serveurs communs [9,10,11]. Le system STORM de classification de paquets par logiciel [12] ainsi que le routeur logiciel accéléré à l'aide de GPU PacketShader [14] démontrent que la flexibilité et la performance élevée ne sont pas mutuellement exclusive sur les plateformes logicielles. Le projet PEARL décrit dans [13] est important puisque j'y ai contribué et développement l'architecture de ce système.

Alors que les librairies de capture et de traitement de paquets telle que Netmap [15], DPDK (Data Plane Development Kit) réduisent le goulot d'étranglement des entrées/sorties dans le système d'exploitation et réduisent la complexité du développement d'application haute performance. Ainsi le goulot d'étranglement se transfère maintenant sur la classification efficace de paquets.

2. Problématiques de recherche

Dans cette thèse, je discuterai principalement de deux problèmes : un problème de classification en une seule dimension, le routage IP, et un problème de classification multidimensionnel fondés sur des arbres de décision. Le problème du routage IP consiste en la recherche du plus long préfixe correspondant à une adresse IP donnée. Je présente ce problème dans la figure 3 ci-dessous.

Le routage IP est fondamentalement un problème de recherche plus long préfixe commun dans une table de routage contenant jusqu'à 500 K préfixes. Au vue de

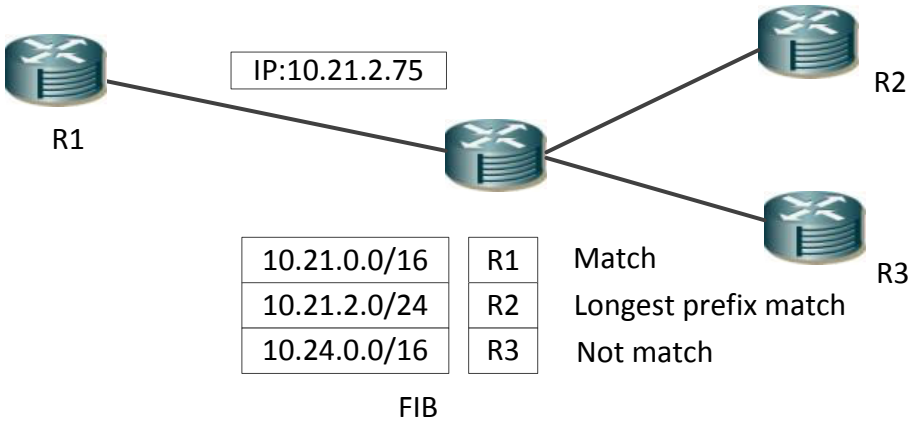


Figure 3- Routage IP par recherche de plus long préfixes communs

l'augmentation constante des débits sur les liens réseaux, le temps qui reste pour faire une recherche de plus long préfixe se réduit de façon régulière. Ainsi sur un lien à 100 Gbps le temps restant pour la recherche de préfixes, est de l'ordre que quelques dizaines de nanosecondes. En plus de la vitesse de recherche, il faut aussi intégrer les changements dans les tables de routage. Nous montrons dans la figure 4 une courbe de variation du nombre de mises à jours dans les tables de routages IP qui montre un débit de changement de l'ordre de 1000 mise à jour par seconde.

Le second thème de ma recherche a porté sur les ensembles de règles multidimensionnelles. Dans la Figure 5 nous montrons un système de classification typique et dans la Table 1 un ensemble de règles typique. Un ensemble de règles multidimensionnelles consiste en un ensemble d'intervalles définis sur plusieurs champs et une action à appliquer à un paquet qui vérifierait la règle.

	SIP	DIP	SP	DP	L4 protocol	Act.
R1	10.1.0.0/16	191.243.60.0/24	0 : 65535	1521 : 1521	TCP	DROP
R2	10.1.0.0/16	58.62.126.0/24	0 : 65535	1724 : 1724	TCP	DROP
R3	10.3.7.0/24	58.62.126.0/24	0 : 65535	1521 : 1521	TCP	PERMIT
R4	23.3.7.0/24	58.49.16.0/24	0 : 65535	14753 : 14753	TCP	PERMIT
R5	23.5.7.0/24	58.49.16.0/24	0 : 65535	5631 : 5631	UDP	RESET

Table 1- exemple de règles multi-dimensionnelles typique

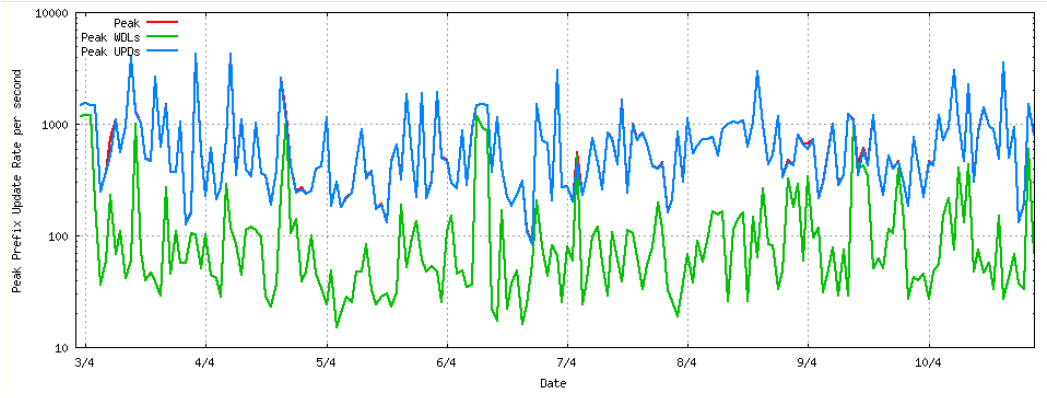


Figure 4: Fréquence de mises à jours des tables de routages.

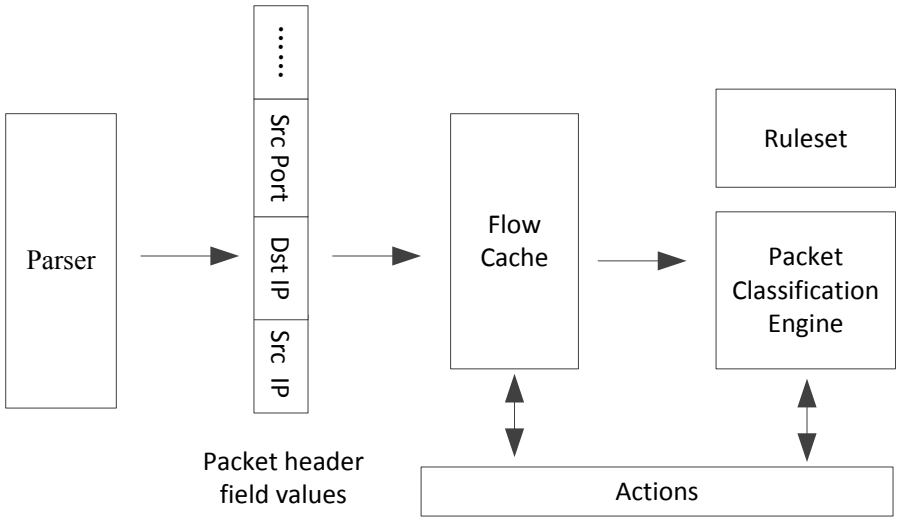


Figure 5- Architecture de classification de paquets

3. Contributions principales de la thèse

Conception d'algorithme

L'objectif principal de cette thèse, a été la conception d'algorithmes de classification de paquets rapide et efficace. A la différence des travaux précédents qui ont tenté de proposer un nouvel algorithme sans faire une analyse rétrospective afin de comprendre pourquoi les performances des algorithmes précédents étaient mauvaise, ma démarche dans cette thèse a été de commencer par une étude approfondie des algorithmes classiques de classification de paquets. Cette analyse m'a permis de lier les les propriétés intrinsèques des ensembles de règles à la performance des algorithmes de classification. Je me suis appuyé sur cette analyse pour proposer un cadre permettant aux ingénieurs de choisir, en fonction des propriétés des ensembles de règles, les briques algorithmes nécessaires à l'implantation d'algorithme de classification rapide et efficace.

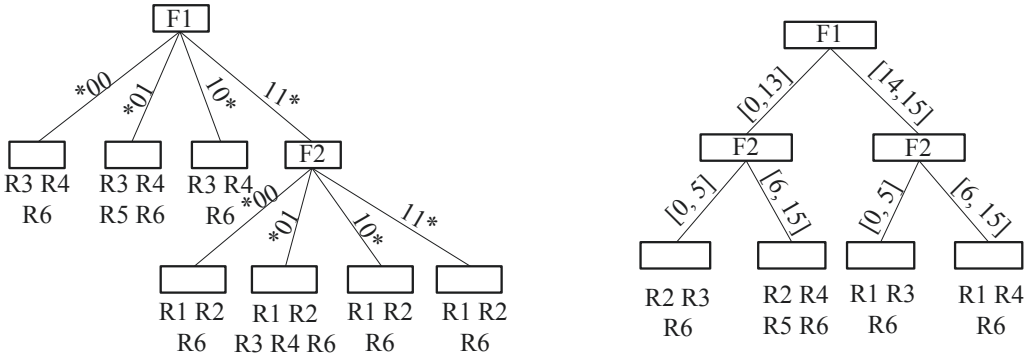
Dans la suite je commencerai pas décrire les observations sur les propriétés des ensembles de règles.

Propriétés des règles d'apprentissage

Un ensemble de règles de classification de paquets peut être considérée comme une collection d'intervalles définis sur plusieurs champs. Nous présentons dans la table 2 un ensemble de règles contenant 6 règles définies sur 2 champs de 4 bits, où « * » signifie n'importe quelle

Rule #	Field 1	Field 2	Action
R1	111*	*	DROP
R2	110*	*	PERMIT
R3	*	010*	DROP
R4	*	011*	PERMIT
R5	01**	10**	DROP
R6	*	*	PERMIT

Table 2- ensemble de règles de test



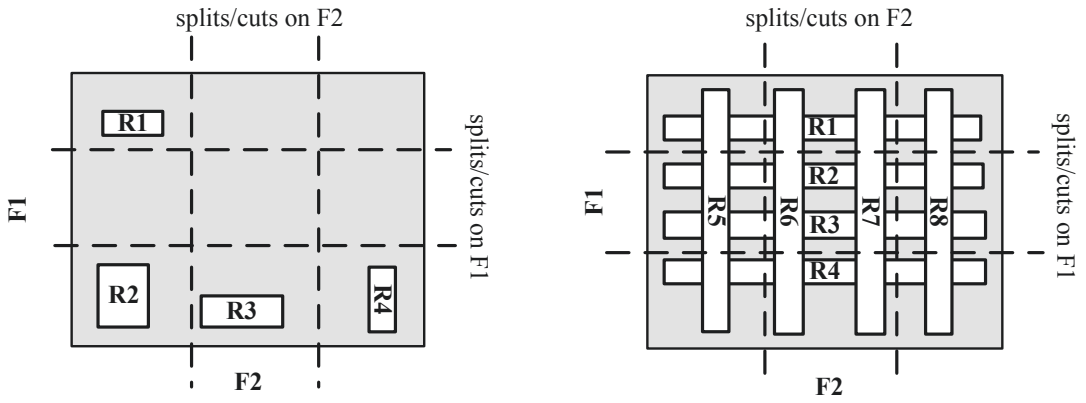
(a) HiCuts (b) HyperSplit

Figure 6- Classification par arbre de décision

valeur. Cet ensemble de règles peut être transformé en 4 intervalles distinct sur le champ 1 (R1 : [14,15], R2: [12,13] ; R5: [4,7] ; autre règles :[0,15]), et 4 sur le champ 2 (R3 :[4, 5] ; R4 : [6, 7] ; R5 : [8, 11] autres règles : [0, 15]). Une règle de classification de paquets peut être interprétée de façon géométrique : une règle définie sur k champs peut être considérée comme un k-orthotope, *i.e.*, un hyper-rectangle dans l'espace de dimension k. Par exemple, la règle R1 dans la table 2, définie dans l'espace des champs, une bande rectangulaire bi-dimensionnelle, où la largeur est 2 unités (de 14 à 15) dans le sens de l'axe du champ 1, et la longueur balaie la totalité de la portée du champ 2 où il y'a un symbole « * ». Similairement, la règle 3 définie une autre région rectangulaire, mais dont la largeur est dans le sens de l'axe du champ 2. Les différentes règles définissent chacune un k-orthotope qui peuvent se couper et définir des formes imbriqués et complexes. Un algorithme de classification est un algorithme auquel on donne les coordonnées d'un point dans l'espace et qui retourne l'identifiant du plus petit rectangle recouvrant ce point.

Maintenant étudions ce que fait un algorithme de classification utilisant un arbre de décision. Nous présentons dans la figure 6-b, un arbre de décision implantant la classification suivant les règles de la table 2. A la racine de l'arbre, une coupe est appliquée au champ 1 qui est divisé en deux parties [0,13] et [14,15]. Au second niveau, une coupe est appliquée au champ 2 qui est divisé en deux parties [0,5] et [6,15]. Ainsi chaque nœud dans l'arbre de décision peut être considéré comme une coupe appliquée dans une l'espace des champs. En appliquant ces coupes en suivant un chemin dans un arbre de décision, l'espace est partitionné en régions de plus plus en petites et ayant une intersection avec de moins en moins de k-orthotopes.

Finalement, la classification est efficace si à la fin il n'y a qu'une règle dans la région définie par la feuille du l'arbre de décision. La description précédente permet de comprendre les propriétés de l'ensemble de règle ayant un impact important sur la performance de la classification. Quand les règles sont clairsemées comme dans la figure 7-a, les coupes peuvent de façon efficaces séparer les différentes règles nécessitant que peu d'empreinte mémoire. Néanmoins, on observe fréquemment dans les ensembles de règles observées en pratique des structures orthogonales comme celles présentées dans la figure 7-b. Dans l'ensemble des règles définies dans la table 2, les règles R1, R2, R3 et R4 sont orthogonales. Quand de telles structures existent dans l'ensemble des règles, aucune coupe ne permet de séparer l'espace en régions ne contenant qu'une seule règle. Au mieux peut on avoir $O(N^K)$ régions contenant K



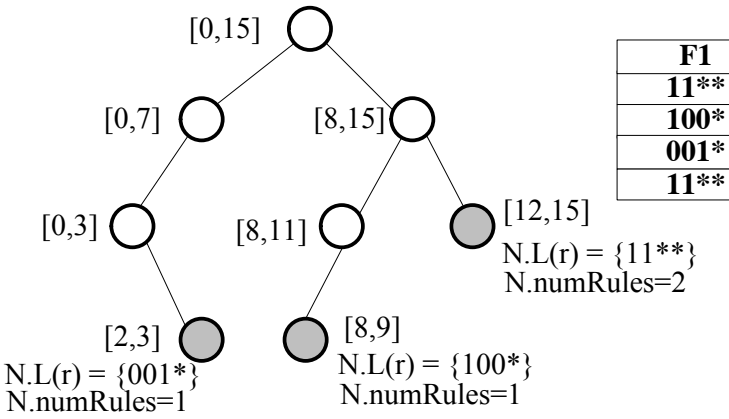
(a) sparse rules (b) orthogonal structure rules

Figure 7- Structures orthogonales et règles clairsemées

règles orthogonales, où *K* est la dimension de l'espace et *N* le nombre de règles orthogonales. Quand ceci arrive, chaque règle orthogonale doit être dupliquée dans toutes feuilles de l'arbre de décision, ce qui aboutit à un gaspillage de mémoire et une empreinte mémoire très importante.

Une seconde propriété importante de l'ensemble de règles résulte de la comparaison entre les deux arbres de décision dans la Figure 6. L'arbre dans la Figure 6-a est un arbre obtenu en appliquant des coupes de tailles égales à chacune des dimensions, alors que celui dans la figure 6-b utilise des coupes de tailles inégales. L'utilisation de coupes à tailles égales, quand cela est possible, réduit l'empreinte mémoire pour le stockage ainsi que le nombre d'opérations nécessaire pour le parcours de l'arbre. Par contre les coupes à tailles égales peuvent résulter en des duplications de règles quand une règle coupe plusieurs intervalles, ou du gaspillage quand aucune règle n'existe dans un intervalle. Plus la taille des intervalles est uniforme, plus l'utilisation de coupes à tailles égales est efficace. Par contre les coupes à tailles inégales sont plus efficaces, quand la taille des intervalles est non-uniforme. Il est ainsi nécessaire d'évaluer par le biais de métriques l'uniformité de la taille des intervalles afin de décider de la bonne stratégie de coupes à appliquer.

A cette fin j'ai développé durant ma thèse une méthodologie d'évaluation de l'uniformité des intervalles, par le biais d'arbres d'intervalles. L'arbre d'intervalles centrée est un arbre binaire permettant de représenter un ensemble d'intervalle. Chaque nœud de l'arbre d'intervalle contient des intervalles de règles et est représenté par un point *x,* le point médian de tous les intervalles contenus dans le nœud. La construction de l'arbre se fait en partitionnant l'ensemble des intervalles de règles contenus dans le nœud en trois sous-ensembles : le premier sous-ensemble contient toutes les règles qui contiennent le point *x*, le second sous-ensemble, nommé le sous-ensemble de droite, contient toutes les règles qui sont complètement à droite du point *x*, et le sous-ensemble de gauche, contient toutes les règles complètement à gauche du point *x*. On construit le fils de droite du nœud en utilisant les intervalles dans le sous-ensemble de droite et le fils de gauche avec le sous-ensemble de gauche. Je présente dans la figure 8 un exemple d'arbre d'intervalle centré. La structure de l'arbre d'intervalles centrés construit sur chaque dimension de l'ensemble de règle donne une indication importante sur l'uniformité des règles. Les règles avec des intervalles larges sont généralement absorbées par les nœuds proches de la racine de l'arbre, alors que les intervalles



F1	F2
11**	111*
100*	11*
001*	001*
11**	0*

Figure 8- Arbre d'intervalle centrés

courts sont généralement associés aux feuilles de l'arbre. Un arbre totalement équilibré signifie que tout l'intervalle rattaché à toutes règles est de même taille, et ainsi qu'une coupe en intervalles de tailles égale sera parfaite. Par contre un arbre non-équilibré privilégie une coupe en intervalles de tailles inégales. En pratique, les arbres d'intervalles construits sur des ensembles de règles réalistes sont non équilibrés. Ils contiennent des nœuds ayant un seul fils, et des feuilles à divers niveaux de l'arbre. En fait ont peut analyser un arbre comme plusieurs sous-arbres quasi-équilibrés de tailles différents (voir figure 9). Afin de caractériser ces arbres, j'ai défini deux métriques pour les arbres d'intervalles. Je défini pour chaque nœud, sa profondeur d'équilibre *BD* comme la hauteur de l'arbre quasi-équilibré auquel le nœud appartient, et la distance d'équilibre *D* comme le nombre d'arbre quasi-équilibrés entre la racine de l'arbre et le nœud. La totalité de l'arbre est caractérisée par D_{max} et BD_{max}, respectivement la valeur maximale de *D* et de *BD* pour tous les nœuds de l'arbre. Une grande valeur de D_{max} signifie que l'arbre contient beaucoup de sous-arbres de petites tailles et que les intervalles dans l'ensemble de règles ne sont pas uniformes. Par contre, une petite valeur de D_{max} signifie une couverture plus uniforme.

Les observations précédentes sur les propriétés des ensembles de règles m'ont permis de développer un estimateur de la taille de l'empreinte mémoire nécessaire pour implanter un arbre de décision en utilisant les principaux algorithmes existants dans la littérature : *hypercut* [16, 17] et *hypersplit* [18].

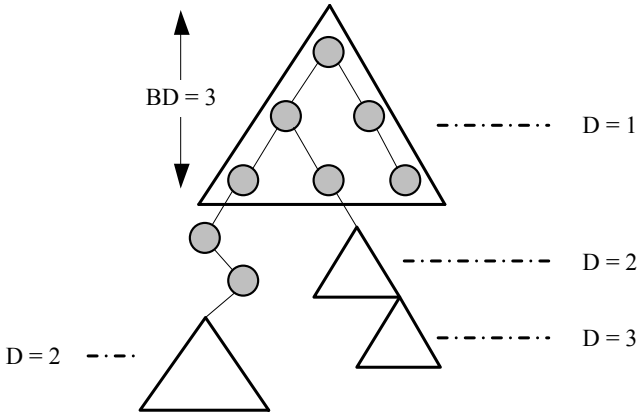


Figure 9- structure de l'arbre d'intervalles centrés

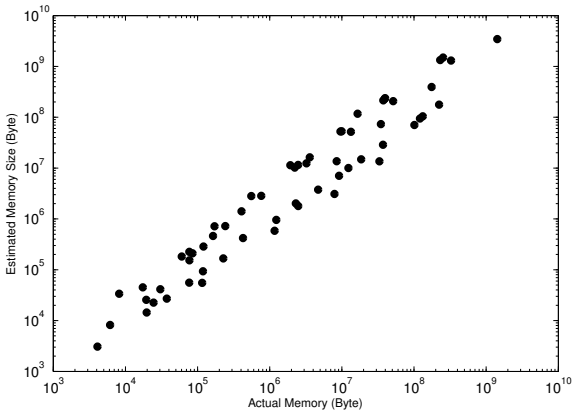


Figure 10- Estimation de l'empreinte mémoire

Cet estimateur qui n'utilise pas les détails d'implantation de chaque algorithme permet de prédire avec une bonne précision l'empreinte mémoire. Les résultats de cette estimation sont présentés dans la figure 10.

L'estimateur d'empreinte mémoire ainsi que la connaissance obtenue sur la structure des ensembles de règles m'a permet d'obtenir une compréhension fine des mécanismes contrôlant la complexité des algorithmes de classification de paquets. Dans la suite, je m'appuie sur cette compréhension afin de concevoir de nouveaux algorithmes ou d'améliorer ceux existant.

Cadre décisionnel pour concevoir des algorithmes de classification

Afin de comprendre quels sont les paramètres qui jouent réellement sur la performance réelle des classificateurs de paquets logiciels, nous mesuré la performance de l'algorithme de classification HyperSplit sur 25 ensembles de règles. Les paramètres mesurés sont les taux de défaut de mémoire cache, la latence moyenne d'accès à la mémoire et la taille de l'empreinte mémoire. Je présente dans la figure 11 la relation de la taille de l'empreinte mémoire avec la latence moyenne d'accès à la mémoire et au taux de défaut de mémoire cache.

La figure 11 montre que la latence moyenne d'accès à ma mémoire augmente lentement quand la taille de la mémoire croît de 10 Ko à 10 Mo. A partir de 10 Mo la latence explose. Cette augmentation notable peut être expliqué en regardant la courbe du taux de défaut de mémoire cache et en prenant en compte la taille de la mémoire cache des processeurs de notre plateforme d'évaluation (4Mo de cache L3). La figure 11 montre que le taux de défaut de mémoire cache reste inférieur à 10% quand l'empreinte mémoire est inférieure à 10 Mo, mais elle augmente brutalement à plus de 50% pour des tailles de mémoires plus grandes. Il apparaît ainsi que dès que la taille de l'empreinte mémoire dépasse la taille de la mémoire cache des processeurs le taux de défaut de mémoire augmente. Il est ainsi important de s'assurer que l'empreinte mémoire ne dépasse pas les 10 Mo. J'ai utilisé ce critère en combinaison avec l'estimateur de taille d'empreinte mémoire afin de construire un outil d'aide à la décision permettant de choisir l'algorithme le plus efficace pour un ensemble de règles données. Cet outil d'aide à la décision est appelé « *smartsplit* ».

SmartSplit traite de deux décisions. Une première décision est liée à l'uniformité des intervalles de règles et est relative à la décision du type de la coupe a appliquée, et au choix de

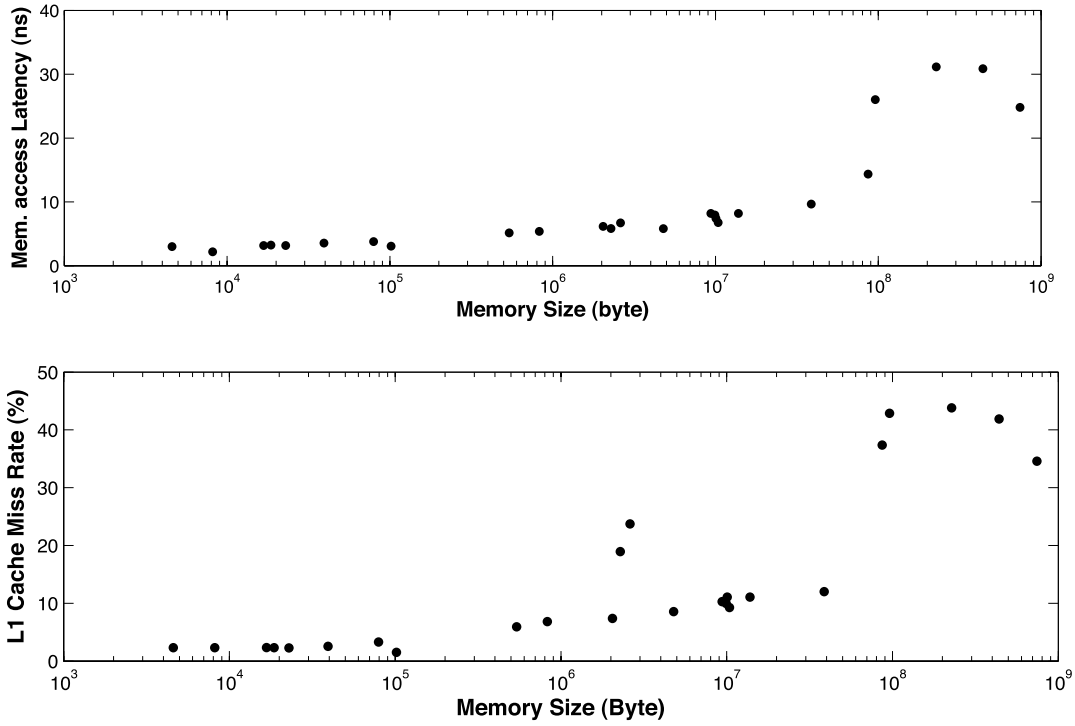


Figure 11- Latence moyenne d'accès à la mémoire et taux de défaut de mémoire cache en fonction de l'empreinte mémoire des algorithmes de classifications de paquets

la dimension à laquelle il faut l'appliquer. La seconde décision est liée à l'existence des structures orthogonales dans l'ensemble des règles et à la pertinence de découper l'ensemble en sous-ensembles sans règles orthogonales.

La première décision est relative aux deux problèmes suivants : 1) comment choisir la dimension sur laquelle il convient de faire une coupe, et 2) comment choisir le type de coupe à appliquer : coupe à taille égale, ou coupe inégales. Je décris dans la figure 12 les éléments qui caractérisent cette première décision. En combinant les différents éléments et en particulier les différentes astuces d'implémentation, j'ai développé plusieurs variantes des algorithmes existants que j'appelle méta-méthodes. Celles-ci sont listés dans la table 3 ci-dessous. L'étape suivante a été de comparer la performance atteinte par ces différentes méta-méthodes.

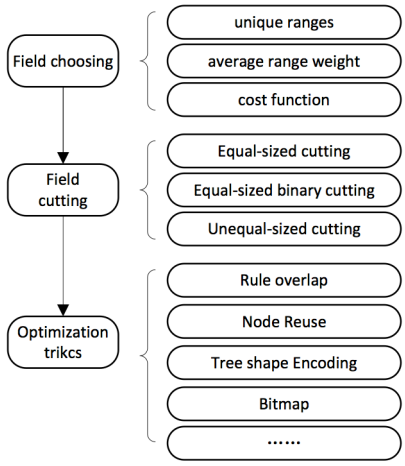


Figure 12- Décision sur le type de coupe

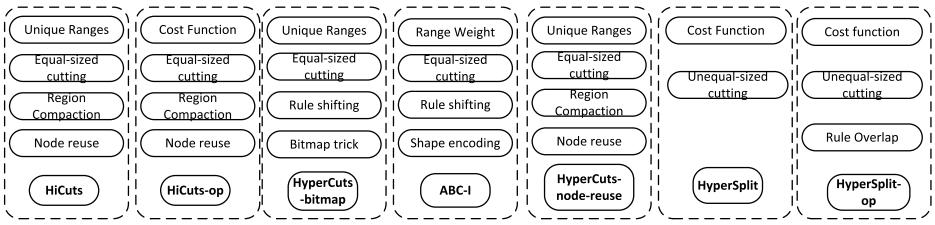
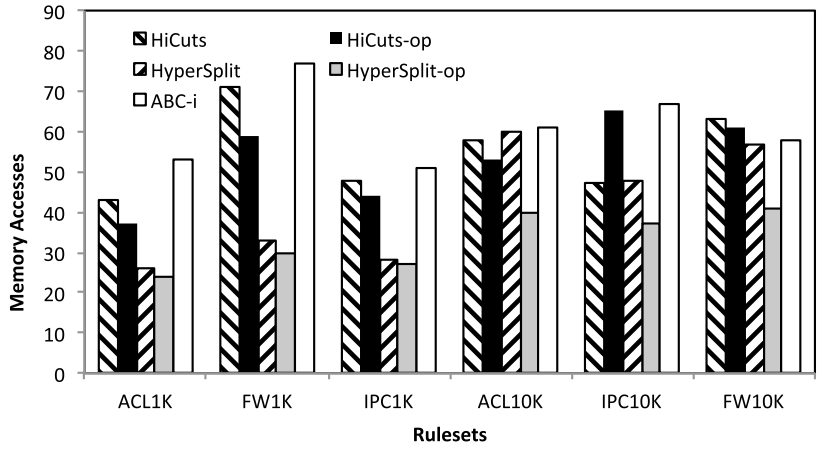
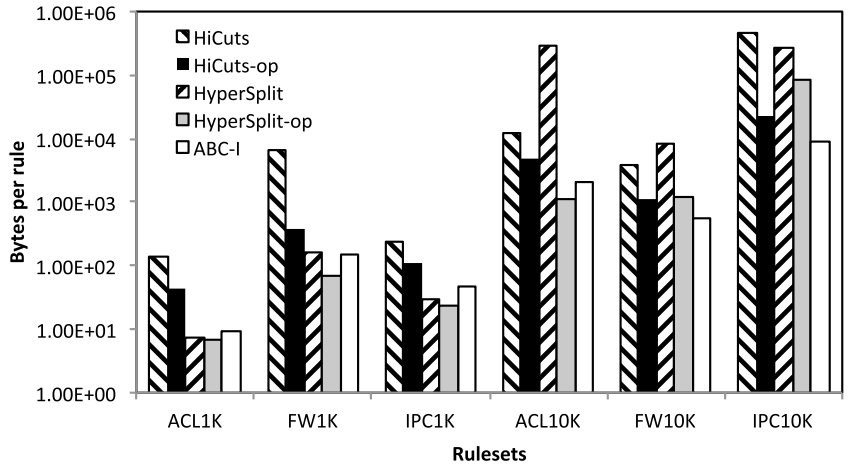


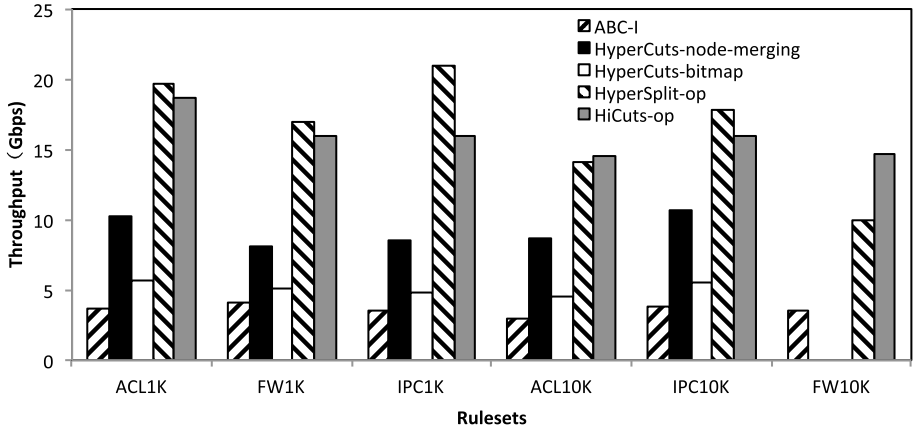
Table 3-meta-méthodes de classification



(a) nombre d'accès mémoire



(b) empreinte mémoire (octets/règles)



(c) débit mesuré avec des paquets de 64 o en utilisant un seul cœur de calcul

Figure 13- performance des méta-algorithmes

Je présente dans la figure 13 les performances obtenues par les différentes méta-méthodes comparées à celle obtenue par l'état de l'art des algorithmes de classifications. La plateforme d'évaluation était un serveur Intel avec un processeur 3.3GHz, avec une cache L1 de 256 Ko, une cache L2 de 1 Mo, une cache L3 de 4 Mo. Toutes les expérimentations ont été effectuées en utilisant un seul cœur de calcul. Nous avons utilisé ClassBench pour générer 24 ensembles de règles de différents types. La figure 13 montre que les optimisations HiCuts-op que j'ai

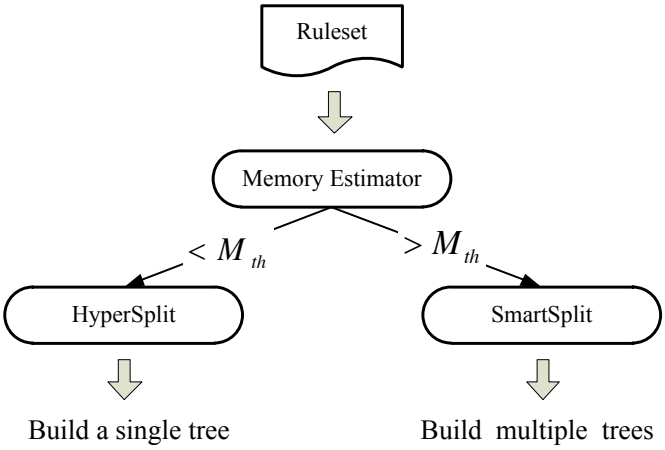
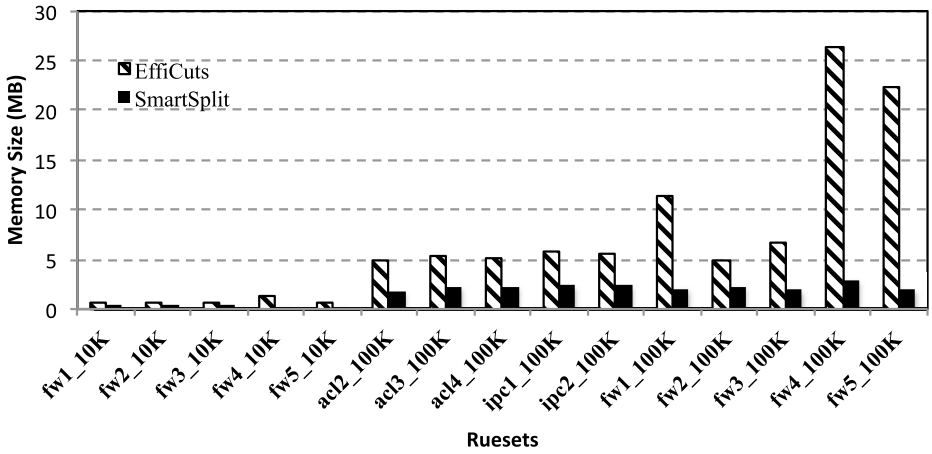


Figure 14- Cadre de décision liée à décision de découper l'ensemble de règles en sous-règles

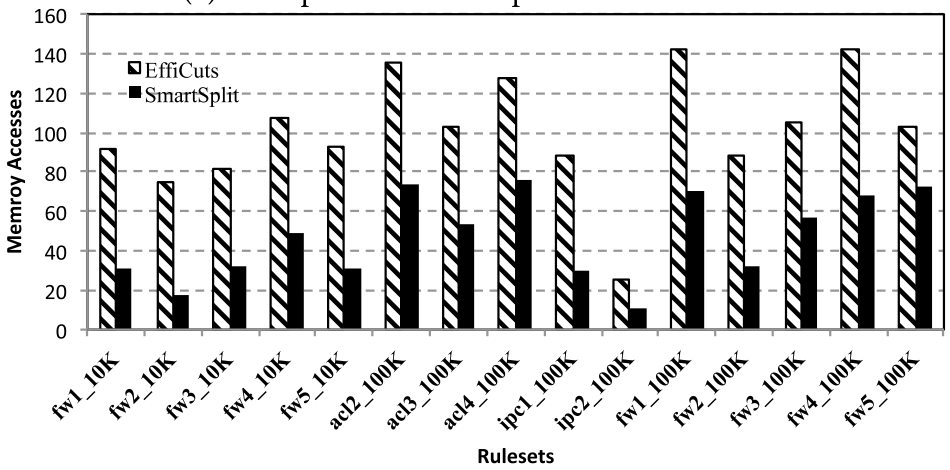
proposées permettent de réduite l'empreinte mémoire de l'algorithme HiCuts de 2 ~20 avec 10% d'accès mémoire en moins. Similairement HyperSplit-op atteint une réduction d'empreinte mémoire 2~200 et 10 à 30% d'accès mémoire en moins. La dernière figure présente le débit mesuré en terme de volume de trafic traité par seconde. Nous observons que sur un lien saturé avec des paquets de 64 octets, nous obtenons un débit supérieur à 20 Go par secondes sur la plupart des configuration en utilisant un seul cœur de calcul. En comparaison, les résultats présentés dans [12] qui définissent l'état de l'art ne dépassent pas 16 Go par seconde en utilisant 8 cœurs de calculs avec des paquets de 128 o. Ceci montre le gain important de performance que j'ai obtenu par rapport à l'état de l'art.

Un second cadre de décision que j'ai développé durant cette thèse, a été la décision liée à l'existence des structures orthogonales dans l'ensemble des règles et à la pertinence de découper l'ensemble en sous-ensembles sans règles orthogonales. Je commencerais par décrire la méthodologie de découpage de l'ensemble de règles. L'objectif de ce découpage est de se débarrasser des structures orthogonales qui aboutissent à une duplication importante des règles et une large empreinte mémoire. La méthodologie de découpage consiste initialement à construire pour chacun des champs IP source et destination, l'arbre des intervalles centrés et à calculer leur valeur du paramètre $D_{\max}$. Ensuite, nous trouvons les règles qui sont définies par un intervalle large sur chacune des adresses IP source et destination. Une règle est définie comme large sur une dimension si l'intervalle qui défini cette règle recouvre plus de la moitié de l'intervalle possible. Ainsi on peut définir sur l'adresse IP source et destination 4 catégories : (*large*, *large*), (*large*, *small*), (*small*, *large*), (*small*, *small*). Les ensembles (*large*, *large*), (*large*, *small*), (*small*, *large*) définissent chacun un arbre de décision dans lequel il n'y a plus de règles orthogonales. L'ensemble des règles (*small*, *small*) est ajoutée à l'ensemble des règles (*large*, *small*) si $D_{\max}$ (srcIP)< $D_{\max}$ (dstIP), et aux règles (*small*, *large*) sinon. Ensuite sur chacun de ces arbres nous appliquons le cadre défini précédemment pour le choix de la méthode de coupe à appliquer : de taille égale ou de taille inégale.

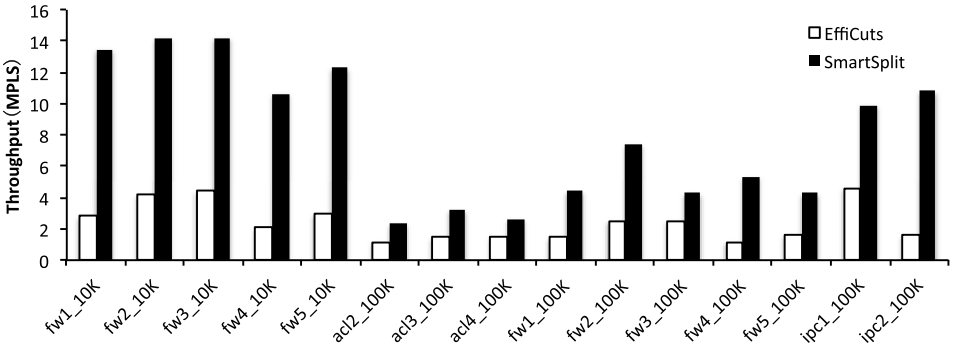
La décision sur la pertinence d'appliquer un découpage est décrite dans la figure 14. J'ai utilisé l'estimateur de taille de mémoire décrit précédemment pour calculer pour un ensemble de règles donnée la taille de l'empreinte mémoire sans découper en sous-ensembles



(a)- Comparaison de l'empreinte mémoire



(b)-Comparaison du nombre d'accès mémoire



(c)- Comparaison des débits atteints en Million de recherche de paquet par seconde

Figure 15- Comparaison des performances atteintes par EffiCuts et SmartSplit

et en découpant en sous-ensemble de la façon décrite précédemment. Si la taille de l'empreinte mémoire sans découpage dépasse le seuil de 10 Mo, je découpe l'ensemble des règles. Ceci aboutit à un cadre décisionnel que j'appelle SmartSplit. Une autre approche de découpage d'ensemble de règles en sous règles est aussi proposée dans la littérature [20]. Cette approche qui est appelé Efficuts, défini un état de l'art dans les méthodes découpant l'ensemble de règles.

Type	Size	AutoPC(MLPS)	EffiCuts(MLPS)	speedup
ACL	1K	11.3*	4.5	2.4
	10K	6.9*	3.1	2.2
	100K	8.6*	2.2	3.9
FW	1K	9.8*	2.4	4.1
	10K	10.7	2.1	5.1
	100K	7.4	2.5	3.0
IPC	1K	12.6*	3.0	4.25
	10K	5.3*	1.48	3.6
	100K	9.91	1.63	6.1
Average Speedup: 3.8				

Table 4- Comparaison des débits atteint par Efficut et SmartSplit en millions de recherche par seconde

Je présente dans la figure 15 la comparaison des performances atteintes par EffiCuts et SmartSplit. On peut observer que SmartSplit a largement de meilleures performances qu'EffiCuts aussi bien en terme d'empreingte mémoire qu'en terme d'accès mémoire, *e.g.*, pour l'ensemble de règle fw5 contenant 100K règles, EffiCuts consomme 22.46 Mo de mémoire, alors que SmartSplit a seulement besoin de 1.98 Mo, un réduction de 11.3. De plus en comparant l'empreinte mémoire après découpage de l'ensemble de règles avec celle-ci avant on observe que pour un même ensemble de règle l'empreinte mémoire passe de plusieurs Go à moins de 2 Mo. Autre point notable est que SmartSplit ne découpe l'ensemble de règles qu'en trois sous-ensembles alors qu'Efficuts peut découper en 9 sous-ensembles et avoir a gérer ainsi 9 sous arbres. Je présente dans la table 4 la comparaison des débits atteints par Efficuts et SmartSplit montre que SmartSplit atteint en moyenne un débit 3.8 supérieur.

Ceci termine la description résumée d'une partie des contributions de ma thèse. C'est travaux ont aboutit a deux publications [20, 21].

PEARL : un prototype pour le SDN/NFV

Déployer, expérimenter et tester de nouveaux protocoles et systèmes sur l'Internet a toujours été un défi important. La simulation n'a jamais remplacé le déploiement en réel de système et l'expérimentation *in vivo*. Ainsi l'accès à une plateforme programmable et flexible pouvant implanter du traitement de paquet a haut débit, et quo permettrait ainsi le déploiement et l'expérimentation des concepts développés en recherche en très importante. De plus, avec l'avancée graduelle vers les architecture de l'Internet du futur qui sera plus polymorphique que l'Internet monolithique actuel, il faudrait pouvoir permettre à plusieurs paradigme architecturaux de coexister, par exemple un routeur pourra avoir a gérer en même temps un réseau NDN [22] et un réseau IP classique. Ainsi, les plateformes pour l'Internet du futur devront permettre l'exécution en parallèle, de plusieurs routeurs virtuels tout en assurant une indépendance entre ces instances virtuelles.

Malheureusement, les architectures classiques de traitement de paquet ne sont pas adaptées à la flexibilité que j'ai décrite plus haut. Ceci m'a motivé pour concevoir et construire une plateforme de routeur virtuel programmable, PEARL (ProgrammablE virtuAl Router pLatform), qui permet de garantir une très haute performance tout en validant des contraintes d'indépendance. Les défis de la conception de PEARL étaient multiples. Il fallait tout d'abord gérer la balance flexibilité *vs.* performance qui se traduit généralement par la volonté de

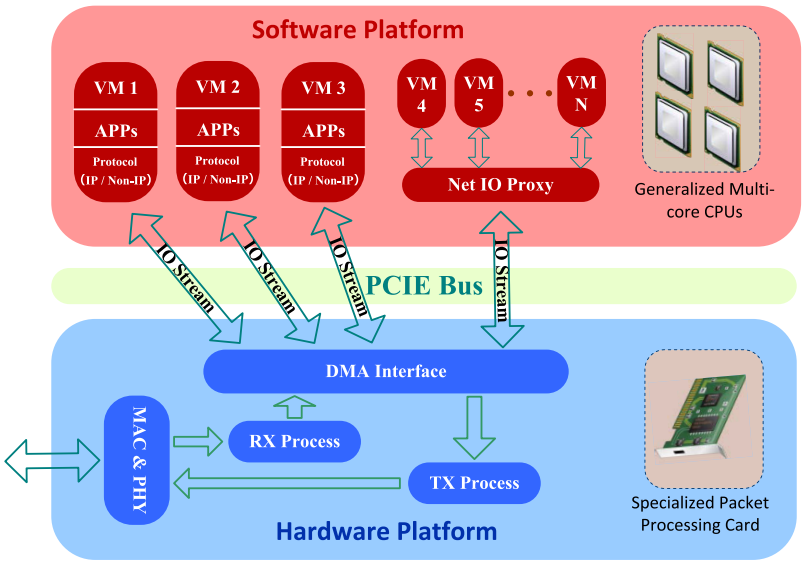


Figure 16- Architecture PEARL

pousser des fonctionnalités dans le matériel, plutôt que de les implanter en logiciel ce qui garantirait la flexibilité. Le second défi était d'assurer l'isolation entre les instances de routeurs virtuels s'exécutant sur la plateforme en utilisant le moins de ressources possibles. Je décris dans cette partie l'architecture de PEARL, ces composants principaux et la performance atteinte par cette plateforme innovante.

Description de la plateforme PEARL

La Plateforme PEARL utilise une plateforme hybride combinant des processeurs multi-cœurs Intel qui exécute du code classique, et des cartes de traitements de paquets spécialisé qui implantent le traitement haut-débit de paquets. La plateforme est présenté dans la Figure 16. L'environnement de virtualisation utilisé est fondée sur LXC, une environnement à base de Linux [22]. Cet environnement autorise l'exécution en parallèle de plusieurs instances de routeurs virtuels, et chaque instance peut être considéré comme une machine totalement séparée.

Les cartes spécialisées dédiées contiennent des éléments de traitement de paquets FPGA ainsi que des mémoires TCAM et SRAM. Cette carte permet le traitement rapide des paquets tout en garantissant une isolation forte entre instances s'exécutant. Ainsi PEARL peut implanter plusieurs plans de données virtuels en allouant à chaque plan des ressources matérielles séparées. Ceci facilite l'isolation au niveau matériel.

L'isolation au niveau logiciel est gérée par LXC. La liaison entre le matériel et le logiciel est faite par le biais d'un bus PCI Express à haut débit et une DMA multi-flot. Chaque flot d'entrée sortie peut être ou dédiés à un routeur virtuel unique ou partagé entre plusieurs routeurs virtuels grâce à un proxy d'entrées/sorties. La flexibilité est atteinte par l'utilisation de ressources TAP/TUN comme interface réseau pour chaque machine virtuelle. Ainsi chaque machine virtuelle peut être considérée comme une machine Linux standard contenant

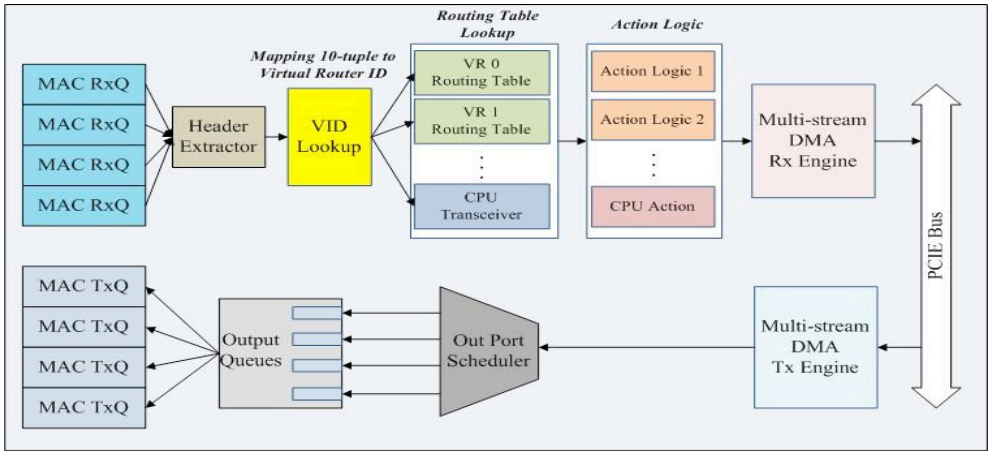


Figure 17-Architecture du plan de données de PEARL

plusieurs ports réseaux. Ainsi une pile protocolaire IPv4, IPv6, OpenFlow peuvent être facilement déployé sur ces instances virtuelles.

L'architecture du plan de données de PEARL dans la carte de traitement matérielle, décrite dans la figure 17, permet de transférer l'étape de distribution de paquets aux différentes machines virtuelles qui est généralement le goulot d'étranglement de performance au niveau matériel ce qui permet une vitesse de transfert de paquet importante. Le plan de données est fondés sur un pipeline contenant deux chemins séparés : un chemin d'envoi et un chemin de réception. De plus, la virtualisation LXC est suffisamment légère pour ne pas avoir d'impact important sur la performance. Ainsi la plateforme atteint un niveau de performance élevé.

La plateforme logicielle de PEARL consiste en plusieurs composants présentés dans la figure 18. Le composant *vmmd* fournit les fonctions de gestion de base nécessaire au routeur virtuel, et aux cartes de traitement de paquets. Le composant *nacd* fournit à toutes les cartes de traitement une interface uniforme vers l'extérieur de l'environnement virtualisé; Le composant *routed* s'occupe du routage et transforme les règles de transfert définies par le noyau ou l'application utilisateur en format uniforme pour chaque routeur virtuel afin de les implanter dans la TCAM. Le composant *netio* permet le transfert de paquets entre les interfaces physique et les interfaces virtuelles.

J'ai défini deux types de routeur virtuel dans PEARL : routeur virtuel de haute et basse priorités. Chaque instance de routeur virtuel haute priorité est lié à une paire de mémoire tampon Rx/Tx de la DMA et à un espace indépendant dans la mémoire TCAM. Grace à la capacité de recherche rapide de la TCAM, et des entrées/sorties rapides fournies par la carte matérielle, le routeur virtuel haute priorité peut atteindre un débit de transfert très important. Les routeurs virtuels basse priorité partagent tous ensembles une paire de mémoire tampons Rx/Tx, et n'utilisent pas la TCAM pour la recherche dans la table de routage.

Dans la suite, je présente les performances obtenues par la plateforme PEARL. La plateforme PEARL a été implanté sur un serveur équipé d'un processeur Xeon 2.5 GHz avec 16 Go de RAM DDR2. Une carte matérielle spécialisée contenant des FPGAs ainsi qu'une TCAM a été ajouté à cette plateforme. Afin de montrer la flexibilité de la plateforme PEARL, j'ai évalué trois configuration : routeur IPv4 virtuel de haute performance, un routeur virtuel

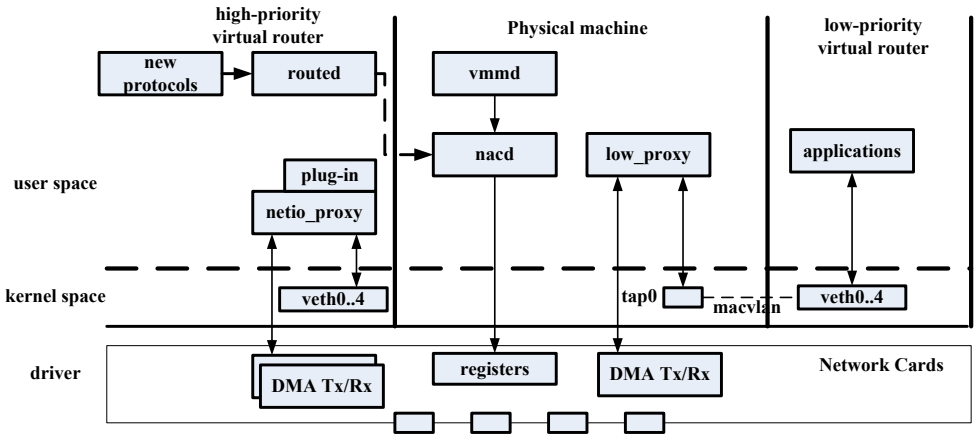


Figure 18- Architecture logicielle de PEARL

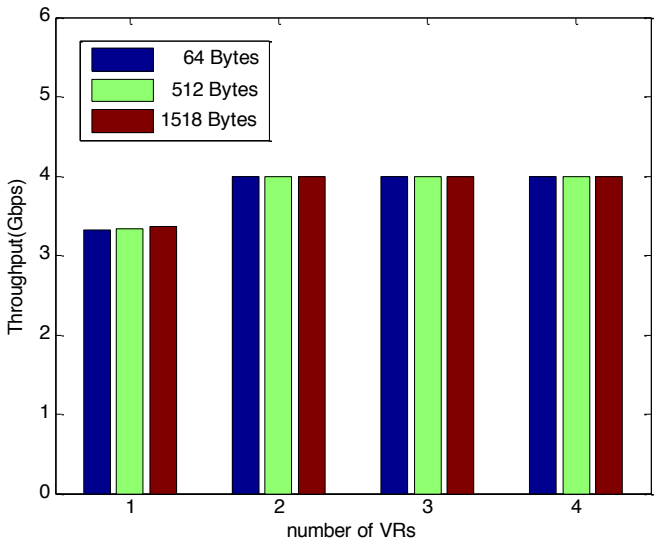
virtuel IPv4/IPv6 au niveau noyau et un routeur IPv4 implanté en tant que routeur virtuel basse priorité. J'ai mesuré pour chaque scenario avec des tailles de paquets (64, 512, 1518 octets pour les paquets IPv4 et 78, 512 et 1518 pour les paquets IPv6).

La figure 19-a montre le débit atteint dans le premier scenarios en fonction du nombre du routeurs virtuels. Chacun des routeurs virtuels est implanté comme un routeur haute priorité. Nous observons que dès que le nombre d'instance de routeur virtuel atteint 2, nous atteignons la limite théorique de notre conception qui est de 4 Gbps.

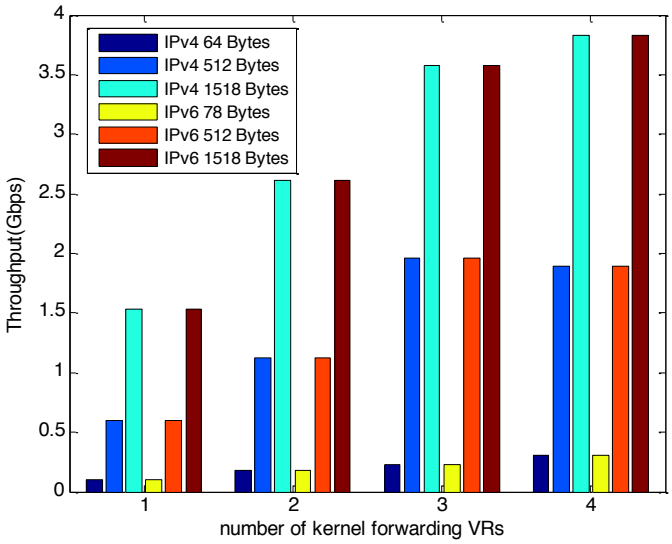
La figure 19-b illustre le débit en fonction du nombre d'instance de routeur virtuel IPv4/IPv6. Ici aussi les routeurs sont à haute priorité sauf qu'aucun routeur n'a d'espace dédiés dans TCAM, et d'autre terme l'espace des tables de routage est partagé dans la TCAM. On observe ici que le débit s'approche des 4 Go avec 4 instances de routeurs virtuels.

La figure 19-c montre le débit atteint dans le troisième scénario, ou on ajoute des routeurs virtuels de basses priorités. Tous ces routeurs virtuels se partagent une paire de canal entrées/sorties Tx/Rx de la DMA. On peut observer que le débit global dans ce scénario ne dépasse pas les 2 Gbps et que le goulot d'étranglement reste principalement le partage des paquets entre les différents routeurs virtuels.

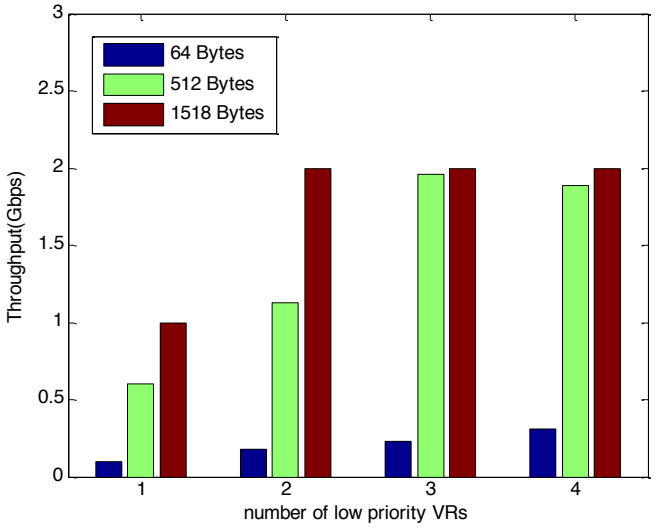
La plateforme PEARL a été le sujet d'une publication [23] qui a obtenu une grande visibilité car elle a été publiée dans un numéro spécial visant l'architecture de l'Internet du futur.



(a)- performance du scenario 1 de PEARL, routeur virtuel implanté en haute priorité en function du nombre d'instance de routeurs virtuels



(b)-Performance de PEARL dans le scénario 2



(c)- Performance de PEARL dans le scénario 3.

Figure 19- Performance de la plateforme PEARL pour différents scénarios d'utilisation

Bibliographie

[1]-Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review, 37(4):1-12, 2007.

[2]- Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In ACM SIGCOMM Computer Communication Review, volume 38, pages 51-62. ACM, 2008.

[3]-Lucian Popa, Norbert Egi, Sylvia Ratnasamy, and Ion Stoica. Building extensible networks with rule-based forwarding. In OSDI, pages 379-392, 2010.

[4]- N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69-74, 2008.

[5]-Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: a retrospective on evolving sdn. In Proceedings of the first workshop on Hot topics in software de_ned networks, pages 85-90. ACM, 2012.

[6]-Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In Hotnets, 2009

[7]- Stefan Nilsson and Gunnar Karlsson. Ip-address lookup using lc-tries. Selected Areas in Communications, IEEE Journal on, 17(6):1083{1092, 1999.

[8]-The amazon ec2 outage no one noticed. http://www.praxicom.com/2008/04/the-amazon-ec2.html.

[9]-B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor pc router. In Proceedings of the 2001 USENIX Annual Technical Conference, pages 333-346, 2001.

[10]-M. Dobrescu, N. Egi, K. Argyraki, B.G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In ACM SOSP, volume 9. Citeseer, 2009.

[11]-Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In Proceedings of the ninth ACM conference on Emerging networking experiments and technologies, pages 97-108. ACM, 2013.

[12]-Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging parallelism for multi-dimensional packetclassi_cation on software routers. In ACM SIGMETRICS Performance Evaluation Review, volume 38, pages 227{238. ACM, 2010.

[13]-Gaogang Xie, Peng He, Hongtao Guan, Zhenyu Li, Layong Luo, Jianhua Zhang, Yonggong Wang, and K Salamatian. Pearl: a programmable virtual router platform. Communications

Magazine, IEEE, 49(7):71-77, 2011.

[14]-S. Han, K. Jang, K.S. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In ACM SIGCOMM Computer Communication Review, volume 40 pages 195-206. ACM, 2010.

[15]-Luigi Rizzo. netmap: a novel framework for fast packet i/o. In Proceedings of the 2012 USENIX conference on Annual Technical Conference, pages 9–9. USENIX Association, 2012.

[16]- P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In Hot Interconnects VII, pages 34–41, 1999.

[17]-S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pages 213–224. ACM, 2003.

[18]- Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In INFOCOM 2009, IEEE, pages 648–656. IEEE, 2009.

[19]–B. Vamanan, G. Voskuilen, and TN Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In ACM SIGCOMM Computer Communication Review, volume 40, pages 207–218. ACM, 2010.

[20]- Peng He, Gaogang Xie, Hongtao Guan, Laurent Mathy, Salamatian Kavé. Toward predictable performance in decision tree based packet classification algorithms. *19th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN), 2013*, Apr 2013, Belgium. pp.1-6, 2013

[21]- Peng He; Gaogang Xie; Salamatian, K.; Mathy, L., "Meta-algorithms for Software-Based Packet Classification," *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on* , vol., no., pp.308,319, 21-24 Oct. 2014

[22]-Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In Proceedings of the 5th international conference on Emerging networking experi- ments and technologies, pages 1–12. ACM, 2009.

[23]-Linux container. https://linuxcontainers.org.

[24]-Gaogang Xie, Peng He, Hongtao Guan, Zhenyu Li, Layong Luo, Jianhua Zhang, Yonggong Wang, and K Salamatian. Pearl: a programmable virtual router plat- form. Communications Magazine, IEEE, 49(7):71–77, 2011.

Design and Evaluation of High Performance Software based Packet Classification Systems

Peng He

A Dissertation
Presented to the Faculty
of Grenoble University
in Candidacy for the Degree
of Doctor of Philosophy

by the Department of
Electrical Engineering
Adviser: Kavé Salamatian, Gaogang Xie

September 2014

Abstract

Packet classification consists of matching packet headers against a set of pre-defined rules, and performing the action(s) associated with the matched rule(s). As a key technology in the data-plane of network devices, packet classification has been widely deployed in many network applications and services, such as firewalling, load balancing, VPNs *etc.* Packet classification has been extensively studied in the past two decades. Traditional packet classification methods are usually based on specific hardware. With the development of data center networking, software-defined networking, and application-aware networking technology, packet classification methods based on multi/many processor platform are becoming a new research interest. In this dissertation, packet classification has been studied mainly in three aspects: algorithm design framework, rule-set features analysis and algorithm implementation and optimization.

In the dissertation, we review multiple proposed algorithms and present a decision tree based algorithm design framework. The framework decomposes various existing packet classification algorithms into a combination of different types of "meta-methods", revealing the connection between different algorithms. Based on this framework, we combine different "meta-methods" from different algorithms, and propose two new algorithms, HyperSplit-op and HiCuts-op. The experiment results show that HiCuts-op achieves $2 \sim 20\times$ less memory size, and 10% less memory accesses than HiCuts, while HyperSplit-op achieves $2 \sim 200\times$ less memory size, and $10\% \sim 30\%$ less memory accesses than HyperSplit.

In the dissertation, we also explore the connections between the rule-set features and the performance of various algorithms. We find that the "coverage uniformity" of the rule-set has a significant impact on the classification speed, and the size of "orthogonal structure" rules usually determines the memory size of algorithms. Based on these two observations, we propose a memory consumption model and a quantified method for coverage uniformity. Using the two tools, we propose a new multi-decision

tree algorithm — SmartSplit and a algorithm policy framework — AutoPC. Compared to EffiCuts algorithm, SmartSplit achieves around 2.9× speedup and up to 10× memory size reduction. For a given rule-set, AutoPC can automatically recommend a "right" algorithm for the rule-set. Compared to using a single algorithm on all the rulesets, AutoPC achieves in average 3.8 times faster.

We also analyze the connection between prefix length and the update overhead for IP lookup algorithms. We observe that long prefixes will always result in more memory accesses using Tree Bitmap algorithm while short prefixes will always result in large update overhead in DIR-24-8. Through combining two algorithms, a hybrid algorithm SplitLookup is proposed to reduce the update overhead. Experimental results show that, the hybrid algorithm achieves 2 orders of magnitudes less in memory accesses when performing short prefixes updating, but its lookup speed with DIR-24-8 is close.

In the dissertation, we implement and optimize multiple algorithms on the multi/many core platform. For IP lookup, we implement two typical algorithms — DIR-24-8 and Tree Bitmap, and present several optimization tricks for these two algorithms. For multi-dimensional packet classification, we have implemented HyperCuts/HiCuts and the variants of these two algorithms, Adaptive Binary Cuttings, EffiCuts, HiCuts-op and HyperSplit-op. The SplitLookup algorithm has achieved up to 40Gbps throughput on TILEPro64 many-core processor. The HiCuts-op and HyperSplit-op have achieved up to 10 to 20Gbps throughput on a single core of Intel processors.

In general, we study the packet classification algorithms from both the perspectives of algorithm and rule-set features. We reveal the connections between the algorithmic tricks and rule-set features and therefore develop an adaptive framework for rule-sets with different features. We also implement various algorithms and compare the real

performance of all these algorithms. Results in this dissertation provide insight for new algorithm design and the guidelines for efficient algorithm implementation.

Acknowledgements

To my parents.

Contents

Abstract iii

Acknowledgements vi

List of Tables xiii

List of Figures xv

1 Introduction **1**

1.1 Motivation 1

1.1.1 The needs of SDN(Software Defined Networking) evolution . . 1

1.1.2 The needs of outsourcing network services 3

1.1.3 The needs of network virtualization 4

1.1.4 The advance of multi-core and commodity hardware 4

1.2 Research problems in this thesis 6

1.3 Main contributions 9

1.3.1 The algorithm design framework 9

1.3.2 Modeling the ruleset features 9

1.3.3 Evaluating multiple packet classification algorithms 10

1.3.4 PEARL: A prototype for SDN/NFV 11

1.4 Paper organization 11

2 State of art in packet classification algorithms **12**

2.1 Packet classification problems 12

2.2 Typical packet classification solutions 14
2.3 IP lookup algorithms 16
2.3.1 A basic algorithm 17
2.3.2 Typical trie-based algorithms 17
2.3.3 Shape Shifting 20
2.3.4 Hash-based IP lookup Algorithms 22
2.3.5 Conclusion 26
2.4 Multi-dimensional Packet Classification Algorithms 26
2.4.1 Complexity of the basic problem 26
2.4.2 Decomposition based algorithms 28
2.4.3 Hash-based algorithms 30
2.4.4 Decision tree based algorithms 30
2.4.5 Conclusion 32
2.5 Key idea: exploit the "sparseness" of rulesets 32

3 Anatomy of Decision Tree Algorithms: Framework and Evaluation 35
3.1 Motivation 36
3.1.1 Studying the performance variation of existing algorithms . . 36
3.1.2 Evaluating and analyzing existing algorithms 37
3.2 Background 37
3.3 The DT-based algorithm design framework 40
3.3.1 Field choosing and Field cutting 41
3.3.2 Optimization tricks 46
3.3.3 Discussion 53
3.4 Experiments Setup 54
3.4.1 Platform 54
3.4.2 Rulesets and traces 54
3.4.3 Implementations 55

3.5 Experiment Results 57
3.5.1 Comparing memory size and memory accesses 57
3.5.2 Real throughput 60
3.6 Conclusion 61

4 Meta algorithms for software-based Packet Classification 63
4.1 Motivation 63
4.2 Background and Observations 66
4.2.1 Influence on temporal performance 67
4.2.2 Influence on spatial performance 68
4.2.3 Application to existing algorithms 69
4.2.4 Discussions 71
4.3 Memory footprint estimation 72
4.3.1 Improving memory size estimation 75
4.3.2 The bound of memory consumption 76
4.3.3 Limitations 78
4.4 Characterizing range distribution uniformity 79
4.4.1 Interval tree 79
4.4.2 Characterizing the shape of interval trees 83
4.4.3 Algorithm decision framework 85
4.4.4 SmartSplit algorithm 86
4.5 The AutoPC framework 88
4.6 Experimental Methodology 88
4.7 Experiment Results 90
4.7.1 Memory Size and Real Performance 90
4.7.2 Memory estimation under different number of partitions . . . 92
4.7.3 Estimated and Actual Memory 92
4.7.4 Study the error of the memory estimation 95

4.7.5 Comparing SmartSplit and EffiCuts 99
4.7.6 Real Performance Evaluation 102
4.8 CONCLUSION 103

5 Evaluating and Optimizing IP lookup on Manycore Processors 105
5.1 Introduction 105
5.2 Background 107
5.2.1 The Tree bitmap algorithm 107
5.2.2 The DIR-24-8-BASIC algorithm 109
5.2.3 The TILEPro64 architecture 110
5.3 IP Lookup on TILEPro64 111
5.3.1 Implementation 111
5.3.2 Optimization tricks 112
5.4 Performance Evaluation 114
5.4.1 Evaluation Traces 114
5.4.2 Single-core Performance Evaluations 117
5.4.3 Parallel Performance Evaluations 118
5.5 A Hybrid IP Lookup Scheme: SplitLookup 121
5.6 Conclusion 124

6 PEARL: A Programmable Virtual Router Platform 126
6.1 Introduction 126
6.2 Design Goals 128
6.3 Platform design and Implementation 129
6.3.1 System Overview 129
6.3.2 Software Platform 134
6.3.3 Evaluation and Discussion 137
6.4 Related Work 139

6.5 Conclusion 141

7 Conclusion **142**

Bibliography **146**

List of Tables

1.1 A toy ruleset 7

2.1 a 3-dimensional ruleset 29

3.1 The performance results of different algorithms 36
3.2 An example ruleset 38
3.3 The cons and pros of all the optimization tricks. 52
3.4 The meta methods of different algorithms 53
3.5 The setup of the experimental platform 54
3.6 The node information and size of different algorithms 56

4.1 Performance comparison on different rulesets 63
4.2 An example ruleset 67
4.3 Ruleset with a lot of distinct *small* ranges on Field 1 72
4.4 the number of unique IP small ranges in large rulesets 80
4.5 Node data structure size in bytes 89
4.6 Estimated and Actual Memory size of Large rulesets 95
4.7 split distribution of each fields 96
4.8 estimate errors after restricting the split fields, $n = 16$. 97
4.9 estimate errors after restring split field, $n = 256$. 98
4.10 Detailed Information of Large rulesets 101
4.11 Real Performance Evaluation of AutoPC and EffiCuts 103

5.1 frequency and cache size 110
5.2 cache system and cache miss penalty 110
5.3 Evaluation Traces 117
5.4 update overhead of two algorithms 123

List of Figures

1.1 The core and edge of SDN 3
1.2 Multi-core and the performance of software-based router scaling . . . 5
1.3 IP lookup 6
1.4 The update frequency peak in backbone network [72] 7
1.5 The packet classification system 7

2.1 Path Compressed Trie 18
2.2 Tree Bitmap encoding 19
2.3 The Shape Shifting Encoding Scheme 20
2.4 Hash based IP lookup 22
2.5 Bloom-Filter based IP lookup algorithm 24
2.6 Distributed BloomFilter 25
2.7 the Crossproducting algorithm 28
2.8 The RFC algorithm 29
2.9 The Tuple Space Search algorithm 31
2.10 The Decision Tree based algorithm and the Decomposition algorithms 31
2.11 Exploring the "sparseness" of the rulesets 33

3.1 Decision tree algorithms 39
3.2 The decision tree design framework 40
3.3 The cutting process of HyperCuts 43

3.4 Calculating the range weight 44
3.5 Minimal cost function 45
3.6 Rule Overlap 47
3.7 Using Bitmap to reduce the memory accesses and memory size 48
3.8 Rule shifting 48
3.9 Node reuse 49
3.10 Region Compaction 50
3.11 Shape encoding 50
3.12 Meta methods of different algorithms 55
3.13 The memory access of different algorithms 57
3.14 The memory size of different algorithms (bytes/rule) 57
3.15 The memory accesses of optimized algorithms 59
3.16 The memory footprint of optimized algorithms(Byte/rule) 59
3.17 The throughput of different algorithms under the low locality traffic . 60
3.18 The thoughput of different algorithms under the high locality traffic . 61

4.1 Decision trees built by different algorithms 65
4.2 Geometric View of Packet Classification Rules 69
4.3 The distribution of $(small, small)$ rules is skewed 75
4.4 Improved Memory Size model 76
4.5 The rule splitting process in the proof 78
4.6 The interval tree data structure 81
4.7 Balanced Tree Distance and Balanced Tree Depth 83
4.8 Measuring algorithm 84
4.9 the measurement results of two interval trees 84
4.10 The AutoPC framework 88
4.11 Average Memory Access Latency and Memory Size 91
4.12 Cache Misses Rate and Memory size 91

4.13 acl10k 92
4.14 acl100k 92
4.15 fw10k 93
4.16 fw100k 93
4.17 ipc10k 93
4.18 ipc100k 93
4.19 Estimated and Actual memory size with $binth = 16$ 93
4.20 Estimated and Actual memory size with $binth = 8$ 93
4.21 The estimated and actual number of rulesets for $binth = 16$(top) and $binth = 8$(bottom) 94
4.22 A decision tree model 99
4.23 Memory and Accesses for EffiCuts and SmartSplit 100
4.24 Comparing the measured performance of SmartSplit and EffiCuts . . 102

5.1 The Tree Bitmap Algorithm 108
5.2 The DIR-24-8-BASIC Algorithm 109
5.3 Fast internal bitmap checking 114
5.4 Random Match Trace Generation 116
5.5 Single core Performance Results 118
5.6 Pipeline Parallel Performance of Tree Bitmap 119
5.7 Run-to-complete Parallel Performance of DIR-24-8-BASIC 120
5.8 Run-to-complete Parallel Performance of Tree Bitmap 121
5.9 Data Structure of the hybrid algorithm 123
5.10 Performance of our hybrid IP lookup scheme 124

6.1 Overview of PEARL architecture 130
6.2 PEARL hardware data plane architectural 131
6.3 Packet Processing Path in software 134

6.4 Throughput of high performance IPv4 virtual routers 138
6.5 Throughput of high performance IPv4 virtual routers 138
6.6 Throughput of high performance IPv4 virtual routers 139

Chapter 1

Introduction

1.1 Motivation

Packet classification enables advanced services in various network applications, such as firewalling, network intrusion detection/prevention, load balancing and QoS *etc.* In general, packet classification consists of matching packet headers against a set of pre-defined rules, and performing the action(s) associated to the matched rule(s).

As one of the key technologies in many network devices, high performance packet classification methods have been extensively studied in the past decade. Traditional packet classification systems are usually based on dedicated hardware. However, recently software based packet classification systems have attracted a lot of research interests [7, 29, 47]. We will list the reason why software based packet classification systems become important.

1.1.1 The needs of SDN(Software Defined Networking) evolution

In Software Defined networks, the control plane and the data plane are physically separated. The switches inside the network forward the packets based on the rules

installed by the control plane. This physical separation eases both the management of large and complex networks and programming network policies. To implement such an architecture, however, a general packet forwarding plane is needed to execute various dataplane lookup tasks. In the current development of SDN, the OpenFlow [38] protocol is the *de facto* standard for the abstraction of data plane function. According to the OpenFlow specification, OpenFlow switches need to provide at least flexible packet classification on 10 fields. However, the hardware resources of current commercial switches can only support a few thousands of 10 tuple rules – the TCAM volume of HP ProCurve 5406zl switch is only capable of storing 1.5 thousands rules. Such limited hardware resources severely hinder the deployment of SDN. Moreover, with the development of SDN, more and more packet header fields have been added in the specification. By the version 1.3, there are more than 30 optional fields in the OpenFlow specification. These multiple matching fields impose technical challenges to SDN development.

Martin Casado *etc.* proposed in [8] a pragmatic solution for SDN evolution. In their paper, Martin suggests to treat Core Network and Edge Network separately. The Edge Network provides interfaces between the network and the hosts, and also provides network services such as virtualization, traffic engineering and QoS *etc.* The Core Network only transfers the packets in high speed. Similar to MPLS, the edge switches add different tags into different types of packets, and deliver these packets into the network core. The network core and edge is managed by logical separated controller. Figure 1.1 shows this network architecture.

The required packet processing performance at the edge of the network is not high because of the low traffic volume of each host, but the Edge Network requires flexible packet processing. Since current network switches use specific hardware for packet forwarding, in the short term, it is difficult to provide programmability based on the specific hardware.

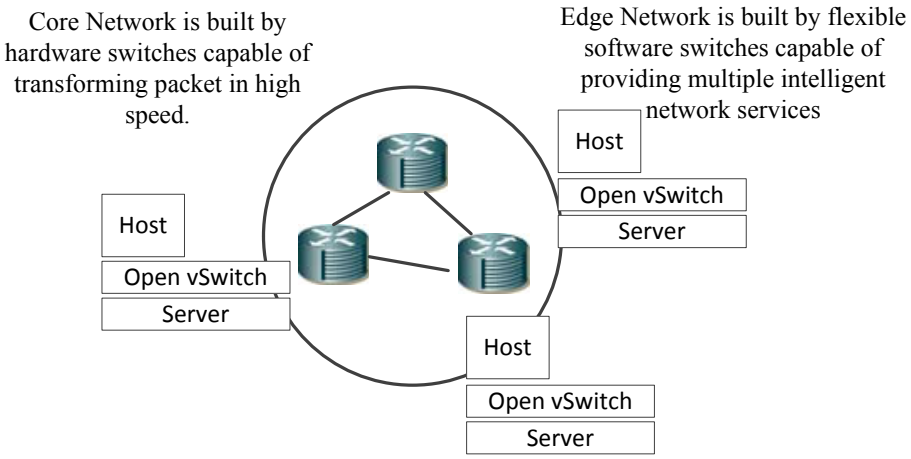


Figure 1.1: The core and edge of SDN

Therefore, Edge Network devices will be built using mainly the software-based systems on commodity hardware. Using efficient algorithms, these systems are capable of forwarding packets according to the flexible rules that specify multiple packet header fields [75]. As a typical example, Open vSwitch [46] transforms packets based on OpenFlow rules between multiple virtual machines.

The packet classification algorithm is one of the key algorithms in the above software-based systems. The study on the adaption and performance of these algorithms on software-based systems is therefore required for SDN evolution.

1.1.2 The needs of outsourcing network services

Cloud computing enables the separation of the ownership of the infrastructure and of the applications. Since the network service is one of the basic services provided by the IT infrastructure, researchers begin to explore the possibility of outsourcing network functionality to the cloud, providing network services for multiple tenants. There are already a lot of pioneers in the industry which aims to provide cloud-based network services, such as AT&T that provides cloud firewalls, reducing the CAPEX/OPEX of small enterprises.

Cisco releases a software image [66] that can be deployed directly in mainstream virtual machines as a firewall filtering packets. Many startups begin to provide cloud-based systems, such as WAN optimizer provided by Aryaka [65], cloud-based IDS (Intrusion Detection System) provided by ZScaler [76], and a middle-box architecture offered by Embrance [67].

Packet classification is an important component in the above systems. Since these systems are also built mostly on commodity hardware, the study on software-based packet classification therefore meets the needs of outsourcing network services.

1.1.3 The needs of network virtualization

The paper [43] shows that in multi-tenants data centers, the network operators need fine-grained rules to separate the network traffic between different tenants. These fine-grained rules consume a lot of hardware resources for packet processing, and even lead to network crash [64]. Experiments [43] show that when the number of newly added flows grows at 3K/s, and the number of rules reaches to 60K, Open vSwitch will consume 25% of one single CPU. Also, when the number of prefix combination increases, Open vSwitch consumes more processing resources.

Therefore efficient packet classification algorithms are needed to process more fine-grained rules and reduce resource consumption.

1.1.4 The advance of multi-core and commodity hardware

As shown in Figure 1.2, with the development of the commodity multi-core processors, software-based routers are able to achieve the same performance as the commercial routers, but with lower prices. The software routers and switches built on multi-core processors [11,16,85], the software based packet classification system Storm [37], the PEARL platform built on the commodity hardware with accelerated network cards [81], and the GPU accelerated software-based router PacketShader [24] all

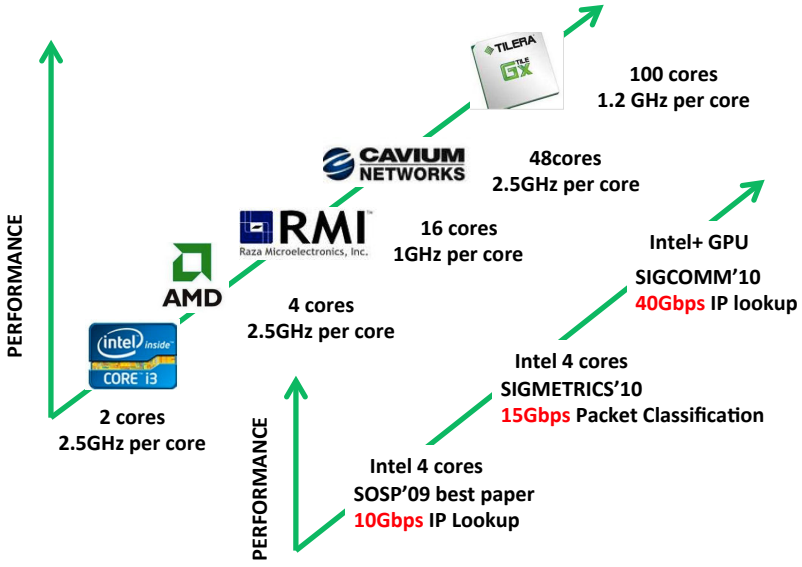


Figure 1.2: Multi-core and the performance of software-based router scaling

demonstrate that on the software platform, flexibility and high performance are not mutually exclusive.

Meanwhile, the high performance I/O libraries, such as Netmap [49, 50], DPDK(Data Plane Development Kit) [70], remove the I/O performance bottleneck in the operating system, and lower the complexity of developing network applications with high performance. The efficient packet classification algorithms running on commodity hardware therefore become a new bottleneck and have attained new research interest [32, 79, 83, 84].

Packet classification plays an important role in SDN evolution, outsourcing network services and network virtualization, so it is important to study the packet classification on the commodity hardware.

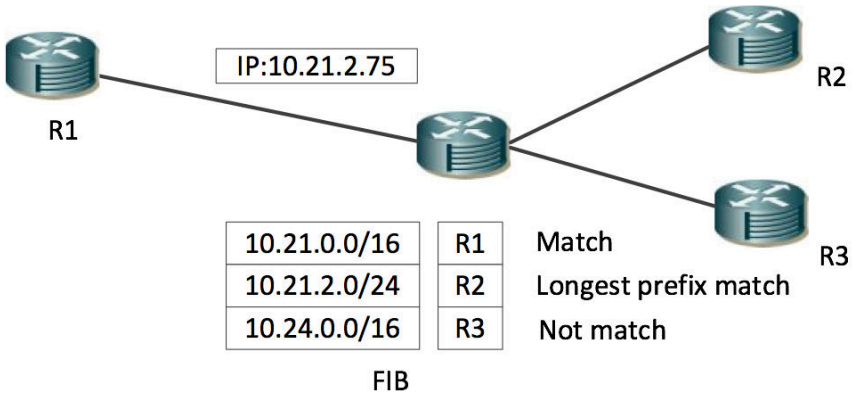


Figure 1.3: IP lookup

1.2 Research problems in this thesis

This thesis will mainly discuss two basic classification problems: the single dimensional packet classification and the multi-dimensional packet classification. For single dimensional packet classification, IP lookup on many core chips is studied. For multi-dimensional packet classification, the decision-tree based algorithms are mainly studied in this thesis.

IP lookup is actually the longest prefix matching of IP addresses. As shown in Figure 1.3, the FIB (Forwarding Information Base) contains three prefixes: 10.21.0.0/16, 10.21.2.0/24 and 10.24.0.0/16. For the IP address 10.21.2.75, the matched prefixes in the FIB are 10.21.0.0/16 and 10.21.2.0/24. Because the prefix length of 10.21.2.0/24 is longer than that of 10.21.0.0/16, the final matched prefix of IP lookup is 10.21.2.0/24.

Besides the matching speed, IP lookup algorithms should also consider the update cost of rules. According to the report of RouteViews [73], the peak update frequency can reach to 1000 times per second (see Figure 1.4 where UPDs means the number of prefix updates and WDLs means the number of prefix withdrawals). On many/multi core platforms, frequent update of rules usually leads to the severe lock overhead, reducing lookup performance.

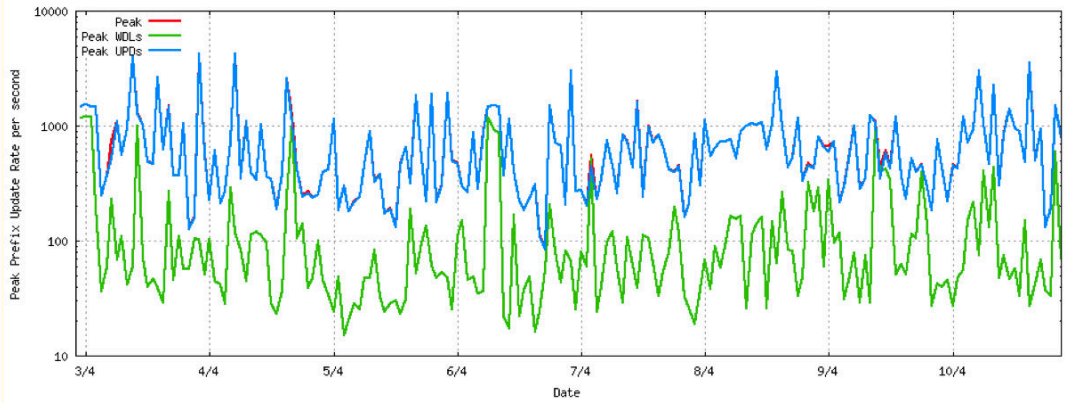


Figure 1.4: The update frequency peak in backbone network [72]

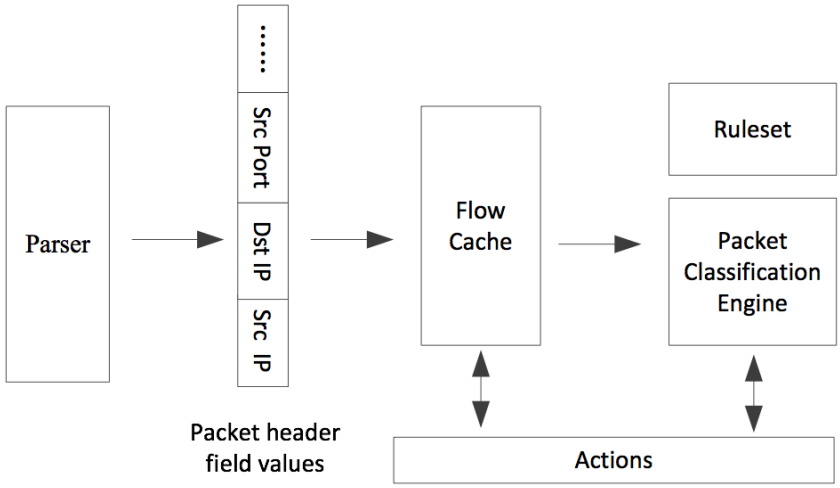


Figure 1.5: The packet classification system

	SIP	DIP	SP	DP	L4 protocol	Act.
R1	10.1.0.0/16	191.243.60.0/24	0 : 65535	1521 : 1521	TCP	DROP
R2	10.1.0.0/16	58.62.126.0/24	0 : 65535	1724 : 1724	TCP	DROP
R3	10.3.7.0/24	58.62.126.0/24	0 : 65535	1521 : 1521	TCP	PERMIT
R4	23.3.7.0/24	58.49.16.0/24	0 : 65535	14753 : 14753	TCP	PERMIT
R5	23.5.7.0/24	58.49.16.0/24	0 : 65535	5631 : 5631	UDP	RESET

Table 1.1: A toy ruleset

Table 1.1 shows a typical multi-dimensional packet classification ruleset. In Table 1.1, each row shows one rule. The column named SIP shows source IP prefixes of each rule and the DIP column shows the destination IP prefixes. SP and DP refers to source and destination port ranges that each rule defines, and L4 protocol refers

to the Layer 4 protocols defined by each rule. The abbreviation of Act. means the action that will be performed if the corresponding rule is matched.

Figure 1.5 shows a typical packet classification system. For each incoming packet, the packet classification system will first parse the packet and retrieve the value on the matched fields. These retrieved values will then be sent to the cache system for a fast lookup. The cache system caches the previous lookup results for packets belonging to the same flow. If there is a matched value in the cache system, the match action is performed for the packet; if not, the retrieved values will be sent to the packet classification engines for ruleset searching/matching. The action associated to the rule with highest priority which matches the packet will be executed. Typical actions include DROP, PERMIT, and RESET, *etc.* As the update frequency of packet classification rules is usually low, most multi-dimensional packet classification algorithms only support static rulesets.

The thesis will study the design and optimization of packet classification algorithms, including:

1. Algorithm design: we will review typical packet classification algorithms and propose an algorithm design framework. The framework will enable the other researchers to design their own algorithms based on the feature of rulesets. We also study the connection between ruleset features and algorithm performance, and propose some automatic analysis tools to choose the right algorithmic policy for a given ruleset.

2. Optimization tricks: We will evaluate the real performance of algorithms. We also propose implementation tricks, including using special instructions to accelerate key operations, and using huge pages to reduce TLB misses.

1.3 Main contributions

1.3.1 The algorithm design framework

While the goal of the thesis is to design efficient packet classification algorithms and systems, before exploring the new world, we need first review typical algorithms for both single and multiple dimensional packet classification problems. In this work, we find that most of the modern packet classification algorithms are based on the exploration of ruleset features. We therefore compare the basic idea of different algorithms from the perspective of ruleset features.

Especially, we present an algorithm framework that views the decision-tree based algorithm as a combination of "meta" methods: field-choosing methods, field-cutting methods and optimization tricks. Based on this framework, we improve the performance of the existing algorithm by combining different "meta" methods of different algorithms. We propose HiCuts-op and HyperSplit-op algorithms and the experiment results show that compared to its original algorithm, the memory footprint of these two algorithms has been reduced by $1 \sim 2$ orders of magnitudes, and the number of memory accesses has been also reduced by $30\% \sim 50\%$.

1.3.2 Modeling the ruleset features

The connection between ruleset features and algorithm performance

We find that the performance of existing decision-tree based algorithms varies for different rulesets. We therefore study the connection between the ruleset feature and the performance of the algorithm. Our research results show that the "coverage-uniformity" of the rulesets determines the number of memory accesses, while the "orthogonal structure" inside the rulesets, the memory footprint of different algorithms. For "coverage-uniformity", we present a method capable of quantifying this uniformity and choosing the right algorithm. We also propose a memory footprint

model based on the feature of "orthogonal structure" which can roughly estimate the memory footprint.

Besides the research interest, the memory footprint model can be used to estimate the memory footprint of large rulesets (100K) in seconds. And the quantify method can reveal the "coverage-uniformity" of the rulesets. These features are powerful to guide the design of efficient packet classification algorithms.

We design the SmartSplit multi-decision tree algorithm and the AutoPC framework based on the analysis of these two features. Compared to the state-of-art algorithms, the number of memory accesses of SmartSplit is reduced by 1/2 in average, and the memory footprint is reduced by up to 10 times. For a given ruleset, the AutoPC framework is capable of choosing the "right" algorithm for the ruleset. Compared to using only one algorithm, the lookup speed is increased by 3.8 times.

The SplitLookup algorithm

We also discuss the relationship between prefix length and update cost in IP lookup. We observe that the number of memory accesses is linear with the prefix length in Tree Bitmap; the update cost is small if the prefix length is short in DIR-24-8 algorithm. Based on this observation, we propose a hybrid algorithm SplitLookup. SplitLookup achieves a lookup speed closed to DIR-24-8 while its update cost is 2 orders of magnitude lower than DIR-24-8.

1.3.3 Evaluating multiple packet classification algorithms

In the thesis, we have implemented and evaluated various existing algorithms and proposed packet classification algorithms, including DIR-24-8, Tree Bitmap, SplitLookup, HyperCuts, HiCuts, Adaptive Binary Cuttings, EffiCuts and SmartSplit algorithms. The experiment results show that the proposed IP lookup algorithm can achieve 40Gbps throughput using 18 Tilera cores, and the proposed multi-dimensional

packet classification algorithm can achieve nearly 10Gbps throughput on a single Intel core under low locality traffic.

1.3.4 PEARL: A prototype for SDN/NFV

Finally, as we are targeting to solve the software-based packet classification problems in SDN/NFV, we build a prototype, namely PEARL, for testing software-based algorithms. PEARL is built on a commodity server with specified FPGA network cards with TCAMs for matching OpenFlow rules. We utilize light weight kernel-level virtual machine LXC to isolate different network applications (or we call them virtual routers). PEARL is capable of running both routing based network applications and middle-boxes network applications. We will introduce the design of the system and some primary performance evaluation results of PEARL.

1.4 Paper organization

We review the existing packet classification algorithms in Chapter 2 and propose our insight on different algorithms. Chapter 3 presents our design framework for decision-tree based algorithms. In Chapter 4, we explore the connection between ruleset features and the performance of various algorithms. We present an evaluation of typical IP lookup algorithms on a many-core processor and the SplitLookup IP lookup algorithm in Chapter 5. The PEARL system is introduced in Chapter 6. All work in this thesis will be concluded in Chapter 7.

Chapter 2

State of art in packet classification algorithms

In this chapter, we review the research on packet classification algorithms, including typical packet classification problems, and main ideas of the state-of-art algorithms for each packet classification problem. We reveal the key insight behind each algorithm from the perspective of ruleset features and conclude that almost all the modern packet classification algorithms are utilizing some "sparseness" inside the real packet classification rulesets.

2.1 Packet classification problems

Packet classification is about matching specific packet header fields against a set of pre-defined rules. Based on the number of classified fields, the packet classification problems can be categorized into two different types: one is called single-dimensional packet classification and the other is called multi-dimensional packet classification. Typical single-dimensional packet classification includes the IP lookup in the router, and the MAC lookup in the switches. Typical multi-dimensional packet classification

includes Access Control List in the firewall, the policy routing in the router, and the $L2 \sim L4$ load balancing policies in the loader balancer.

We first formalized some important terms in packet classification problems.

- *packet*

  A packet p contains d fields in its header. The value on these d fields are $p[1]$, $p[2]$, ..., $p[d]$. Each field is formed by a bit string with certain length. For the field F, $D(F)$ means the boundary of all the possible values on the field F. For example, the source port number in the TCP protocol is formed by 16 bits, so we have $D(F) = [0, 65535]$.

- *rule*

  A d-dimensional classification rule R is formed by d **ranges**, $R[1]$, $R[2]$, ..., $R[d]$ and an associated action $R.act$. For example, one IP forwarding rule is usually formed by one range on the destination field and the forwarding port as the action. If the packet p and the rule R satisfies $\forall i \in [1, d], p[i] \in R[i]$, we say that the packet p matches the rule R. We use $p \in R$ to denote this relationship. For one packet, there may be multiple rules matching it. In this case, the action associated to the rule with highest priority $R.pri$ will be performed.

In the forthcoming, we introduce the different ways of matching.

- *exact match*

  If each range $R[i], i \in [1, d]$ of one rule R is a specific value (meaning the range include only one value), and the packet p satisfies $p[i] = R[i]$, we call this an *exact match*. In the real world, the MAC lookup in the switches and the flow lookup in TCP/IP stack belong to this match.

- *prefix match*

If each range $R[i], i \in [1, d]$ of the rule R can be expressed by a prefix, also the packet p satisfies $p[i] \in p[i]$, we call the match between the packet p and the rule R a prefix matching. The most common prefix match is IP lookup in the router.

- *range match*

  Range match is the most general form in all the matches. Actually, the prefix match and exact match are special cases of range match.

We will study both the single and multiple dimension classification problems. Specifically, we study the IP lookup problem and the multi-dimensional packet classification problem. The priorities of IP lookup rules are ordered by the length of the prefixes, while the priorities of multi-dimensional classification rules are usually defined manually. The IP lookup belongs to prefix match while the multi-dimensional packet classification includes exact match and range match.

2.2 Typical packet classification solutions

Typical packet classification solutions can be categorized into two types: solutions based on TCAM (Ternary Content Address Memory) and solutions based on RAM (Random Access Memory). There are even hybrid solutions based on both TCAM and RAM [35]. Because in this thesis we do not study TCAM-based solutions, here we only briefly review the research on TCAM-based solutions.

The TCAM-based solutions provide deterministic and high performance. However, TCAMs are also expensive and power hungry devices with small capacity. Current research on TCAM-based solutions aims to reduce the power consumption of TCAM devices and increase the TCAM capacity. The key idea of reducing the power consumption is to avoid the search on all the TCAM blocks. To do so, many research works [36, 58, 82] split the ruleset into multiple small non-overlapped sub-rulesets and

install these sub-rulesets on small TCAM blocks. Before the TCAM lookup, a simple logic is used to determine the target TCAM block where the matched rule resides, then this target TCAM block is searched while other blocks are disabled. One way to increase the TCAM capacity is to reduce the storage requirement of a ruleset. The main idea is to merge the rules which have the same action [17], remove redundant rules [33], and translate the ruleset into an equivalent small ruleset [34, 39–41].

The RAM-based solutions are usually known as the algorithmic solutions. The algorithmic solutions utilize efficient packet classification algorithms and build compact data structures on RAM for packet classification, yielding cheap and power efficient solutions. However, the RAM-based algorithms have unstable performance. The algorithmic solutions usually use low latency but capacity limited on-chip memory and higher latency but large external SRAM to store the data structure, and use ASIC (appplication-specific integration circuit) or FPGA (Field Programmable Gate Array) to implement the specific algorithms. Due to the current advance in multi-core technology, many works [11, 16, 24] explore how to implement high performance packet processing systems on commodity servers. Therefore, there are also researches on the high performance software-based algorithms [32, 79, 83, 84].

While the on-chip memory of ASIC/FPGA has a very low access latency ($< 1ns$), and also the access bit width can be adjusted to adapt to the requirement of different algorithms, its capacity is quite small. The researches on algorithms focus on designing compact data structure to fit these algorithm data structures into the small on-chip memory. Meanwhile, since the classification speed is related to the number of memory accesses, reducing both the memory accesses and the memory footprint is the key challenge for packet classification algorithms. Although the Multi-core hardware is different from the ASIC/FPGA, the memory hierarchy (Cache/DRAM) of the multi-core platform also implies the importance of small memory footprint.

On commodity servers, one access to L1/L2 cache usually needs several nanoseconds while the memory access latency for DRAM is usually around 50 nanoseconds.

All in all, an efficient packet algorithm should satisfy the following conditions:

- Small memory footprint

  The size of the memory footprint will determine whether or not the data structure of the algorithms can be cached or fit into the on-chip memory. Besides, small memory footprint means that more rules can be processed with the same capacity of the RAM device.

- Fast classification speed

  Wire speed classification is necessary in many scenarios. In a 40Gbps link, the packet classification system is required to complete 60 million lookups per second. On the ASIC/FPGA hardware, the computing required by the algorithms can be implemented in parallel, therefore the number of memory accesses is usually a limited factor for the whole packet processing throughput. While in the multi-core systems, the complexity of the operations is also a limited factor for the classification speed.

As the control plane is also resource-constrained, some research work [48] proposes that the preprocessing time of algorithms is also a criterion for algorithm comparison. Long preprocessing time of the algorithms renders the packet classification system incapable of being deployed in the scenarios which require frequent rule update.

2.3 IP lookup algorithms

We now introduce the IP lookup algorithms. The IP lookup algorithms can be categorized based on the used data structure: the trie based IP lookup algorithms and

the hash based IP lookup algorithms. We will first introduce the time and space complexity of basic IP lookup algorithms and then introduce some state-of-art algorithms in each category.

2.3.1 A basic algorithm

The most basic IP lookup algorithm is based on the binary tree (or we call it single-bit trie). Assuming that the IP address is of W bits and the ruleset consists of N rules, the time complexity in the worst case is $\mathrm{O}\big(W\big)$ and the space complexity is $\mathrm{O}\big(NW\big)$. For a 32-bit IP address, a binary tree requires at most 32 memory accesses to finish the lookup. In order to promote the classification speed and reduce the number of memory accesses, multi-bit tree is then used for IP lookup. Multi-bit tree translates the sub-trees of the original binary tree into a node in the multi-bit tree. For multi-bit tree with the stride k, a node represents 2^k nodes of a binary tree, the tree depth is reduced to $\mathrm{O}\big(\mathrm{W/k}\big)$, while the space is increased to $\mathrm{O}\big(2^k NW/k\big)$.

2.3.2 Typical trie-based algorithms

The space complexity of the single bit trie is closed to optimal. However the main drawback is the large number of memory accesses. Therefore the challenge for single-bit trie-based algorithms is to reduce its memory accesses. While the multi-bit trie reduces the number of memory accesses, its memory footprint is larger as each node needs to encode multiple nodes of the binary tree. Therefore, the challenge for multi-bit tree trie based algorithms is to design compact data structures to compress the information stored in each node. We now introduce a typical single trie-based algorithm, the path compression algorithm.

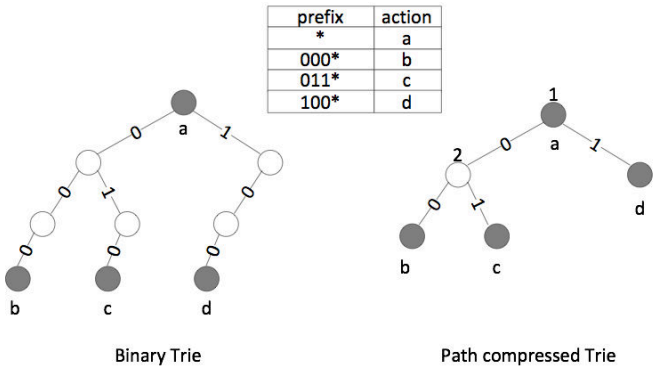


Figure 2.1: Path Compressed Trie

Path Compression

Path Compression [42, 53] collapses the single child nodes to reduce the height of the binary tree. As shown in Figure 2.1, the parents of the leaf b and leaf c are single-child node, and the ancestor of the leaf d has two single-child nodes. These single-child nodes are useless for separating the rules, and therefore can be removed in the search tree. The removal of these nodes not only saves memory but also reduce the tree height. In Path Compression Trie, each non-leaf node stores the position of matching bit. When searching the trie, at each node, the algorithm picks the specific bit in the IP address and follows the corresponding branch according to the value of the bit. To ensure a correct lookup, the leaf nodes store the concrete rule information. A complete rule checking will be performed in the leaf node. In one trie searching path, there may be multiple prefixes matching the target IP address, therefore multiple rules may be checked in the lookup of a single IP. A linear search is needed to find out the longest matching prefix.

In Figure 2.1, assuming that we need to look up the bit string 0010, the traversal path includes the root node and the node b. There are prefixes residing in both nodes. However, the bit string does not match the prefixes 000* stored in the node b, therefore the final match result is the prefix *.

Path Compression utilizes the "sparseness" in the shape of the binary tree and reduces the compress tree. If the shape of the binary tree is close to a full binary tree, the Path Compression will be degenerated to the binary tree.

Tree Bitmap

The Tree Bitmap algorithm [18] is a well-known trie-based IP lookup algorithm as it has been adopted in Cisco CRS-I router. Tree Bitmap algorithm uses two bitmaps to encode the shape of the subtrees, the internal bitmap and external bitmap. For a k-bit subtree, the internal bitmap uses one bit per node to represent the existence of the next-hop information in the special node, and the external bitmap uses one bit per branch to represent all the possible "egress" branches of the k-bit subtree. The k-bit subtree consists of at most $2^k - 1$ nodes, so the length of internal bitmap is $2^k - 1$ bits. As the k-bit subtree has at most 2^k child nodes, the length of external bitmap is therefore 2^k bits.

Figure 2.2 shows the internal and external bitmap for a 3-bit subtree.

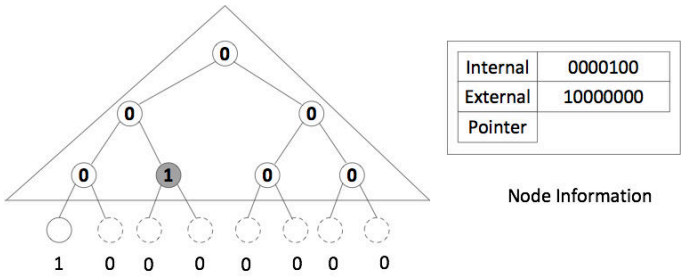


Figure 2.2: Tree Bitmap encoding

The internal bitmap is encoded by traversing the subtree in level order, and set the bit if the traversed node contains a prefix. In the subtree shown in 2.2, only the second node in the last level contains a prefix, so the internal bitmap contains only one bit setting to 1. A 3-bit sub-tree consists of at most 8 "egress" branches, and the 8 bit external nodes correspond to the existence of the 8 possible egress branches.

We see in Figure 2.2, in a typical implementation of the Tree Bitmap, each node only needs to record two bitmaps and one pointer.

Tree Bitmap compresses the memory footprint as each node only stores bits instead of pointers for non-existent egress branches. It also supports fast update. We will discuss the Tree Bitmap algorithm in detail in Section 5.2.1.

2.3.3 Shape Shifting

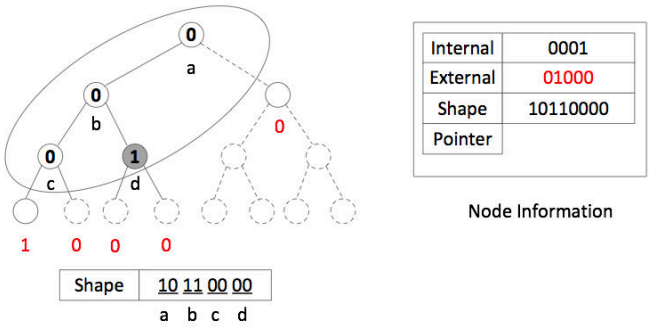


Figure 2.3: The Shape Shifting Encoding Scheme

The Shape Shifting algorithm uses the shape bitmap on the basis of Tree bitmap encoding. The shape bitmap is actually a variant of the Succinct Data Structure [26]. For any binary tree consisting of k nodes, the shape bitmap can use $2k$ bits to record the shape of the binary tree. For a binary consisting of k nodes, the internal bitmap of the Shape Shifting is of k bits, while the external bitmap is of $k+1$ bits. Therefore, each node in Shape Shifting tree needs $4k+1$ bits. Assuming that the height of the binary tree is H, Tree Bitmap needs $2 \times 2^h - 1$ bits, when the shape of the tree is quite sparse ($k \ll 2^h$), the Shape Shifting will be an attractive scheme for its higher space efficiency.

We see in Figure 2.3 that the nodes with dash line are redundant, as they do not exist in the subtree. The Shape Shifting tree does not use any bit for these nodes, while Tree Bitmap uses one bit for every possible node. However this "redundant

removal" encoding technique is harmful for fast update as one update may turn these non-existent nodes into existent nodes. In this case, the encode bitmap, even the shape of Shape Shifting trie, may need to be completely changed.

When the binary tree is very sparse (there are a lot of single child nodes, recall that $k \ll 2^h$), Shape Shifting will run faster than Tree Bitmap. This is because using the same space, Shape Shifting tree can encode more nodes, therefore increase the stride and reduce the height of Shape Shifting tree.

In essential, Shape Shifting explores the "sparseness" of the binary tree, improving both the time and space complexity of Tree Bitmap. According to [54], Shape Shifting on IPv6 rulesets runs two times faster than Tree Bitmap does.

Other Algorithms

Lulea [13] is another trie-based algorithms. Lulea splits the whole binary tree into 16, 8, 8 bit sub-trees. Besides, Lulea invents the Leaf Pushing technique that pushes the prefixes stored in non-leaf nodes into leaf nodes, reducing the memory footprint of the binary tree implementation. However, the Leaf Pushing technique renders the update of prefixes difficult. Also it is quite difficult to extend Lulea algorithm to the IPv6 addresses.

Unlike Lulea, LC trie [44] uses variable stride to adapt to the shape of the binary tree. In LC trie, the stride is different for different rulesets.

There are some other IP lookup algorithms, such as using binary search on the prefix lengths, and then performing one hash lookup for fixed length prefixes [78] *etc.* Since these algorithms are rare to be seen in the real world, we will not introduce the details of these algorithms.

2.3.4 Hash-based IP lookup Algorithms

The basic idea

Compared to exact match, the longest prefix match is in fact the match on two dimensions (length, value). As the hash lookup can only be used for exact match, for the longest prefix match, one needs to conduct multiple hash tables for different prefix lengths in the longest prefixes match. Since the IPv4 addresses is formed by 32 bits, 32 hash tables are needed for hash-based IP lookup.

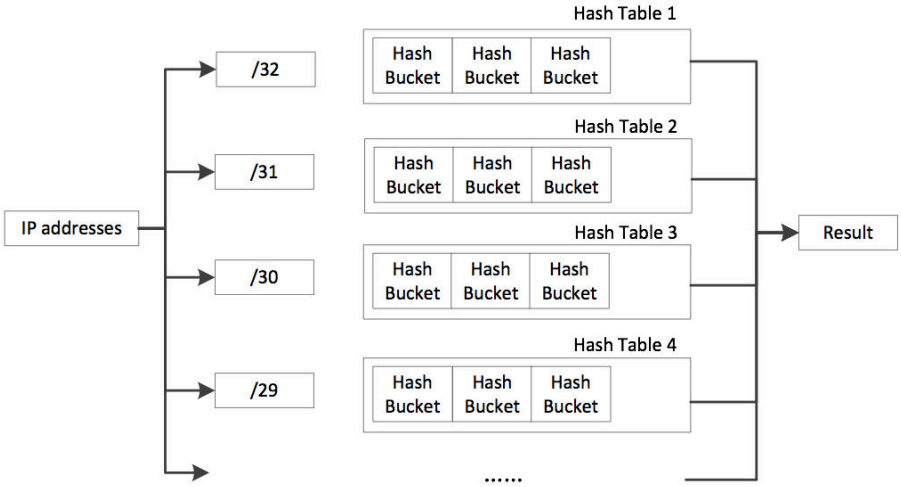


Figure 2.4: Hash based IP lookup

Figure 2.4 shows the basic idea of hash-based IP lookup. For each hash table, the IP address is masked with the corresponding mask code and then the masked value is used for hash lookup. The search results of all the hash tables will be sent to an arbitrator and the results with the highest priority (the longest prefix length) is the final lookup result.

Assuming that each hash table lookup needs only $O(1)$ lookup time, in the worst case, the basic hash-based IP lookup algorithm needs $O(W)$ time. Also, since hash-tables support update, the hash based IP lookup algorithms can support incremental rule update. However, there are still some drawbacks:

1. There may be hash conflicts during the hash lookup, resulting in a nondeterministic lookup performance.

2. Each hash table requires large memory. The hash table needs space for indexes no matter there are prefixes stored or not. The large memory footprint renders it difficult to store in the on-chip memory.

3. The length of IP prefix is not unevenly distributed in the real world, resulting in the non-uniform hash-table. Some hash tables contain a lot of prefixes while the others contain very few prefixes.

In order to overcome the issues listed above, many hash-based variants have been proposed. Here, we only discuss the Bloom-Filter based IP lookup algorithms.

Bloom-Filter based IP lookup algorithms

Bloom-Filter based IP lookup algorithms utilize the bloom filter [5] to prune the times of hash table probes required by the original one. Bloom Filter is in fact a compact representation of membership information in a set, and can be used for the membership query for this set. The key method is to use k hash functions to map all the elements to one bit string. Each element is represented by k bits stored in k locations in this bit string. BloomFilter uses very small memory footprint, therefore can be fit into the on-chip memory for fast lookup.

The basic idea of Bloom-Filter based IP lookup algorithms is to use one bloomfilter for each prefix lengths (each hash table) in the FIB. When lookup an IP address, all the bloom filters are searched in parallel to determine whether or not there is a matching prefix in a certain hash table. Only when the bloom filter reports that there maybe a prefix match, the corresponding hash-table stored in the external SRAM chips will be searched. Figure 2.5 shows the idea of this Bloom-Filter based IP lookup algorithm.

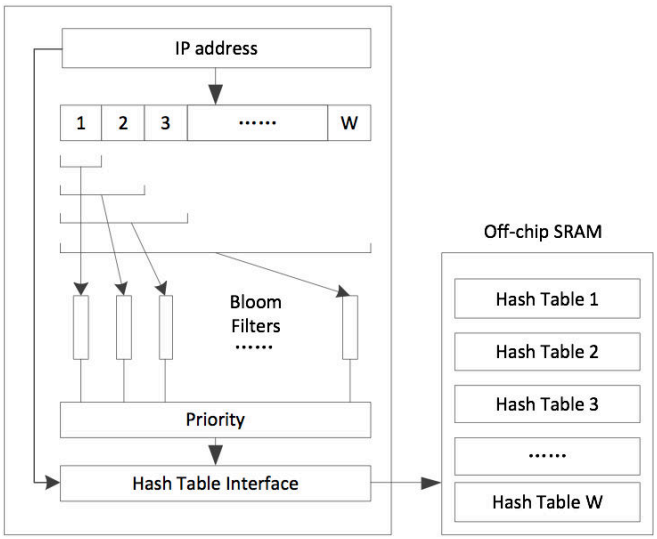


Figure 2.5: Bloom-Filter based IP lookup algorithm

The Bloom-Filter based IP lookup algorithms improve the original one by putting bloom filter instead of hash table on the on-chip memory. The on-chip bloom filter not only reduces the memory footprint, but also reduces the required hash table probes. However, as mentioned above, the non-uniform distributed IP prefix lengths make the size of bloom filters different. The paper [14] suggests to use mini-bloom filter to adapt to this non-uniformity of IP prefix length distribution. When the number of prefixes with certain prefix lengths is too large, one can use multiple mini bloom filters as one big bloomfilter to store the prefixes.

The paper [56] presents DLB (Distributed and Load balanced BloomFilter, DLB) for the non-uniformity of prefix lengths distribution. Unlike traditional bloom filters, DLB uses W same-sized bloomfilter. Each prefix, no matter its prefix length, will be hashed into W bloom filters. Each prefix takes one bit in each bloom filter. One needs to query k bloomfilters in parallel for one IPv4 address lookup. The final search result is the "AND" of all the search results of W bloom filters.

DLB is essential to divide a large bloom filter into multiple units of bloom filter. This modularized design eases the hardware implementation. Meanwhile, all the

prefixes will only use one bit in each bloom filter. In this case, no matter how the distribution of prefix looks like, there will not be a single bloom filter with higher load. DLB decouples the prefix lengths with the number of bloom filters, and solve the problem induced by the non-uniformity of prefixes distribution. Figure 2.6 shows the process of IP lookup using DLB.

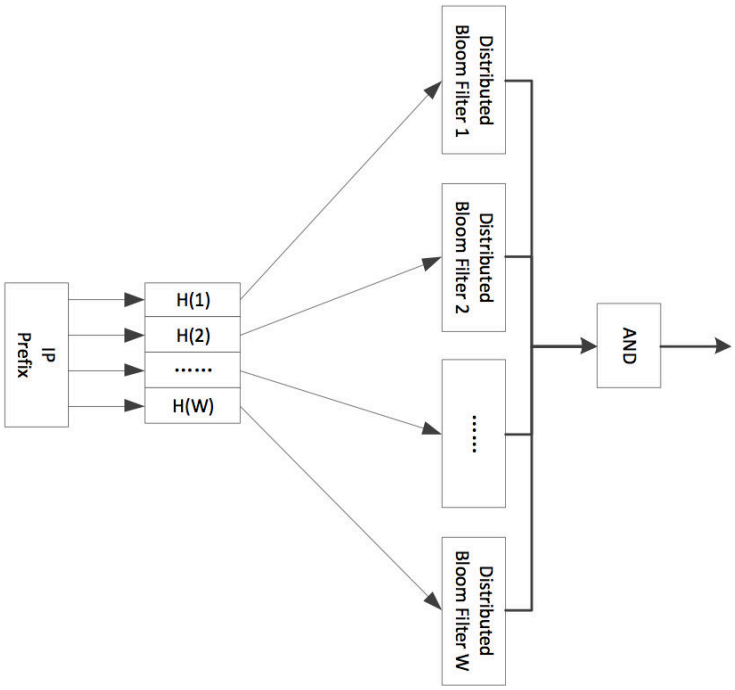


Figure 2.6: Distributed BloomFilter

Other Algorithms

Multiple hashing can be used to overcome the non-deterministic performance induced by hash conflict. The paper [60] presents "semi-perfect hash", for a ruleset, the hash table restricts the length of the conflict list into some certain value. However, generating such a hash function is very time-consuming. The paper [6] proposes using multiple independent hash functions for one item and put the item in the bucket with the smallest load. However, one needs to perform multiple hash probes for one item.

Multiple hashing does not need "semi-perfect hash" function. It uses multiple independent hash functions to reduce the length of the conflict list in one hash table. This idea can be used not only in IP lookup, but also in any search algorithm which uses hash tables.

2.3.5 Conclusion

We have introduced some representative IP lookup algorithms. We see that all the improvement of the IP lookup algorithms come from the needs to overcome some non-uniformity inside the rulesets. For example, the Path Compressed Trie and Shape Shifting Trie algorithms improve the basic binary trie through exploring the "sparseness" of the binary trie. The hash-based IP lookup algorithms are trying to overcome the non-uniformity of the prefix length distribution.

2.4 Multi-dimensional Packet Classification Algorithms

The multi-dimensional packet classification algorithms can be categorized into three types: decomposition algorithms, hash-based and the decision tree based algorithms. We will first discuss the basic idea of multi-dimensional packet classification, and introduce some typical packet classification algorithms.

2.4.1 Complexity of the basic problem

The simplest algorithm is to perform linear search of the rulesets. For a ruleset consisting of n rules, the linear search needs $\mathrm{O}\big(n\big)$ time to finish the searching, and needs $\mathrm{O}\big(n\big)$ space for storage. The algorithm uses the minimal space, but uses too much searching time. It cannot be used in the high-speed packet classification system.

In theory, the general packet classification problem has a very high degree of complexity. Previous literature [9,10] states out that to search n ranges on k dimensions, one needs either $\mathrm{O}\Big((\log n)^{k-1}\Big)$ time for linear space or $\mathrm{O}\Big(k*(\log n)\Big)$ time using $\mathrm{O}\Big(n^k\Big)$ space. For a search of 1000 rules ($n = 1000, k = 4$), one needs $\mathrm{O}\Big((\log n)^{(4-1)} = 1000\Big)$ memory accesses or needs $\mathrm{O}\Big(n^k = 10^{12}\Big)$ space. This lower bound states out that using algorithmic solution for packet classification, one needs either long search time or large space. Neither is acceptable for the real system.

Fortunately, Gupta and McKeown *etc.* [23] investigated many real rulesets and find that the real rulesets are usually sparse; this sparseness can therefore be exploited to design heuristic algorithms for high-speed classification. Their observations are listed below:

- In typical rulesets, the prefixes are usually non-overlapped. The ranges are usually small in most rules.
- There are very few unique protocols specified in one rule. Most rules are specified for TCP/UDP protocols. Very few rules are specified for the ICMP, IGMP protocol.
- Very few rules overlap with one another, meaning that one packet will match few rules at the same time.
- One packet will match at most 20 source and destination IP prefixes specified in one ruleset, which means that using source and destination IP field can separate most of the rules.

Due to the high degree of complexity in theory, almost all the multi-dimensional packet classification algorithms are based on the exploration of the features of the real rulesets, and no algorithms can be efficient in all the rulesets. One algorithm may have a performance variant in different rulesets. If one ruleset does not have the

features that are assumed by one algorithm, the algorithm will not run fast on this ruleset.

2.4.2 Decomposition based algorithms

The search on multiple dimensions can be decomposed into the combination of the search on single dimension. Concretely, one can first perform longest prefix lookup on the single field $p[0], p[1], \ldots, p[d]$ in the packets and combine the results to retrieve the final search result. Figure 2.7 shows the Crossproducting algorithm [61].

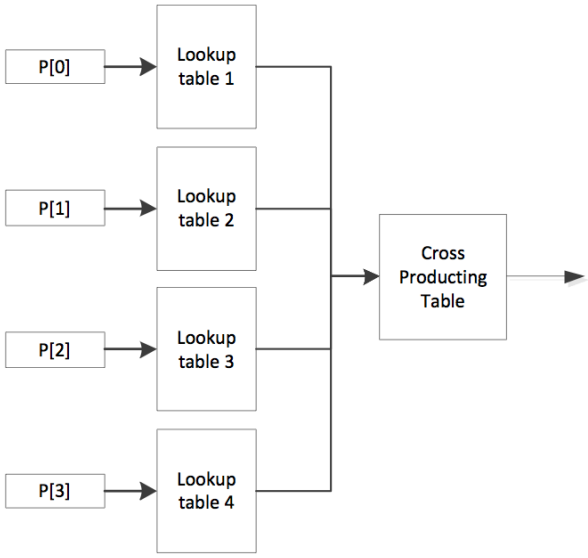


Figure 2.7: the Crossproducting algorithm

Since the search on single field is independent, these search can be performed in parallel in hardware. The main drawback is that the Crossproducting table requires large memory. The Crossproducting algorithm needs to translate the ranges on the single field into non-overlapped ranges. Assuming that the field F contain $P(F)$ non-overlapped ranges, the crossproducting table needs $\mathrm{O}\Big(P(F_1) \times P(F_2) \times \cdots \times P(F_d)\Big)$ prefixes. Usually, for a ruleset consisting of n rules, the crossproducting table needs $\mathrm{O}\Big(n^2\Big)$ space, therefore the Crossproducting algorithm is used for small rulesets.

In order to overcome the large memory footprint of crossproducting table, the paper [23] proposes the Recursive Flow Classification algorithm. This algorithm uses middle crossproducting table to remove the redundant combination for small memory footprint. Figure 2.8 shows the RFC algorithm.

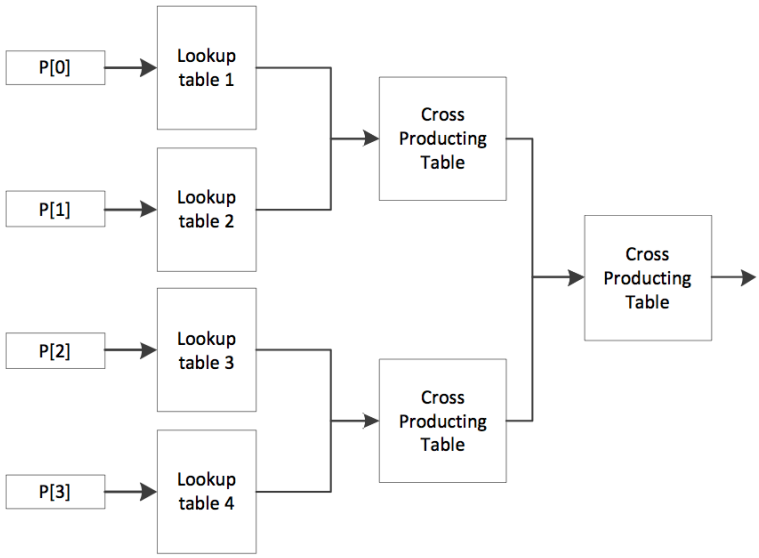


Figure 2.8: The RFC algorithm

Again, RFC utilizes the "sparseness" feature of ruleset: the number of prefix combinations in a ruleset is small. To illustrate this, we use Table 2.1 as an example. We see in Table 2.1, on the fields F_1 and F_2, the prefix combinations $\{B, C\}, \{B, D\}, \{B, E\}$ do not exist. One can merge the search results of lookup on F_1 and F_2 to eliminate these combination to reduce the memory size of the final crossproducting table.

F_1	F_2	F_3
A	C	G
A	D	G
A	E	H
B	F	I

Table 2.1: a 3-dimensional ruleset

There is some other work [3,31] which is also based on multiple fields composition. These algorithms do not rely on crossproducting table to merge the search results. Instead, they use bit vector. In these algorithms, the size of bit vector is linear with the size of ruleset. Therefore, these algorithms also need to manipulate wide bit vectors. Due to the restriction of external SRAM, wide bit vectors need multiple memory accesses to retrieve, so these algorithms are still useful for small rulesets.

2.4.3 Hash-based algorithms

Similar to the hash-based IP lookup algorithms, the hash-based algorithms for multi-dimensional packet classification conduct multiple hash-tables for different prefix combinations in the rulesets. One needs multiple hash probes for one packet. These algorithms are based on the observation that in one ruleset the number of prefix combinations is far less than that of rules. Figure 2.9 shows a typical hash-based algorithm, Tuple Space Search [59]. Similar to IP lookup, Tuple Space Search masks the search key with different mask codes and performs the hash probe on the corresponding hash table. In order to reduce the number of hash probes, one can use the longest prefix search on a single field to prune the search space of prefix combinations. As an important feature of TSS is that it supports incremental update, it has been used in Open vSwitch [46].

Similar to IP lookup, there is also other work [15] using bloomfilter to prune the number of hash probes.

2.4.4 Decision tree based algorithms

There are many recent works about the decision tree based packet classification algorithms [20, 48, 55, 77, 80]. Decision-tree based algorithms can run as fast as the RFC algorithm does, however uses fewer memory.

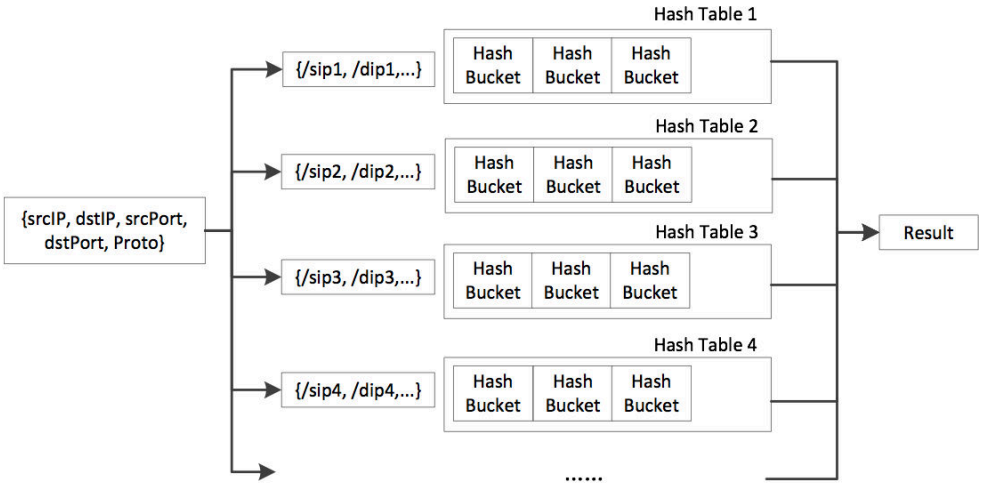


Figure 2.9: The Tuple Space Search algorithm

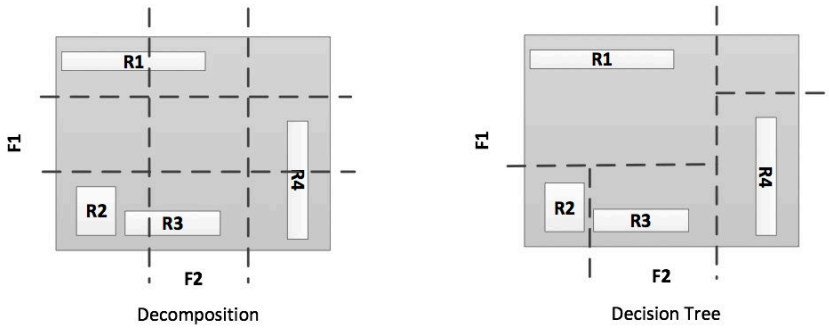


Figure 2.10: The Decision Tree based algorithm and the Decomposition algorithms

Figure 2.10 compares the decision tree based algorithms and the decomposition algorithms from the geographic perspective. A multi-dimensional rule R can be viewed as a "hyper-rectangle" in the multi-dimensional space, and a packet can be viewed as a point in this space. The search on rulesets is in fact to decide which "hyper-rectangle" the point falls in. We can define the Cartesian product $D(F_1) \times D(F_2) \times \ldots D(F_d)$ as the search space.

Different algorithms are actually using different cutting methods to reduce search space until it includes one or a few rules. According to Figure 2.10, the decision-tree based algorithm is essential to cut the search space iteratively. In each cut sub-space, the algorithms choose different fields to cut based on the feature of the ruleset included

in the sub-space. However, the decomposition algorithms are just cutting the whole space on all the fields from the beginning. Therefore, compared to the decomposition algorithms, the decision-tree based algorithms are capable of adapting to the ruleset feature, and are likely to use less time and smaller space. The decision tree based algorithm is like the Path Compression Trie on multiple dimensions. Both of them are based on the similar data structure and the same lookup process (lookup the trie and check the rules).

The decision-tree based algorithm [22, 52, 80] is an important branch of the multi-dimensional packet classification algorithms. We will discuss the details in Chapter 3.

2.4.5 Conclusion

In this section, we discuss the multi-dimensional packet classification algorithms. We see that similar to IP lookup, most multi-dimensional packet classification algorithms are based on the exploration of some "sparseness" of the rulesets.

2.5 Key idea: exploit the "sparseness" of rulesets

Efficient packet classification algorithms have been studied for more than 20 years. There are more than ten representative algorithms in both IP lookup and multi-dimensional packet classification.

We have introduced the basic and the typical algorithms in both single and multi-dimensional packet classification. In the review, we can see that most of these algorithms are based on some observation of the "sparseness" feature of the rulesets. Here we will give one example to illustrate how these algorithms utilize the features of the ruleset.

Assuming that we have the ruleset shown in Figure 2.11. The ruleset includes two IP prefixes 10000* and 10001*. As shown in Figure 2.11, the binary tree needs 6 memory accesses for the search. However, the Path Compressed Trie compares the fifth bit of the IP address, and separates the ruleset into two sub-rulesets including only one rule. Therefore it only needs 1 memory access for trie lookup and 1 memory access for rule checking. The hash-based algorithm finds that the two rules have the same prefix length, so we can conduct one hash table for the ruleset, and only one hash probe is needed for searching the ruleset. Tree Bitmap uses multi-stride with compact encoding technique to reduce both the memory size and accesses while Shape Shifting Trie uses more compact shape encoding for future improvement. At last since TCAM supports wide search entry, it can also be viewed as a solution for sparse rules. These algorithms utilize different "sparseness" feature of the rulesets to achieve fast and efficient searching.

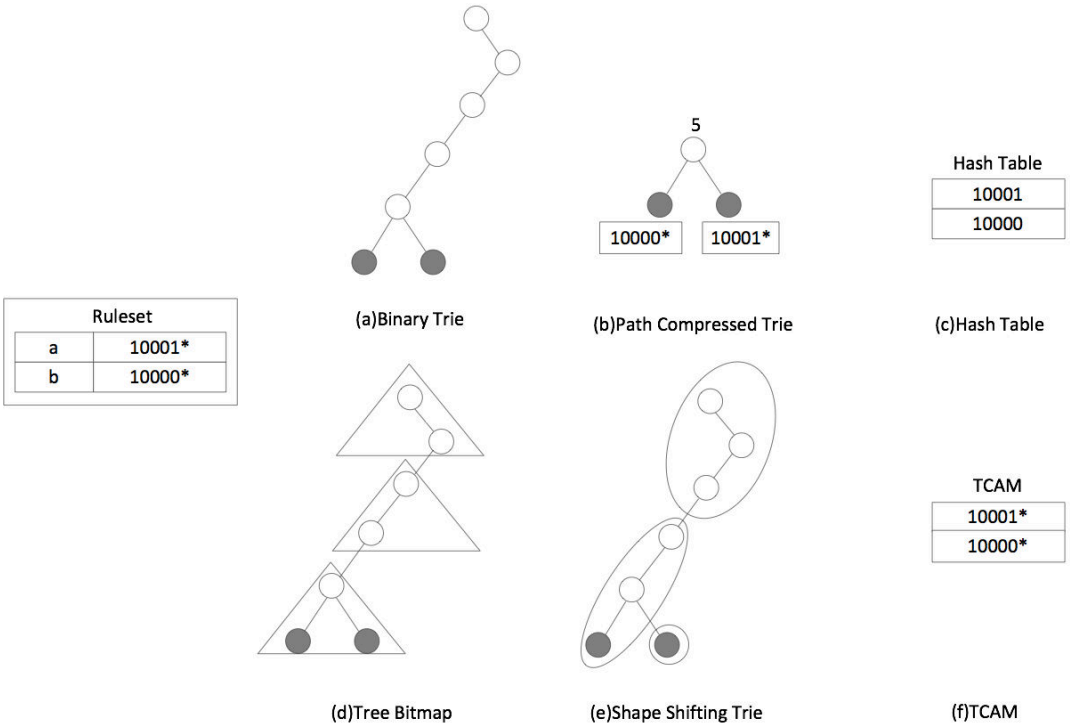


Figure 2.11: Exploring the "sparseness" of the rulesets

The research goal of algorithmic solutions for packet classification is to explore the "sparseness" of rulesets to fit into the limited resources of the real systems, and to fulfill the needs of the network applications.

This chapter focuses on how different algorithms utilize the ruleset features. As a reference, the paper [51] is a survey for IP lookup algorithms and the paper [57, 62] are surveys for multi-dimensional packet classification algorithms.

Chapter 3

Anatomy of Decision Tree Algorithms: Framework and Evaluation

Packet classification algorithms have been extensively studied. In the past decades, as an important branch of packet classification algorithms, many decision-tree based algorithms, such as HiCuts [22], HyperCuts [52], HyperSplit [48], have been proposed. These algorithms adopt different heuristics and space division methods, achieving high performance on different rulesets. Therefore, before exploring the new world, it is necessary to evaluate the existing algorithms.

In this chapter, we review four typical decision-tree based algorithms and present a design framework of decision tree algorithms. In the design framework, we view each decision tree algorithm as a combination of three types of "meta" methods -- the heuristic of choosing fields to partition, the space division methods and the tricks that optimize the division results. We call these three types of method field choosing, field cutting and optimization tricks respectively. We show that most of existing decision tree based algorithms can be adapted into this framework, and the

performance of the algorithms can sometimes be significantly improved by combing different "meta" methods from the existing algorithms. We evaluate the classification performance of existing and hybrid algorithms, the performance results show that the hybrid algorithms outperforms the existing algorithms in both the memory access and memory size.

3.1 Motivation

3.1.1 Studying the performance variation of existing algorithms

Algorithms	Ruleset	Memory size	Memory access
HyperCuts	IPC_10K	78MB	67
	FW_10K	2.2GB	51
ABC-I	IPC_10K	5MB	70
	FW_10K	9MB	48
HyperSplit	IPC_10K	11MB	38
	FW_10K	770MB	53

Table 3.1: The performance results of different algorithms

In the evaluation of existing algorithms, we found that different algorithms perform quite differently on different rulesets. We show the performance results of three typical algorithms on two rulesets. We use ClassBench [63] to generate one IPC and FW ruleset IPC_10K and FW_10K. As shown in Table 3.1, the performance varies when using different algorithms on different rulesets.

We see in Table 3.1 that, ABC algorithm has small memory footprint on IPC_10K; however, its memory access is twice as that of HyperSplit. Although the memory consumption of HyperSplit is 85× larger than that of ABC on FW_10K, HyperSplit does not gain any advantages on the memory access. HyperCuts suffers on two rulesets, and its memory consumptions on two rulesets are quite different.

Table 3.1 shows that existing algorithms perform differently on different rulesets. No algorithms achieve good performance on all the rulesets. This *performance unpredictability* issue has severely hindered the adoption of packet classification algorithms in the real world. We therefore study the reason of the performance variation. In this chapter, we will study this problem from the algorithm perspective.

3.1.2 Evaluating and analyzing existing algorithms

Existing works mostly focus on proposing new algorithms, while very few of them evaluate and analyze the existing algorithms. The evaluation of multiple packet classification algorithms is needed as a performance benchmark for newly proposed algorithms. The analysis of algorithm heuristics is also important. Existing algorithms usually has some tunable parameters. In order to achieve a full understanding of the performance, researchers have to conduct more experiments to test the impact of different parameter settings on the performance results. However, by comparing different heuristics of algorithms, one can also tell the internal differences of different algorithms, understanding the advantages of algorithms without extra experiments.

The last motivation is that a full analysis and evaluation is helpful to reveal the reason of performance variation. With the evaluation results (with recent algorithms and with all types of ClassBench rulesets), we can identify that which possibility among others is related to the poor heuristic of packet classification algorithms and which is related to the features of rulesets.

3.2 Background

We now give a high-level introduction of packet classification algorithms. Table 3.2 shows a toy ruleset. We can see that, a packet classification rule is specified by the

ranges on different fields and the action. On the same field, different rules may have the same ranges. For example, R1 and R2 share the same ranges [0-3].

Rule#	F_1	F_2	$R.act$
R1	[0-3]	[5-8]	drop
R2	[2-4]	[5-7]	drop
R3	[5-8]	[2-7]	permit
R4	[0-3]	[1-4]	drop
R5	[5-8]	[1-4]	permit

Table 3.2: An example ruleset

Decision-tree (DT) based algorithms are in fact geometric algorithms. Each packet classification rule can be viewed as a "hyper-rectangle" in multi-dimensional space. Through space partitioning on different dimensions, decision tree algorithms separate rules through dividing the space into sub-spaces consisting of fewer rules, reducing the search space of the whole rulesets. Figure 3.1 shows the decision tree built on the ruleset shown in Figure 3.2.

In the decision tree shown in Figure 3.1, each node represents a sub-space of the full space (the space represented by the root node). In each intermediate node of the decision tree, the tree building algorithms will choose one or multiple dimensions (fields) to partition the space(in the figure, we use the word "cut" for this partition). The building algorithms keep cutting the space until the number of rules overlapping the subspace is less than a preset parameter *binth*. As shown in Figure 3.1, after performing Cut 1 on the root node, the full space is divided into two sub-spaces. The left sub-space contains three rules: R1, R2 and R4, while the right sub-space contains two rules, R3 and R4. In the left sub-space, the building algorithm chooses the F_2 field to partition, generating two sub-spaces consisting of R1, R2 and R4 separately. At the same time, the right sub-space is also divided into two parts (the Leaf 3 and Leaf 4). As Cut 3 crosses the rule R3, both Leaf 3 and Leaf 4 contain the rule R3. In this case, we say the rule R3 is duplicated once.

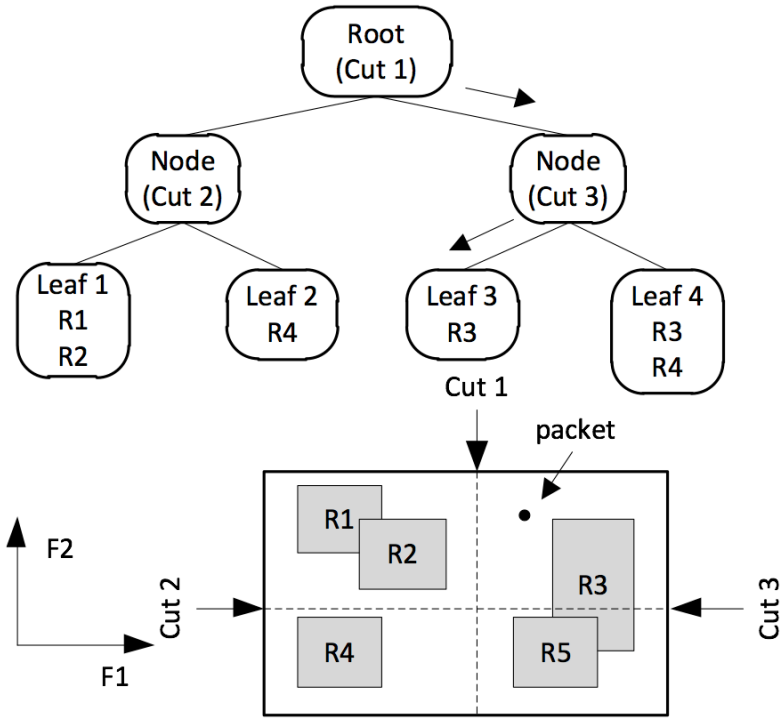


Figure 3.1: Decision tree algorithms

In this geometric view, each packet can be viewed as a coordinate in the multi-dimensional space. The packet classification is in fact about finding the most specific subspace in the decision tree where the point belongs. For example, in order to search the packet shown in Figure 3.1, the searching algorithms will first locate the point in the Leaf 3. Since Leaf 3 contains only one rule, the searching algorithm takes only one more memory access to complete the rule matching. If doing linear searching (checking rules one by one), one needs 5 memory accesses for 5 rules, but we reduce the number of memory accesses from 5 to 2 memory accesses for tree traversing plus one memory access for rule checking.

When designing packet classification algorithms, one needs to consider the following problems: 1) How to choose the dimension to cut in each intermediate node? 2) How to cut the space after choosing the dimension? An efficient DT algorithm usually should meet two requirements, small memory footprint (the number of rule

duplication is negligible) and few memory accesses (the height of algorithm tree is small). In order to achieve these two goals, algorithms need to choose the **"right"** dimension to perform **efficient** space division to separate rules.

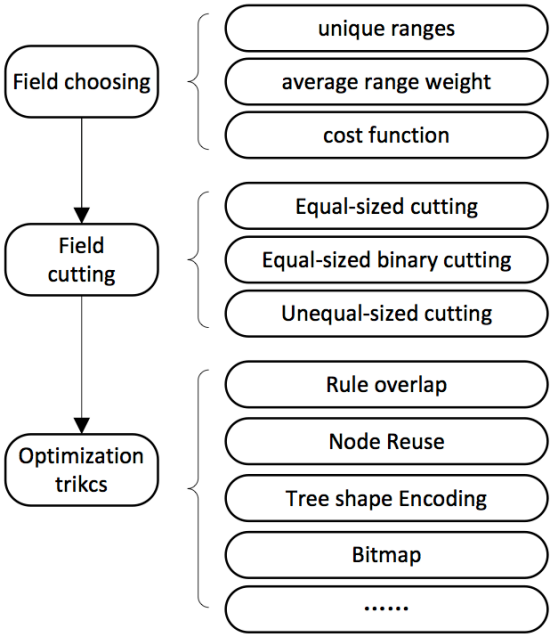


Figure 3.2: The decision tree design framework

3.3 The DT-based algorithm design framework

In order to understand the main source of performance variation and explain the performance degradation of different algorithms, we divide the building process of a DT algorithm into three "meta" methods:

- Field choosing. Field choosing is usually based on the information of rules contained in the current intermediate node.
- Field cutting. Field cutting is usually related to search speed. Complicated field cutting methods usually slow down the searching on the CPU because each operation needs more instructions.

- Optimization tricks. After cutting the space, optimization tricks can help to encode, compress or compact the memory size and the number of memory access of the DT.

The proposed decision tree design framework is shown in Figure 3.2. Different algorithms can be viewed as a combination of different meta methods listed in Figure 3.2. We will apply this design framework to four typical algorithms and compare them by comparing their meta algorithms. The proposed decision tree design framework can reveal the connection of different algorithms and explore the new possibilities of algorithm design.

3.3.1 Field choosing and Field cutting

HiCuts

HiCuts is the first DT based algorithm. In each intermediate node, HiCuts will choose the dimension that contains the most unique ranges, and cut the field into equal-sized intervals (divide the interval into $2^1, 2^2, 2^3, \ldots, 2^k$ sub-intervals, each intervals has the same length). For example, in Table 3.2, there are 3 unique ranges on the field F_1: [0-3], [2-4] and [5-8], there are 4 unique ranges on the field F_2: [5-8], [5-7], [2-7] and [1-4]. Therefore, HiCuts will choose F_2 to cut.

HiCuts uses the cost function shown below to control the number of cuts in each node. We denote the number of cut as np. Assuming that the target node contains N rules, after choosing the cutting field, HiCuts chooses the largest possible np as long as it satisfies the condition below. Therefore, the parameter $spfac$ controls the aggressiveness of cutting. Larger $spfac$ usually means more cuts in one node and large memory footprint induced by the rule duplication, while small $spface$ trades more memory accesses for smaller memory footprint due to fewer cutting in one node. $spfac$ is usually set at $1 \sim 8$.

$$N \times spafc \geq \sum N_{leaf} + np \tag{3.1}$$

HyperCuts

HyperCuts extends HiCuts by allowing each node to choose multiple dimensions to cut. Its field choosing method is to choose the dimensions which have more than average number of unique ranges. As mentioned, on F_1 there are 3 unique ranges while on F_2 there are 4. The average unique ranges is therefore $\frac{3+4}{2} = 3.5$. Of the field F_1 and F_2, only F_2 contains excess unique ranges. Therefore, HyperCuts will also choose F_2 to cut.

Similar to HiCuts, HyperCuts adopts equal-sized field cutting. Assuming that the chosen field set is $\{F_1, F_2, \ldots, F_k\}$, HyperCuts will alternately cut each field into $2^1, 2^2, \ldots, 2^n$ intervals. Figure 3.3 shows the cutting process when the chosen filed set contains F_1 and F_2. We use $\prod nc$ to denote the product of the number of cuts on each field. $\prod nc$ is actually the total number of cuts on each node. Similar to HiCuts, HyperCuts uses the largest possible $\prod nc$ as the number of cuts in each node as long as the equation below is satisfied.

$$\sqrt{N} \times spafc \geq \sum N_{leaf} + \prod nc \tag{3.2}$$

Besides allowing multiple dimensions to cut in each node, the HyperCuts paper also proposes a lot of optimization tricks. We will introduce these tricks in the next section.

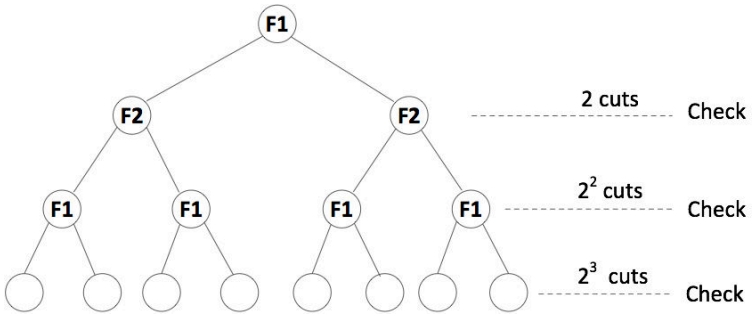


Figure 3.3: The cutting process of HyperCuts

HyperSplit

Compare to HiCuts and HyperCuts, HyperSplit [48] has different field choosing and filed cutting methods. HyperSplit chooses a single dimension and split the interval into two sub-intervals using a split value. Because the split value can be an arbitrary one, the split sub-interval may have different lengths. This is different from HiCuts and HyperCuts which only perform equal-sized cutting. We call this field cutting method the *unequal-sized cutting*.

Meanwhile, HyperSplit's field choosing method is not based on the number of unique ranges. When choosing dimension to cut, HyperSplit first translates these ranges into non-overlapped small ranges, and weights each non-overlapped range based on the number of rules intersecting the range(See Figure 3.4). In the rule-set shown in Table 3.2, the ranges on F_1 can be translated into [0-1], [2-3], [4-4] and [5-8]. Rules R1 and R4 intersect with the range [0-1] on F_1, rules R1, R2 and R4 insect with the range [2-3], R4 intersects with the range [4-4] and rules R3 and R5 intersect with the range [5-8]. Therefore, the weights of [0-1], [2-3], [4-4] and [5-8] are 2, 3, 2 and 1. HyperSplit computes the average range weight of each field, and chooses the field with the smallest average range weight. For F_1, the average range weight is $\frac{2+3+2+1}{1+1+1+1} = 2$.

In fact, this field choosing method is actually to quantify the degree of dispersion of ranges on a specific field. If on some fields, rules are evenly distributed on the

ranges and these ranges are non-overlapped, it can be expected that the average range weight of this field is small. On contrast, if rules are not evenly distributed, and there are a lot of overlapped ranges, the average range weight of the target field is large, and it is quite difficult to separate the rules by using this field.

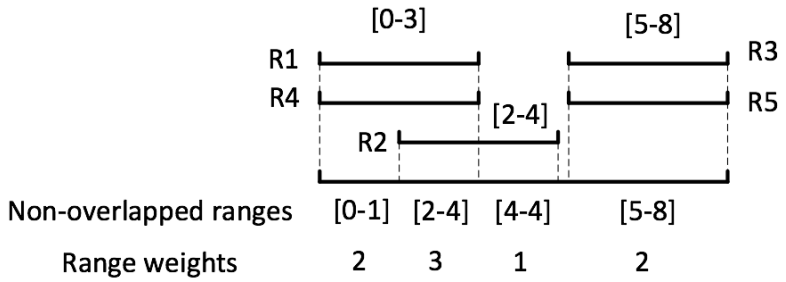


Figure 3.4: Calculating the range weight

Adaptive Binary Cutting

ABC [55] algorithm is different from previous algorithms. First, ABC uses equal-sized cutting, but it only cuts the field into two equal-sized intervals each time. Second, ABC uses a new field choosing method. In each node of ABC tree, for each field, ABC will first cut the field into two parts, and denote the number of rules overlapping with the left sub-space as R_l, and the number of rules overlapping with right sub-space as R_r. The original number of rules is R. ABC algorithm will choose the field with smallest $R_l^2 + R_r^2 - R^2$ to cut. The polynomial $R_l^2 + R_r^2 - R^2$ is similar to the cost function used in linear regression, so we call this field method the minimal cost function for short.

Figure 3.5 shows the field choosing method of ABC. In each node, ABC will evaluate all the fields. If choosing F_1 to cut, due to the duplication of rule R3, the cut subspace contains three rules each, therefore the "cost function" will be larger than choosing F_2. In fact, the field choosing method of ABC is to quantify the cost of each binary equal-sized cut and choose the field with the smallest cost. When

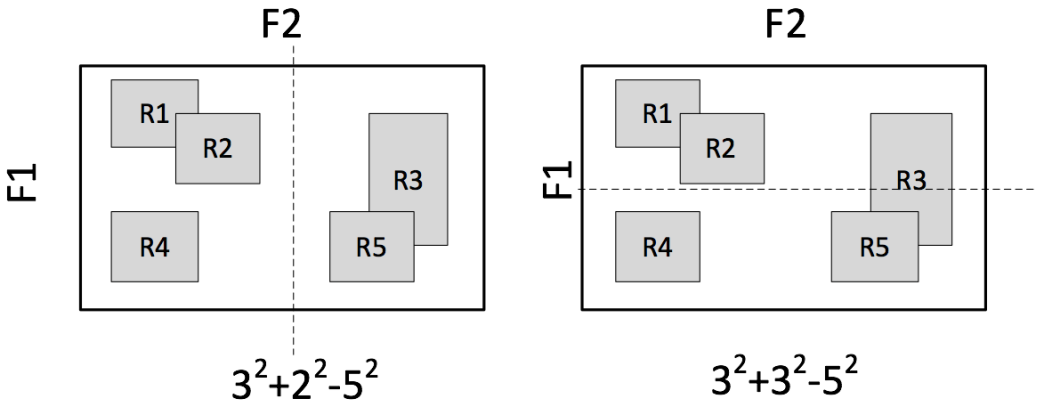


Figure 3.5: Minimal cost function

$R_l = R_r = \frac{R}{2}s$, the cost function $R_l^2 + R_r^2 - R^2$ achieves the smallest value. From the information theory perspective, as the ruleset is divided into two sub-rulesets containing equal number of rules, we say that this cut achieves the largest information gain.

Because each time the ABC algorithm only divides the space into two subspaces, the ABC tree is a binary tree without single child nodes. For such a binary tree, the succinct data structure [26] can be used to encode the shape of the tree into a bit string. Therefore, one can divide the built binary tree into fixed sized sub-trees, and encode these sub-trees using succinct data structures. This encoding reduces the number of memory accesses during the search of the ABC tree, and one needs only one memory access to retrieve the bit-string for checking the multi-bit sub-tree. We will introduce this data structure in the next section.

Comparing field choosing and field cutting methods

We have introduced the field choosing and field cutting methods of four efficient DT based algorithms. By comparing these methods, we make two key observations here:

- The field choosing methods of HyperSplit and ABC methods are more sophisticated than that of HiCuts and HyperCuts. The field choosing methods of

HiCuts/HyperCuts are only based on the number of unique ranges, ignoring the ranges overlapping information in the rulesets. The key difference is that unlike HyperSplit/ABC, the HiCuts/HyperCuts algorithm lacks an evaluation of the cutting cost when choosing fields. This will usually result in inefficient rule separating.

- The cutting of HyperSplit/ABC is fine-grained. These two algorithms only perform binary cuts in each node. After each binary cut, the algorithm will again choose the right field to cut. This frequent field choosing actually makes the cutting more efficient, because at each tiny step, the algorithm tries to choose the best field for separating rules.

3.3.2 Optimization tricks

The optimization tricks are used after cutting the nodes. They are usually used to optimize the cutting results for further cutting. In this section, we will introduce 6 popular optimization tricks.

Rule Overlap

Rule Overlap is to remove the redundant rules that are fully covered by the rules with high priority in the cut sub-spaces. Figure 3.6 shows the idea of Rule Overlap. In the figure, R1 is the rule with higher priority. In the right sub-space, the space represented by R2 is fully covered by R1, meaning that if the packet falls into this area, it will match R1 instead of R2. Therefore, in the built decision tree, R2 can be removed in the right child node to save the memory.

In practice, Rule Overlap can reduce the memory size of algorithms significantly [80]. Another benefit of using this trick is that unlike other tricks, Rule Overlap will not complicate the searching of decision tree. The only overhead is that it makes the building process time-consuming as for each rule, Rule Overlap needs to check if

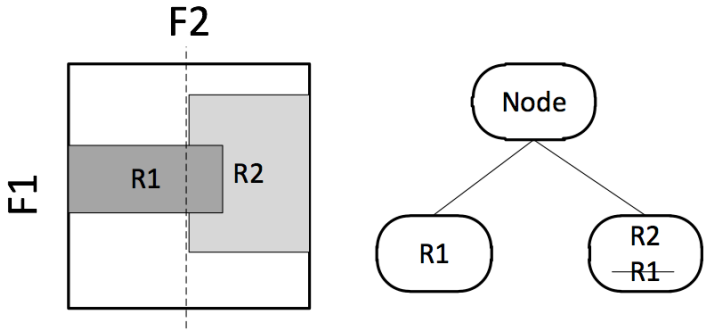


Figure 3.6: Rule Overlap

there is a rule with high priority which fully covers it. This usually leads to a $O(N^2)$ time complexity.

Bitmap

In the original implementation of HyperCuts, each node contains head information and one pointer pointing to an array of pointers; each pointer points to one child node. This pointer array worsens both time and space complexity of DT algorithms. First, this pointer array itself consumes large memory. Second, thanks to the pointer array, it needs two memory accesses, one for node, the other for the pointer array to locate the position of the child node. One can reduce the extra memory size and memory accesses using Bitmap. This bitmap is actually the encoding of the pointer array. Each pointer has its own bit in the bitmap. The bit is set to 1 if the corresponding pointer is not a null pointer. When locating the child node, one needs first find the bit position of the target child, and then count how many bits are set to 1 in front the target bit. This bit count is the offset value which can be used with the base pointer to locate the child node. Figure 3.7 shows how this bitmap trick works. The bitmap of node A is 101. If we need to locate the position of F, as shown in left sub-figure, F corresponds to the third child in the original array, one needs to count how many

set bits before the third bit. In this case, the number of the set bits is 1, so the F can be located through $baseptr + 1$.

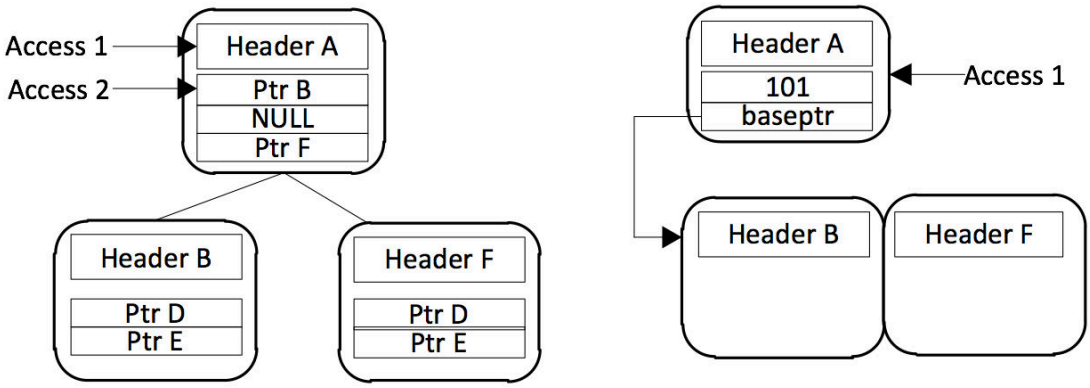


Figure 3.7: Using Bitmap to reduce the memory accesses and memory size

This bitmap trick can eliminate the extra memory access. However, since the size of bitmap is limited, the number of cuts per node is therefore limited.

Rule shifting

Rule shifting is to shift the rules contained by all child nodes to the parent node. This technique reduces the rule duplication. However, one needs more memory accesses to retrieve the shift rules and perform matching. Figure 3.8 shows this optimization tricks.

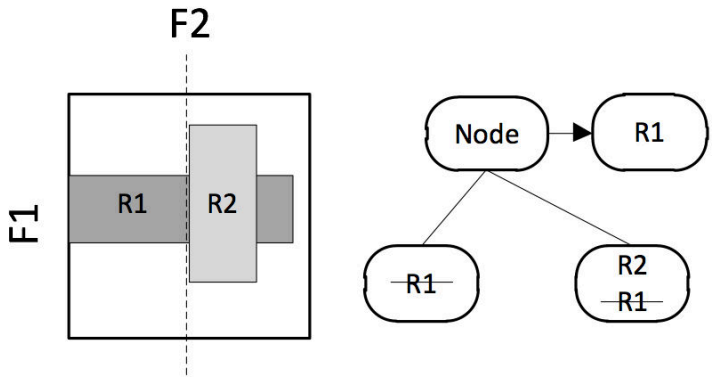


Figure 3.8: Rule shifting

Node reuse

Real rulesets usually exhibit a non-uniform range distribution. In this case, equal-sized cutting will generate a lot of identity child node (child nodes contain the same rules). As shown in Figure 3.9, after the cuts, node A will have two identity nodes, both containing rules R1 and R2. One can eliminate this redundancy by setting the two pointers pointing to the same node. In fact, this trick merges some inefficient cuts to adapt to the non-uniform range distribution. In Figure 3.9, the node reuse technique turns equal-sized cuts (1/3 per node) into unequal-sized cuts (2/3 and 1/3).

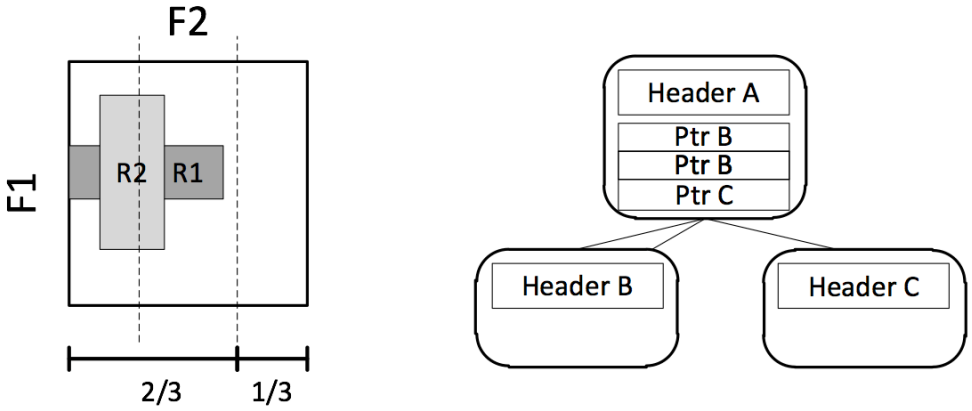


Figure 3.9: Node reuse

Region Compaction

When the boundary of the rules does not reach to the boundary of the subspace, one can compact the boundary of the subspace to make the space division more efficient (see Figure 3.10). However this technique requires each node to record the boundary information of the subspace, since the compaction makes the boundary irregular. For an IPv4 address, the boundary information requires at least 8 bytes per node, which is not a negligible overhead in many DT algorithm implementations.

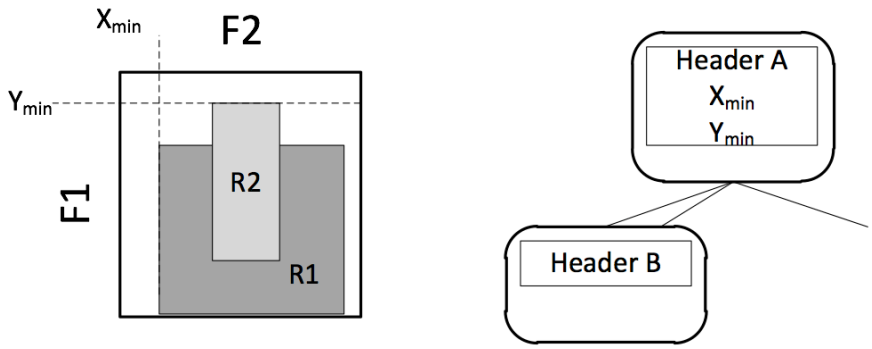


Figure 3.10: Region Compaction

Shape encoding

Shape encoding is used in ABC algorithm, while it also has applications in IP lookup [54]. Shape encoding is based on the succinct data structure [26] which encodes the shape of a binary tree into a bit string. Concretely, the encoding begins by traversing the tree by level, and set the bit of traversed node with child nodes at 1 and that with zero child nodes at 0. As the root node always has two child nodes, one can save one bit for root node. We show a sub-tree of ABC tree in the left part of Figure 3.11. The shape code of this tree is 010100. The sub-tree contains four leaves. Therefore the node A of the ABC tree shown in the right contains four child nodes. The shape encoding is very compact. Encoding a tree with k leaves requires only $2k - 2$ bits.

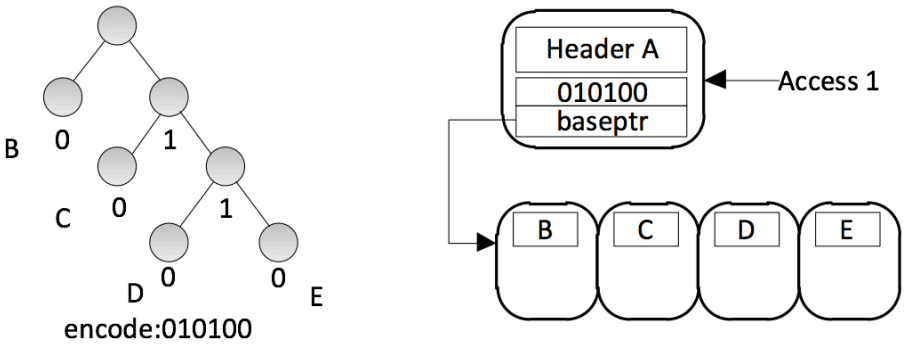


Figure 3.11: Shape encoding

While shape encoding can reduce the memory size of DT, it slows down the searching process because the decoding requires operation bit by bit. We use $ones(i, j)$ to denote the number of "1" between the ith and jth bit in the shape code, and use $zero(i, j)$ to denote the number of "0" between the ith and $(j-1)$th bit in the shape code. Assuming that the currently decoded bit is at the ith bit in shape code S and the input bit is x, the position of the next checked bit is $2 \times ones(0, i) + x$. If $S(i)$ equals to 0, we have already reached to the leaves of the tree, similar to the bitmap trick, $zero(0, i)$ is the offset for locating the child node; one can locate the position of child node with $baseptr + zero(0, i)$

Assuming that the input bit string $B = 111$, the shape code $S = 010100$, the decoding begins at the position $i = 0$, the first bit position is therefore $2 \times ones(0, 0) + B(0) = 1$. Since $S(1) = 1$, we continue the decoding. The next bit position is $2 \times ones(0, 1) + B(1) = 3$. Again, since $S(3) = 1$, the decoding continues. As $2 \times ones(0, 3) + B(1) = 5$ and $S(5) = 0$, the decoding completes. Because $zero(0, 5) = 3$, the position of the child node is $baseptr + 3$, the position of the node E shown in Figure 3.11.

We see that each step of the decoding relies on the result of last step. Therefore, the decoding step is difficult to parallel. To decode the shape code of a sub-tree with k leaves, one needs k clocks in the worst case. For a software implementation, each step needs several cycles to compute the number of "1" or "0" bits which is quite an overhead in data-plane.

Comparing different optimization tricks

We list all the pros and cons in Table 3.3. In practice, it is unnecessarily true that adopting all optimization tricks will result in the best performance. This is because that some tricks requires the extra information stored in the head information in the node. Moreover, combining some optimization trick will bring extra complexity. For

example, Node reuse and Bitmap trick are usually not used together. When enabling the Node reuse, the pointers in the pointer array may point to the same child node. However, each bit in the bitmap can only represent if the corresponding pointer is none or not, and it cannot tell that if more than two pointers share the same destination. In this case, the algorithm designer needs to add another data structure to record such information. This in fact adds more complexity in the searching process of DT, slowing down the final performance.

Optimization tricks	cons	pros
Rule Overlap	Reducing the memory size significantly by removing redundant rules.	Increasing the preprocessing time. The time complexity of this optimization tricks is $\mathrm{O}\left(N^2\right)$ when there are N rules
Bitmap	Accessing one node requires only one memory size. Memory size is reduced also by eliminating the pointer array	The fix-sized bitmap limits the number of cuts per node. This optimization trick also cannot be used with the Node reuse technique.
Rule shifting	Reducing the memory size by reducing the duplication of rules	Increasing the memory accesses since one needs to retrieve the shift rules for each node.
Node reuse	Reducing the memory size by reusing the child node.	Pointer array increase the memory access and requires extra memory.
Region compaction	Making the cuts more efficient by compacting the node region.	This optimization trick needs the boundary information stored in the node which increases the size of the tree node.
Shape encoding	Reducing the memory size by encoding the shape of the tree into a bit string.	The searching may be slow due to the complicated decoding.

Table 3.3: The cons and pros of all the optimization tricks.

The above optimization tricks can be categorized into four groups. Rule Overlap alone should be put into one group, because this trick can be used in any DT-based algorithms. Since Bitmap trick eliminates the pointer array, the rule shifting technique can therefore utilize the saved space for adding an extra pointer per node pointing to the shift rules. We therefore put Bitmap trick and rule shifting techniques in the same group. We put Region Compaction and Node Reuse in one group. The last group includes only shape encoding. In our evaluation, this technique is only used in the ABC algorithm.

3.3.3 Discussion

We investigate many open sourced implementations [68, 69, 77] of DT algorithms and show in Table 3.4 the meta methods of four typical DT based algorithms.

Algorithm	Meta methods		
	Field choosing	Field cutting	Optimization tricks
HiCuts	most unique ranges	equal-sized cutting	Rule Overlap + Node reuse + Region Compation
HyperCuts	average unique ranges	equal-sized cutting	Rule Overlap+(Node reuse, Region Compaction) or (Bitmap trick, rule Shifting)
HyperSplit	minimal average range weight	unequal-sized binary cutting	None
ABC	minimal cost function	equal-sized binary cutting	Shape encoding + Rule Overlap + Rule Shifting

Table 3.4: The meta methods of different algorithms

We have already compared different field choosing, field cutting methods and optimization tricks in Section 3.3.1 and Section 3.3.2. A key insight here is that the field choosing and field cutting methods can actually be decoupled, meaning that we can improve one algorithm by "borrowing" the field choosing method from

another algorithm. We acknowledge that the Field choosing methods of HiCuts and HyperCuts are coarse-gained since they ignore the status of ranges overlapping in the rulesets. We then use the field choosing methods of HyperSplit to improve HiCuts. We call this modified HiCuts as HiCuts-op.

We also find that the original HyperSplit implementation does not use any optimization tricks. Of all the optimization tricks, the Rule Overlap is the only technique we can use for HyperSplit. We therefore improve HyperSplit by adding the Rule Overlap technique. The modified HyperSplit is denoted as HyperSplit-op in the following.

3.4 Experiments Setup

3.4.1 Platform

We use a commodity server as the platform for performance evaluation. We show the detailed information about the experimental platform in Table 3.5.

Hardware	Setup
CPU	3.3GHz L1 Cache: 256KB L2 Cache: 1MB L3 Cache: 4MB
Memory	24GB

Table 3.5: The setup of the experimental platform

3.4.2 Rulesets and traces

We use ClassBench [63] to generate synthetic rulesets. We have used all the types of ClassBench including ACL, IPC, FW, in the total 24 rulesets (12 1K rulesets and 12 10K rulesets). We also use ClassBench to generate two types of traces: the high

locality and low locality traces. On each trace, we measure the average lookup time of the code as the speed of different algorithms.

3.4.3 Implementations

We implement HiCuts, HyperCuts, ABC algorithms. We use the code provided by the authors for HyperSplit [69]. HyperSplit-op and HiCuts-op are implemented based on the codes of HyperSplit and HiCuts. In order to maximize the performance, we use different programs for tree building and searching. The tree-building program will collect statistics of the decision tree such as memory size, the number of memory accesses and the number of nodes. The searching code is optimized for performance. For example, the Intel SSE instructions *popcnt* is used to count the number of "1" in a bit string.

The searching speed of each algorithm is evaluated by matching against 1 million five tuples. When performing searching, these five tuples are stored directly in the array, and are searched one by one.

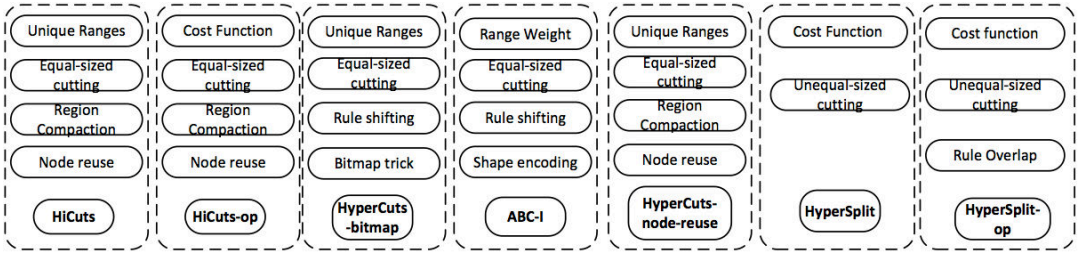


Figure 3.12: Meta methods of different algorithms

For a fair comparison, we set the $binth = 16$ for all the algorithms in our experiments. For HiCuts and HyperCuts, the $spfac$ is set to 4. The ABC paper presents three variants of ABC algorithm. In our experiments, we use the one with the best performance (smallest memory footprint and fewest memory accesses): ABC-I. For the algorithms with Rule shifting technique, the number of shift rules is set to 1. Because our platform supports 64 bits integer, we use 64-bit bit string for Bitmap

optimization trick. The configuration of meta methods in different algorithms are shown in Figure 3.12.

Algorithm	Bytes	Explanation
HiCuts	1	Head information, such as the field to cut, leaf node label*etc.*
	1	Shift bit for locating the child nodes
	2	The number of child nodes. Two bytes can support 65535 child nodes
	4	The min value of the single dimension boundary
	4	The pointer pointing to the pointer array.
HyperCuts-node-reuse	1	Head information, such as the chosen fields (at most 2) to cut
	12	The boundary information for two dimensions. Two min values are needed
	2	Shift bit for locating the child nodes
	4	The pointer pointing to the pointer array.
HyperCuts-bitmap	1	Head information
	8	Bitmap. 64 bits for at most 64 cuttings
	4	Pointer pointing to the pointer array.
	4	The shift rule pointer. Only one pointer is supported.
	1	4 bits for the shift bit on first field, 4 bits for the shift bit on second field.
	1	The total number of cuttings, at most 64.
ABC-I	1	Head information
	4	32bit for shape code capable of encoding subtrees with at most 16 leaves
	2	bitmap for at most 16 child nodes.
	4	One shift rule pointer
	8	The field to cut for each leaf; 3 bits per field.
	1	The pointer pointing to the child node pointer array.
HyperSplit	8	The split value, the child node pointer *etc*

Table 3.6: The node information and size of different algorithms

We show the size and stored information of the node in different algorithms in Figure 3.6. The HyperCuts variant with the Bitmap trick is denoted as HyperCuts-bitmap, while the variant with the Node reuse technique is denoted as HyperCuts-node-reuse.

3.5 Experiment Results

3.5.1 Comparing memory size and memory accesses

We first compare the memory size and the memory accesses of the existing algorithms. The number of memory accesses in the worst case is used as the criterion for classification speed. In the HyperCuts-node-reuse and HiCuts algorithms, it requires two memory accesses to access one node (one for accessing the node information and one for accessing the pointer array). For simplicity, when performing linear searching, we assume that each rule needs one memory access and every shifted rule needs one extra memory access for rule checking.

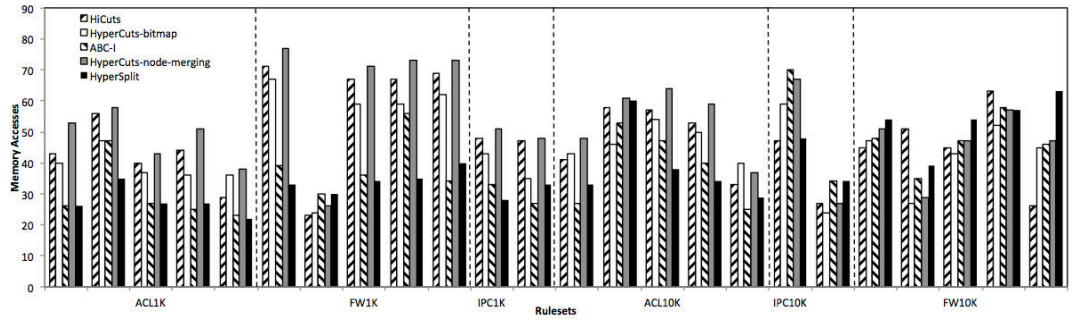


Figure 3.13: The memory access of different algorithms

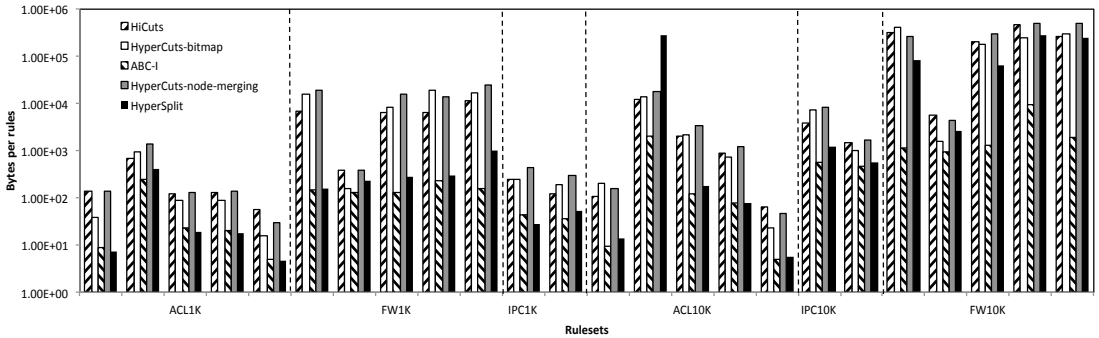


Figure 3.14: The memory size of different algorithms (bytes/rule)

We show the memory footprint and the memory accesses of different algorithms in Figure 3.13 and Figure 3.14. As shown in Figure 3.14, of all the existing algorithms,

the ABC-I has the smallest memory footprint. On some rulesets, especially the firewall rulesets, the memory footprint of ABC-I algorithm is one to two orders of magnitude smaller than that of HiCuts and HyperCuts. The memory footprint of HyperSplit is close to that of ABC-I on some rulesets, while on other rulesets, such as the FW10K rulesets, the memory size is $10 \sim 100\times$ larger than that of ABC-I.

We see in Figure 3.13, compared to other algorithms, HyperSplit requires only half of the memory accesses on some rulesets. However, on some FW rulesets, HyperSplit has severe performance degradation. As shown in Figure 3.13, the number of memory accesses of HyperSplit doubles on the FW5_10K ruleset compared to HiCuts.

We can draw some conclusion from the existing experiment results:

1. As stated, we test two variants of HyperCuts, one with rule shifting and bitmap trick, the other with node reuse and region compaction. In the results shown in Figure 3.13 and Figure 3.14, we can see that these two variants achieve close performance results. The node reuse trades memory accesses for transforming the equal-sized cutting into unequal-sized, while the bitmap trick eliminates the extra memory accesses at the cost of incapable of performing unequal-sized cutting (See the discussion in 3.3.2). These two variants achieve the same performance. We therefore conclude that the non-uniform distribution of ranges in the rulesets is the main source of the large memory accesses of HyperCuts and HiCuts.

2. While HiCuts/HyperCuts is capable of performing multiple cuts in one node, HyperSplit uses nearly equal number of memory accesses through the binary split. This shows that the HiCuts/HyperCuts fails in choosing a right dimension to cut.

3. HyperSplit and ABC-I are better than HiCuts and HyperCuts. However, HyperSplit suffers a severe performance degradation on some rulesets. We can

expect a performance improvement after we find out the reason for the performance degradation.

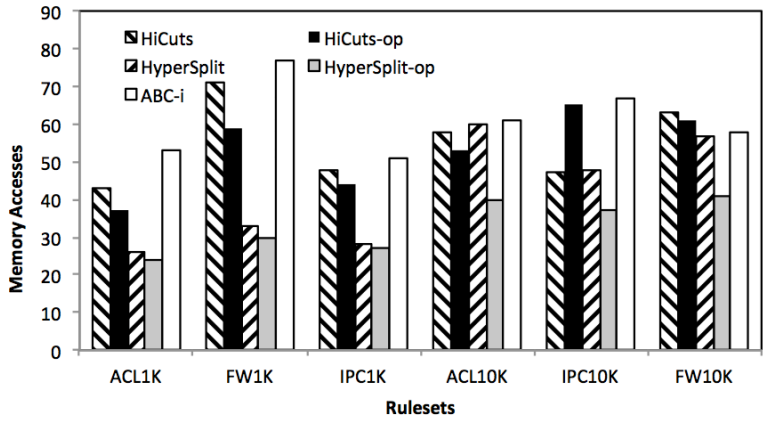


Figure 3.15: The memory accesses of optimized algorithms

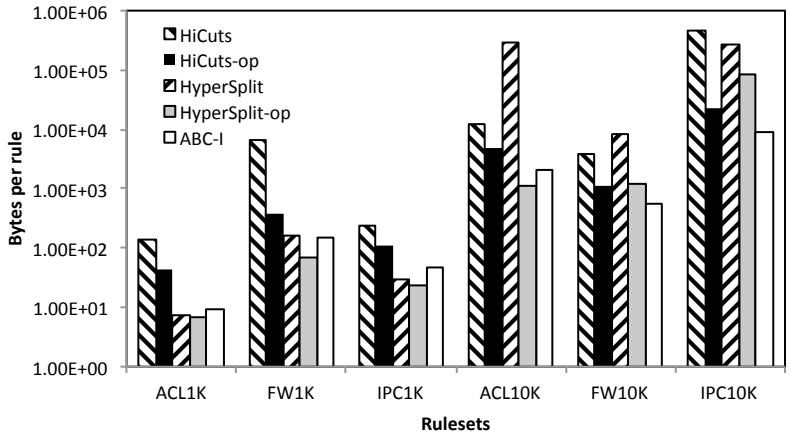


Figure 3.16: The memory footprint of optimized algorithms(Byte/rule)

We show in Figure 3.15 and Figure 3.16 the memory footprint and the memory accesses of the HyperSplit-op and HiCuts-op algorithms. We compare these two algorithms with ABC-I, HiCuts and HyperSplit. As shown in Figure 3.16, HiCuts-op achieves $2 \sim 20\times$ memory size reduction and 10% fewer memory accesses. The memory size of HyperSplit-op is $2 \sim 200\times$ smaller than that of HyperSplit, while the number of memory accesses reduces by $10\% \sim 30\%$.

We can conclude that the inefficiency of HiCuts is related to its field choosing method. The performance of HiCuts algorithm can be significantly improved by changing its field choosing method. The reason for performance degradation of HyperSplit is that the FW10K rulesets consist of too many redundant rules. After removing these redundant rules, the memory footprint and the number of memory accesses reduce significantly. Of all the algorithms, HyperSplit-op achieves the best performance.

3.5.2 Real throughput

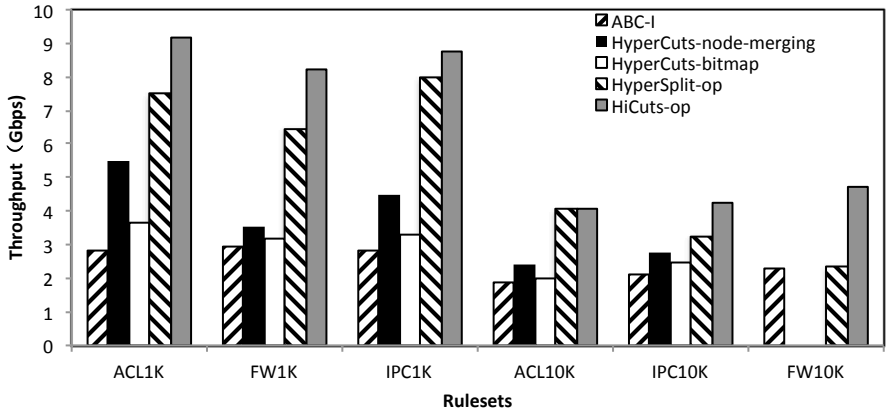


Figure 3.17: The throughput of different algorithms under the low locality traffic

Figure 3.17 and Figure 3.18 show the throughput of different algorithms under the traffic with both low and high locality. Because the large memory footprint of the HyperCuts algorithm on FW10K ($> 1G$), we did not evaluate the throughput of HyperCuts on these rulesets. As shown in the two figures, even the network link is saturated with 64 byte packets, and the HyperSplit-op and HiCuts-op algorithms are capable of processing 10Gbps traffic by using a single core. The throughput is $2 \sim 5\times$ compared to the other algorithms. Under the high locality traffic, our algorithms achieve up to 15Gbps of throughput.

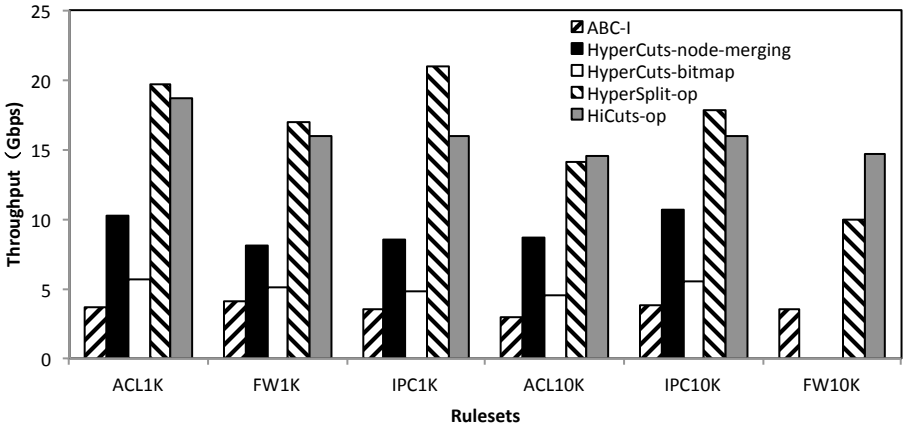


Figure 3.18: The thoughput of different algorithms under the high locality traffic

Compared to the state-of-art work [37] which achieves 15Gbps of throughput (traffic with 128 bytes packets) by performing the rule caching on 8 cores, our work demonstrates by improving the algorithm that one can achieve the same or higher throughput by using single core.

One interesting fact shown in Figure 3.17 and Figure 3.18 is that the real throughput of ABC-I is slow. This is because the decoding of shape code is slow on CPU. The HiCuts-op achieves higher throughput with more memory footprint due to the small average memory accesses for each lookup.

3.6 Conclusion

We present a design framework for decision-tree based algorithm, which views the DT based algorithm as a combination of three types of meta methods: the field choosing, the field cutting and the optimization tricks. We find that the field choosing method is more important for designing an efficient DT based algorithm.

We analyze the cons and pros of different meta methods, and find out the reason why some algorithms suffer severe performance degradation on some rulesets. We find out that the Rule Overlap technique can reduce the memory footprint of the

DT based algorithm significantly and the field choosing method based on the range weight performs better than the method based on the number of unique ranges.

We therefore improve the HiCuts algorithm by changing its field choosing method, and improve HyperSplit algorithm by using the Rule Overlap technique. The experiment results show that the memory footprint of HiCuts and HyperSplit reduces by $1 \sim 2$ orders of magnitude. These two algorithms are capable of processing 10Gbps and beyond traffic on a single core.

Chapter 4

Meta algorithms for software-based Packet Classification

4.1 Motivation

Although in the last chapter, we have revealed the reason of performance degradation in different algorithms, and proposed two new algorithms HyperSplit-op and HiCuts-op. We find that the performance variation issues still exist when the same algorithm encounters different rulesets.

Algorithm	Ruleset(size)	Memory size	Mem. accesses
HyperSplit	ACL1_100K	2.12MB	32
	ACL2_100K	83MB	43
EffiCuts	ACL1_100K	3.23MB	65
	ACL2_100K	4.81MB	136

Table 4.1: Performance comparison on different rulesets

To illustrate this issue, we present in Table 4.1 the performance in terms of memory size and maximum number of memory accesses[1] of two state-of-art algorithms

[1]The number of memory accesses is the limiting factor, and thus a direct indicator, of classification speed.

(HyperSplit[2] [48] and EffiCuts [77]) on two ACL rulesets. We can see that for two similar firewall rulesets, ACL1_100K and ACL2_100K, containing nearly equal number of rules, the memory size needed by HyperSplit for ACL1_100K is around 40 times larger than ACL2_100K (from 2.12MB to 83MB). While the memory requirement of EffiCuts on ACL1_100K and ACL2_100K are nearly equal and small, the maximum number of memory accesses needed by EffiCuts on ACL1_100K is 2 times that of HyperSplit.

These wide variations in performance demonstrate how crucial applying, in practice, the "right" algorithm to a given ruleset, actually is. For example, recent CPUs usually contain several Mbytes of last level cache and several GBytes of DRAM. Therefore, the memory size of HyperSplit algorithm on ACL1_100K can fit in the CPU's cache but the memory size for ACL2_100K cannot. Generally accessing a data in the external DRAM requires around 50 nanoseconds, while accessing a data in cache requires only $1 \sim 5$ nanoseconds, meaning that one should use HyperSplit algorithm on ACL1_100K for smaller memory size and fewer memory accesses, but should use EffiCuts on ACL2_100K to trade more memory accesses for fewer memory access latency. In general, for a given ruleset, we need to select a "right" algorithm for the memory size and the number of memory accesses trade-off.

A straightforward method to solve the problem would be to implement various algorithms on a given ruleset and choose the one with best performance results. However, packet processing platforms are often resource-constrained, and such comparison is sometimes very time consuming (*e.g.* The HiCuts [22] algorithm may need over 24 hours to process some large rulesets [48]), making this approach at best impractical, and at worst infeasible in more dynamic environments, such as OpenFlow-based networks or virtual data centers, where rulesets may change much faster than this processing time.

[2]Here, we use an improved HyperSplit implementation, see Section 4.7 for details.

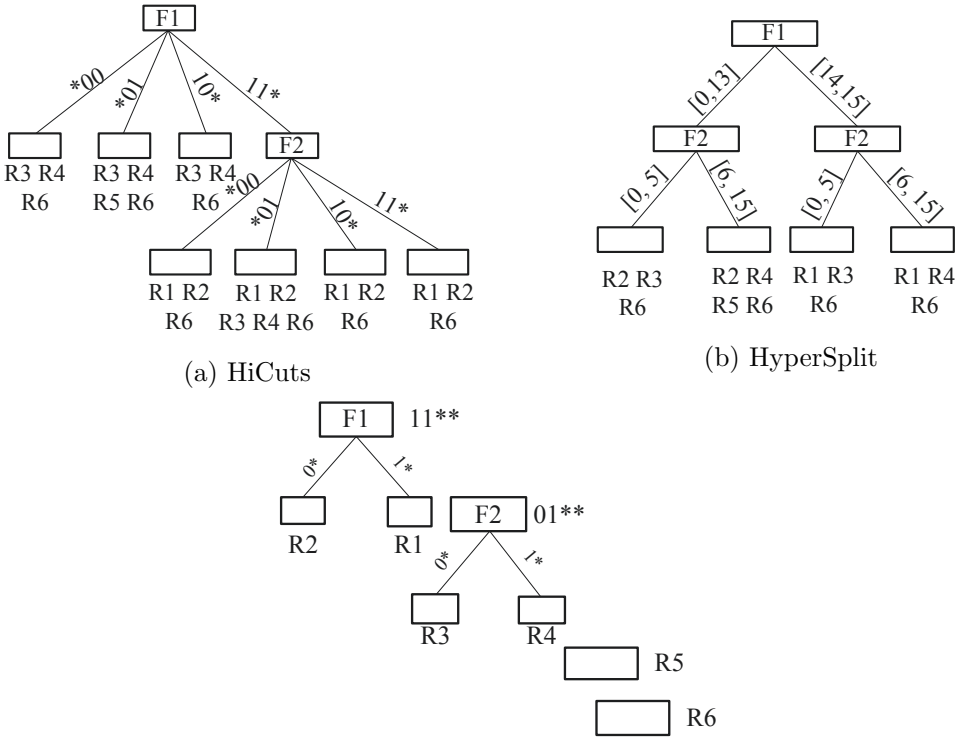


Figure 4.1: Decision trees built by different algorithms

In this work, we therefore seek to understand the reasons behind these observed temporal and spacial performance variations, with a view to quickly identify the "right" classification algorithm for a given subset. In Section 4.2, we analyze the characteristics of rulesets that do have a primary bearing on both the memory footprint and classification speed and we review three of the main state-of-the-art packet classification algorithms.

As the memory footprint of the ruleset for a given algorithm is an important factor, we present in section 4.3 a memory consumption model, to be used as a fast memory size checker, which helps to select for best memory-performance tradeoff.

In Section 4.4, we describe an offline recommendation algorithm that analyses rulesets for the above mentioned characteristics, and recommends algorithms for the given ruleset, based on classification performance alone. With this analysis tool,

we present in Section 4.4 a new multi-tree algorithm SmartSplit. The SmartSplit algorithm is built on recent work [77] that showed how to trade classification performance for much reduction in memory consumption by splitting the ruleset into several subsets and classifying against these subsets in sequence. However, going beyond [77] which uses HyperCuts [52] on every subset, SmartSplit seeks to maximize classification speed, while meeting overall memory consumption constraints, by using different classification algorithms for the stages of the classification sequence (e.g. for the various sub-rulesets). We also present a packet classification framework AutoPC in Section 4.5. The AutoPC framework, which is based on the memory consumption model, tries to further improve the performance by avoiding ruleset splitting if the memory size of rulesets is shown to be small.

Sections 4.6 and 4.7 present our evaluation methodology and experimental results, respectively. Section 4.8 concludes the paper.

4.2 Background and Observations

We first give a brief review of factors explaining why the performance of packet classification algorithms can exhibit wide variations from one ruleset to another. More detailed explanations are available in [45]. A packet classification ruleset can be considered as a collection of *ranges* defined on different fields. Table 4.2 shows an example of classification ruleset containing 6 rules defined over two 4-bit fields, where "*" represents a "don't care" value. This ruleset can be translated into four distinct *ranges* on Field1: $[14, 15]$ (defined by rule R1), $[12, 13]$ (R2), $[4, 7]$ (R5), $[0, 15]$ (all rules); and four on Field 2: $[4, 5]$ (R3), $[6, 7]$ (R4), $[8, 11]$ (R5), $[0, 15]$ (all rules).

A packet classification ruleset has a simple geometric interpretation: packet classification rules defined on K fields can be viewed as defining K-*orthotope*, *i.e.* hyper-rectangle in the K-dimensional space, and rulesets define intricate and overlapping

Table 4.2: An example ruleset

Rule #	Field 1	Field 2	Action
R1	111*	*	DROP
R2	110*	*	PERMIT
R3	*	010*	DROP
R4	*	011*	PERMIT
R5	01**	10**	DROP
R6	*	*	PERMIT

patterns of such orthotopes. For example, rule R1 in Table 4.2 defines a rectangular band over the two dimensional space of features, where the short side is 2 units long (from 14 to 15, along the axis defined by Field 1), and the long side spans the whole range of the second dimension. This structure results from the wildcard existing on the second dimension field that generates a *large range*. Similarly, rule R3 defines another rectangular region but with the short side along the second dimension.

4.2.1 Influence on temporal performance

A node in a packet classification decision tree (DT) can be considered as making a spatial partition of the geometric space into non-overlapping parts. The aim of a DT is to partition the space of features into regions that will hopefully contain the smallest number of rules. Different classification algorithms apply different heuristics for dividing the space. In particular two types of partitioning is applicable. The first type is the "cut", that consists of dividing a given range into multiple equal-sized intervals, the second type is a "split", consisting in dividing an interval at a split point into two sub-intervals, a right and a left one.

At first glance the cut-based division seems more efficient than the split-based one. Indeed, when ranges have roughly similar sizes and are uniformly distributed along a dimension, equal-sized cuts can be very efficient at separating those ranges. However, ranges observed in practice are sometimes non-uniformly distributed (*e.g.* dissimilar

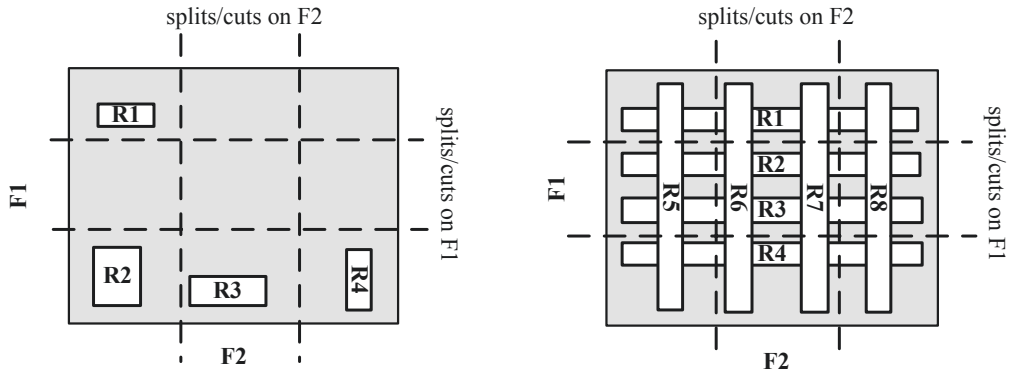
and/or in clusters along the dimension), in which case applying equal-sized cuts will become inefficient as either some cuts will simply split a rule in regions of the space where this rule has already been isolated, and/or deeper (i.e. finer-grained) cuts will be necessary in other regions, to isolate clustered rules. Under such conditions, the resulting DT would be skewed, with some branches significantly longer than others. We need to evaluate the uniformity of ranges before applying cuts or split.

4.2.2 Influence on spatial performance

In real-world rulesets, some specific patterns are commonly encountered that can have a bearing on the efficiency of the corresponding DT. Such patterns include: orthogonal structures like that resulting from rules R1, R2, R3, R4 (a more general case is show in Figure 4.2b), and sparse structures like the one defined by rule R5 (more general case is shown in Figure 4.2).

A major problem occurs when orthogonal structures are present in the ruleset. In this case, rules cannot be completely separated into regions containing a single rule with hyperplane divisions, and the best that can be achieved is to use divisions, forming $\mathcal{O}(N^K)$ regions containing K orthogonal rules, where N is the number of orthogonal rules and K is the dimension of the feature space. Moreover, each division is likely to intersect with $\mathcal{O}(N)$ other rules' subregions. When this happens, each rule that is cut has to be duplicated in the DT nodes as the cut does not separate these rules, *i.e.* rules with orthogonal structure will cause a large amount of rule duplication in Decision Tree based algorithms, creating large memory footprints.

On the other hand when the rule structure is sparse, $\mathcal{O}(N)$ spatial divisions can isolate each rule without cutting through other rules, yielding modest memory requirements.



(a) sparse rules

(b) orthogonal structure rules

Figure 4.2: Geometric View of Packet Classification Rules

4.2.3 Application to existing algorithms

We briefly describe three major packet classification algorithms proposed in the literature – HiCuts, HyperSplit and EffiCuts – and identify the specific factors that negatively impact their performance. For illustration purposes, three decision trees built on the example ruleset using three algorithms are shown in Figure 4.1 .

HiCuts and HyperCuts

We first describe HiCuts [22] and HyperCuts [52], two closely related and classical DT based algorithms. The two algorithms work essentially by cutting the full range of each dimension of the multi-dimensional feature space into equal-size intervals. In Figure 4.1a, we show the decision tree generated by HiCuts algorithm where the Field 1 is cut into 4 equal-sized sub-spaces: $[0, 3]$, $[4, 7]$, $[8, 11]$, $[12, 15]$, and Field 2 is further cut into 4 equal-sized sub-spaces. HiCut suffers from a combination of the previous described issues. On one hand as the distribution of ranges is non-uniform, *e.g.*, the ranges in Table 4.2 leaves 50% of the full range $[0, 15]$ uncovered, *equal-sized cutting* becomes inefficient as several cuts are spurious. Moreover as orthogonal rules are present, each spurious cuts, which intersects with orthogonal rules result in rule duplication in several leaves of the decision tree. As empirically up to 90% of memory

footprint of a built DT is consumed by pointers pointing to rules, rules duplication increases the memory footprint significantly.

HyperCuts which extends HiCuts by allowing to cut multiple fields in each node of the tree, suffers from the same issues caused by the inefficiency of equal-sized cuts when there are non-uniform ranges.

HyperSplit

In order to overcome the non-uniformity of range coverage described earlier, HyperSplit [48] adopts a different method to separate rules. It splits the chosen field into unequal ranges that contain nearly equal number of rules, *e.g.*, in Figure 4.1b the Field 1 is split into two unequal size intervals: $[0, 13]$ and $[14, 15]$, which separate R1 and R2 using a single memory access. In order to minimize the number of comparison, HyperSplit implements a binary tree, *i.e.*, each node contains only one split point splitting the given range into two regions.

By using *unequal-sized splitting*, HyperSplit avoids unneeded cuts reducing the memory footprint. However the main source of redundancy remains because splits intersect with orthogonal rules. Moreover, the binary tree structure adopted by HyperSplit increases the tree depth, resulting in more memory accesses than HiCuts and HyperCuts.

EffiCuts

Instead of building a single decision tree for all the rules, EffiCuts [77] builds multiple trees for one ruleset. To do so, EffiCuts categorizes the ruleset-defined ranges into *small* and *large* ranges. A range is labelled as *large* if it covers a large enough proportion, determined by a threshold of the full range. Otherwise, this the range is labelled as *small*. The threshold is set as 0.50 for most of fields. The ruleset shown in Table 4.2 has one *large range*: $[0, 15]$ and three *small ranges*: $[14, 15]$, $[12, 13]$ and

[4, 7] on Field 1. Based on this labeling one can classify each rule in the ruleset into at most 2^K categories in $\{small, large\}^K$ for a K dimensional classifier. For example, for the ruleset in Table 4.2, R1 and R2 are classified as $(small, large)$, R3 and R4 as $(large, small)$, R5 as $(small, small)$ and R6 as $(large, large)$. EffiCuts builds separate decision trees for rules in each category. We show in Figure 4.1c, the resulting decision trees.

By putting rules with the *large* label on different fields in separate decision trees rules, EffiCuts untangles existing "orthogonal structures" and remove completely the induced rule duplication. This results in a dramatic reduction of the memory size compared to HiCuts and HyperCuts. However, one need to traverse all trees in order to find the most specific match, resulting in a large number of memory accesses and this reduces significantly the throughput [77].

4.2.4 Discussions

The above description of different packet classifications gives insight for understanding classification performance issues. Using the geometrical view, we observed the major impact of "orthogonal structures" and the non-uniformity of range sizes on memory footprint and on the performance. A noteworthy case happens when a ruleset contains only *small* ranges in at least one of its dimension, like the ruleset in Table 4.3. For such cases one can separate all the rules, using a decision tree working only on the dimension with only *small* ranges, as the cuts/splits on this dimension will not intersect any "orthogonal structures" happening in other dimensions. In this case, using EffiCuts that would generate two trees for the two categories $(small, small, large)$ and $(small, large, small)$, will be inefficient. The above observations, and the fact that all in all the main issue is to be able to separate subregions with a small number of memory accesses, drive us to propose these guidelines:

Field 1	Field 2	Field 3
00*	*	01
01*	01	*
10*	*	10
11*	10	*

Table 4.3: Ruleset with a lot of distinct *small* ranges on Field 1

1. "Orthogonal structures" should be considered, and rules should be eventually splitted in order to untangle these structures and avoid memory explosion.

2. When splitting a ruleset, if a dimension appears that contains only *small* ranges, it should be used to separate the rules with a single tree.

3. Equal-sized cutting becomes more efficient when ruleset ranges are uniform, if not splitting with non-equal sized intervals should be considered.

Indeed, these obvious observations, cannot be used by a network operator if the structure of the ruleset is not analyzed. We therefore propose and evaluate methods and algorithms that analyze rulesets in order to extract metrics that will help in deciding the best packet classifier for a given ruleset.

4.3 Memory footprint estimation

Given a ruleset, the first concern is whether the size of built DT can fit in the available memory (CPU cache). As we saw in Section 4.2.2, orthogonal structures within the ruleset are a major cause of large memory requirements. We have therefore to characterize these orthogonal structures in order to estimate the DT memory footprint. The goal here is not derive a precise estimation of the memory footprint, it is to use rulesets features in order to achieve a rough estimate which gives an order of magnitude of the size.

We will adopt the ruleset portioning into 2^K categories in $\{small, large\}^K$ described previously in EffiCuts [77]. As in practice, $50\% \sim 90\%$ of the cuts or splits are performed on the IP source and destination fields [55], we will first concentrate on these two dimensions and ignore others, without losing much in estimation accuracy. We therefore analyze orthogonal structures involving only the IP source and destination fields, and label rules as $(small, small)$, $(large, small)$, $(small, large)$ or $(large, large)$ based on these fields. The number of rules in each category is denoted respectively as ss, ls, sl, and ll.

To simplify, for the time being, we will assume that large range rules cover the whole span of the associated dimension, *i.e.*, the corresponding IP address range is a wildcard. This will result in overestimation of the memory footprint which we will address in the next section. We also denote the number of distinct ranges on the source and destination IP fields as us and ud. These two values can be calculated by a simple scan of the ruleset. Let $\alpha = \frac{us}{us+ud}$ be the proportion of distinct source IP ranges.

The $(small, small)$ rules can be easily separated by either using source or destination IP ranges. We assume that they are separated by source or destination IP field without duplication and in proportion to α and $1 - \alpha$. The memory needed to separating these $(small, small)$ rules is therefore $M_{ss} = ((1 - \alpha) \times ss + \alpha \times ss) \times PTR = ss \times PTR$, where PTR is the size of a pointer (pointing to a rule).

Orthogonal structures are created by $(small, large)$ and $(large, small)$ rules. When isolating the small range side of any of these rules (i.e. when cutting in the direction of the dimension of their large range), all large ranges along the other dimension are cut, resulting in the need to duplicate the corresponding rules on either side of the cut. For instance, all the cuts (or splits) on source IP field, to separate every $(small, large)$ rules, will duplicate all $(large, small)$ rules, generating

ls duplicated $(large, small)$ rules, and similarly for each $(large, small)$ rule, there will be sl duplicated $(small, large)$ rules.

Furthermore, the $ss \times \alpha$ rules labelled $(small, small)$ that have been separated using the source IP ranges, will also duplicate each $(large, small)$ rule, and similarly the $ss \times (1 - \alpha)$ rules labelled $(small, small)$, separated using the destination IP ranges, will duplicate each $(small, large)$ rule.

Overall, the upper bound on the number of duplication of ($large$, $small$) rules is thus $ls \times (sl + ss \times \alpha)$, while that for the duplication of $(small, large)$ rules is $sl \times (ls + ss \times (1 - \alpha))$. However, in practice DT algorithms stop building the DT when there is at most a given threshold number, $binth$, of rules in any leave. This means that the number of duplicates are over-estimated by a factor of $\frac{binth}{2}$ (2 rules per leaves .vs. $binth$ rules per leaves) yielding:

$$M_{ls} = ls \times \frac{sl + ss \times \alpha}{binth/2} \times PTR \tag{4.1}$$

$$M_{sl} = sl \times \frac{ls + ss \times (1 - \alpha)}{binth/2} \times PTR \tag{4.2}$$

The last category of rules, the $(large, large)$ one, will get duplicated either by splitting of cutting on source or destination IP fields. The $(large, large)$ rules need therefore a memory size:

$$M_{ll} = ll \times \frac{sl + ss \times \alpha}{binth/2} \times \frac{ls + ss \times (1 - \alpha)}{binth/2} \times PTR \tag{4.3}$$

The total memory size is finally estimated as the sum of the four elements: $M = M_{ss} + M_{ls} + M_{sl} + M_{ll}$.

4.3.1 Improving memory size estimation

Our memory size model is based on two assumptions: 1) all cuts/splits on source IP fields will definitely cause the duplication of $(large, small)$ rules, while cuts/splits on destination IP fields will definitely cause the duplication of $(small, large)$ rules. 2) Cuts or splits in one decision tree are performed *only* on IP fields. All these assumptions will lead to the over-estimation of the real memory size.

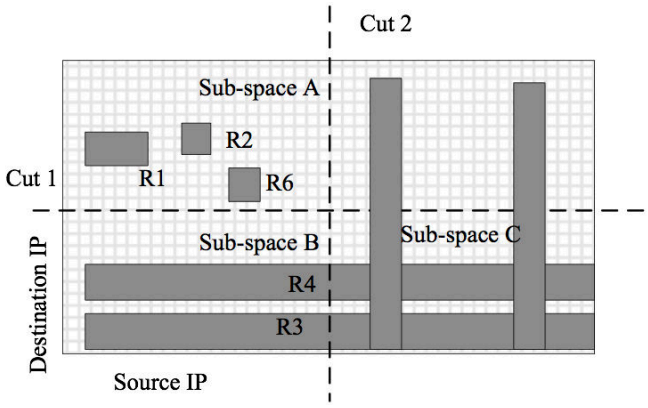


Figure 4.3: The distribution of $(small, small)$ rules is skewed

First, the assumption that all orthogonal rules are duplicated over-estimates the memory requirement, as some large ranges might not cover the full range and therefore might not be duplicated in all cases. Second, splitting $(small, small)$ rules does not always lead to the duplication of either $(small, large)$ or $(large, small)$ rules. As shown in Figure 4.3, $(small, small)$ rules are not uniformly distributed. After the Cut 1 and Cut 2, cuts or splits in the subspace A, on either source and destination IP fields will split $(small, small)$ rules, however not cause any duplication of $(small, large)$ or $(large, small)$ rules.

We can improve the memory estimation by partitioning the feature space into smaller subspace. The key insight is that, in a smaller space, the first assumption is more likely to hold as the *large* ranges are more likely the "full range" in the smaller subspace.

So in order to reduce the over-estimation and improve the quality of the memory footprint estimation, we first divide the large feature space into n equal-sized rectangular sub-space, and apply the memory estimation model to each one of these subs-space separately. We will illustrate this with the ruleset example in Figure 4.4. In the initial memory estimate, R1 and R4 are considered as $(large, small)$ rules, and Cut 2 is supposed to cause duplication of R1 and R4. However, as the R1 and R4 are not wide enough, they are not duplicated by Cut 2. After dividing the space into sub-space, we can witness that any cut on the source IP field in sub-space A (*.resp.* C) will surely cause the duplication of R1 and R4, but not in subspace B. This therefore improves the memory footprint estimation.

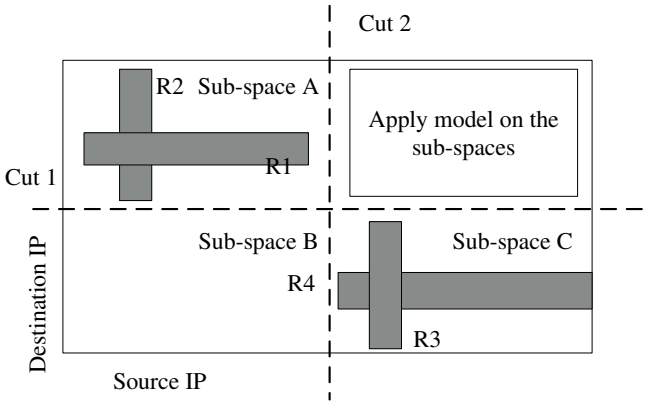


Figure 4.4: Improved Memory Size model

It is noteworthy that in the process of dividing the space into sub-spaces, some $(large, large)$ rules may become fully covered by more specific and higher priority rules in this sub-space. These redundant rules must be removed before calculating parameters ll, ls, sl and ss of the the orthogonal structure in the subspace.

4.3.2 The bound of memory consumption

We now give a proof to show that when 1) there is no $(large, large)$ and $(small, small)$ rules 2) cuts or splits are only allowed to be performed on IP fields, our memory

consumption model gives actually the lower bound of the memory consumption of a ruleset.

Proof: For separating $(large, small)$ rules into X rules per leaves, one need at least $\frac{ls}{X}$ splits, and all the $(small, large)$ rules get duplicated. we have:

$$M_{sl} = sl \times \frac{ls}{X} \times PTR \tag{4.4}$$

$$\tag{4.5}$$

And similarly, for separating $(small, large)$ rules into Y rules per leaves, we need at least $\frac{sl}{Y}$ splits, and all the $(large, small$ rules get duplicated. We therefore have:

$$M_{ls} = ls \times \frac{sl}{Y} \times PTR \tag{4.6}$$

$$\tag{4.7}$$

The total memory should be:

$$M_{ls} + M_{sl} = (sl \times \frac{ls}{X} + ls \times \frac{sl}{Y}) \times PTR = ls \times sl \times \frac{X+Y}{XY} \times PTR \tag{4.8}$$

In each leaf, we have $X + Y$ rules, and $X + Y = binth$. According to

$$XY \leq (\frac{X+Y}{2})^2 = (\frac{binth}{2})^2 \tag{4.9}$$

we have:

$$M_{ls} + M_{sl} = ls \times sl \times \frac{X+Y}{XY} \times PTR \geq \frac{4 \times ls \times sl}{binth} \times PTR \qquad (4.10)$$

Since there are no $(small, small)$ and $(large, large)$ rules, M_{ll} and M_{ss} should be 0. The Formulation 4.10 and 4.1 are actually equal, therefore that our memory model estimates the lower bound in such special case.

Since the $(small, small)$ rules do not duplicate a lot, usually M_{ss} is small, so when there are few $(large, large)$ rules, the memory size estimation should be closed to the actual memory size. We illustrate the rule splitting process in the proof in Figure 4.5.

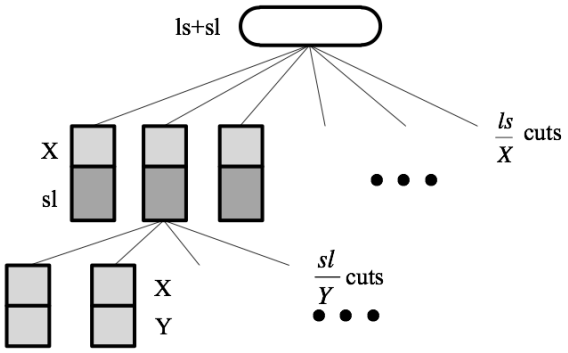


Figure 4.5: The rule splitting process in the proof

4.3.3 Limitations

The assumption that all the splits are performed *only* on IP fields is also a source of the memory size over-estimation, as splitting or cutting on other dimension can reduce the impact of orthogonal structure (see Section 4.2.4).

However the main aim of the calculation in this section is to obtain a rough estimate giving an order of magnitude of the memory footprint. We will show in Section 4.7 that software based packet classification performances are not sensitive to the precise memory size but roughly to its order of magnitude. Our memory footprint

estimation can therefore be used as a fast memory size checker, especially for large rulesets. We will also show a detailed analysis of the error of the memory estimation in Section 4.7.

The last limitation of the model is that we assume that we can separate N rules with N cuts/splits. While this is usually correct for splits, this can be incorrect for cuts due to the inefficiency of equal-sized cutting over non-uniform rules. We expect therefore better estimates for HyperSplit than HiCuts/HyperCuts.

4.4 Characterizing range distribution uniformity

As explained in the previous section the uniformity for small range distribution (we call it coverage uniformity for short) is an important factor for deciding to apply cuts or splits when building the decision tree. We show in Table 4.4 the number of unique small ranges in large rulesets and observe that, the number of unique small ranges on IP fields is usually comparable to the total number of rules. Therefore, the rulesets can be separated *only* by the small ranges on IP fields and the uniformity of small ranges on IP fields is important for choosing cut or split. In the forthcoming, we will propose a simple variant of a centered interval tree [12] and characterize the coverage uniformity by computing shape metrics on such trees.

4.4.1 Interval tree

A centered interval tree [12] is a well-known tree used to efficiently represent intervals or ranges (in the context of packet classification). Each node of the interval tree is defined by a center point which is used to separate ranges: The ranges completely to the left of the center point (left ranges for short), those completely to the right of the center point (right ranges), and those containing the center point. The latter are then associated with the node itself (and removed from further consideration). A

Ruleset	unique src. IP small range	unique dst. IP small range	#src/rules(%)	#dst/rules
acl1_10K	4023	750	41%	7%
acl2_10K	6069	6527	64%	69%
acl3_10K	1017	1110	10%	11%
acl4_10K	918	1864	10%	19%
acl5_10K	371	1527	5%	21%
fw1_10K	3389	6665	36%	70%
fw2_10K	8309	3080	86%	32%
fw3_10K	2835	6209	31%	69%
fw4_10K	3884	6797	44%	76%
fw5_10K	3414	5327	39%	60%
ipc1_10K	1332	2768	14%	29%
ipc2_10K	4748	8923	47%	89%
acl1_100K	99053	236	99%	0.2%
acl2_100K	8315	8092	11%	11%
acl3_100K	85355	86603	86%	87%
acl4_100K	88434	32766	89%	33%
acl5_100K	43089	78952	43%	80%
fw1_100K	26976	66173	30%	74%
fw2_100K	81565	30602	85%	32%
fw3_100K	15960	62993	19%	75%
fw4_100K	38076	67073	45%	80%
fw5_100K	29786	54004	35%	64%
ipc1_100K	86210	90433	87%	91%
ipc2_100K	47228	89135	47%	89%

Table 4.4: the number of unique IP small ranges in large rulesets

left sub-tree is then built using the left ranges and a right sub-tree is built using the right ranges. This procedure is repeated until all ranges have been associated with nodes in the tree. Figure 4.6 shows the interval tree built on the ranges on the F1 field. Each node N in the interval tree contains the following information:

1. The split value. In the figure, we use intervals ([12, 15], [8, 9] *etc.*) to represent the split value.

2. The ranges set recording all the ranges, denote as $N.L(r)$. All the ranges in the set include the split value.

3. The total number of rules consisting of any range on the specific field in the range set, denoted as $N.numRules$.

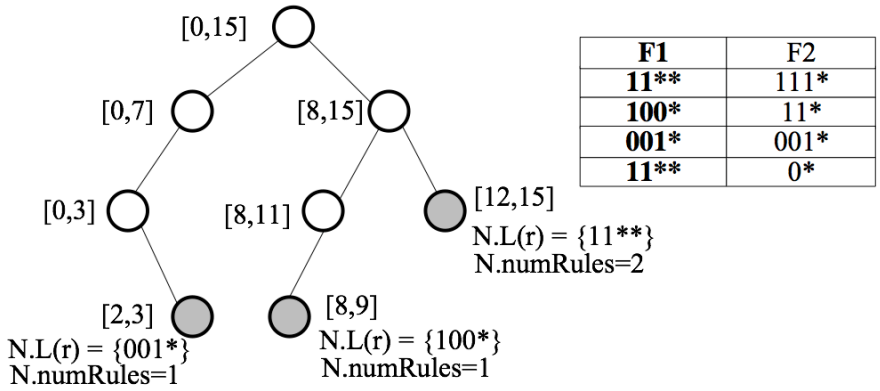


Figure 4.6: The interval tree data structure

While the original centered interval tree algorithm picks center points to keep the tree as balanced as possible, we use a slightly different strategy to build a tree whose shape will reflect the degree of uniformity in the ranges. We start with the full range, its widest possible span, for the field under consideration and pick as centre point for the root node the middle of this range. We then use the left (resp. right) half range for the left (resp. right) child. Note that with this approach, the centre point in a node depends solely on the original full range and the position of the node in the tree. As in practice DT algorithms stop cuttings/splittings on nodes associated with less then *binth* rules, we will stop the growth of our interval tree when the number of rules associated with a node containing less than *binth* rules.

In the interval tree, the large ranges are likely to be "absorbed" by the nodes near to the root, while the small ranges are usually associated with leaf nodes. So the shape of interval trees actually represents the distribution of small ranges. The main insight into our method is that centre points correspond to equal-sized cuts of the original full range. And since a branch of the tree only growth if there are ranges on either side of the corresponding centre point, a balanced tree would indicate uniform coverage of

ranges. In such a case, an algorithm using equal-sized cuts (e.g. HiCuts/HyperCuts) would very efficiently separate the ranges and these associated rules and produce a very fast classifier.

In fact, each node at the kth level of the tree, with the root being the level 0, covers a portion $\frac{1}{2^k}$ of the full range. These range portions can be efficiently represented by 2^k equal-sized cuts on the full range. Assume a node N resides in the kth level of the interval tree, rules intersecting with the range portion managed by N can be found by collecting associated rules in the path from the root to N. These intersected rules will be duplicated when performing $2^l, l > k$ equal-sized cuts on the full range. Since rules in nodes at the same level of the tree are non-overlapping, a node is missing in this tree means that there is no rules on that side of the parent node, in which case, performing any cut in this interval would be useless (separate no rules but duplicate the intersected rules). This means that the interval tree structure gives interesting insights into the efficiency of using cuts. When the interval tree is balanced, or as will be explained later quasi-balanced, it is meaningful to use cuts and there will be not any, or better said not too many, spurious cuts. If the interval tree is un-balanced, using splits will avoid these spurious cuts resulting in smaller duplicates.

However, a perfectly balanced interval tree may be too strict a condition to pick equal-sized cutting. We therefore define *quasi-balanced tree* as a tree where the following condition is verified at each level of the tree:

$$\frac{\#Nodes\ in\ the\ k^{th}\ level}{\#Nodes\ in\ the\ (k-1)^{th}\ level} \geq B_{ratio} \tag{4.11}$$

As our interval tree is a binary tree, $B_{ratio} \in (0, 2]$. We will set $B_{ratio} = 1.5$ for a good approximation of balance for the tree. Note that since we set $B_{ratio} > 1$, a quasi-balanced tree contains at least 3 nodes, and the height of one quasi-balanced tree is at least 2. This is the reason why chains of isolated nodes do not belong to any quasi-balanced subtrees as in Figure 4.7.

4.4.2 Characterizing the shape of interval trees

In practice, interval trees built from rulesets are unbalanced, containing nodes with single child or even leaves at various levels in the tree. These nodes break the overall tree into several quasi-balanced subtrees (triangles) of different sizes (see Figure 4.7). In order to characterize these quasi-balanced subtrees, we define two for each node metrics: the *balanced depth* BD, the height of the quasi-balanced subtree the node belongs to, and *balance tree distance*, D, the number of quasi-balanced sub-trees between a given sub-tree and the top one.

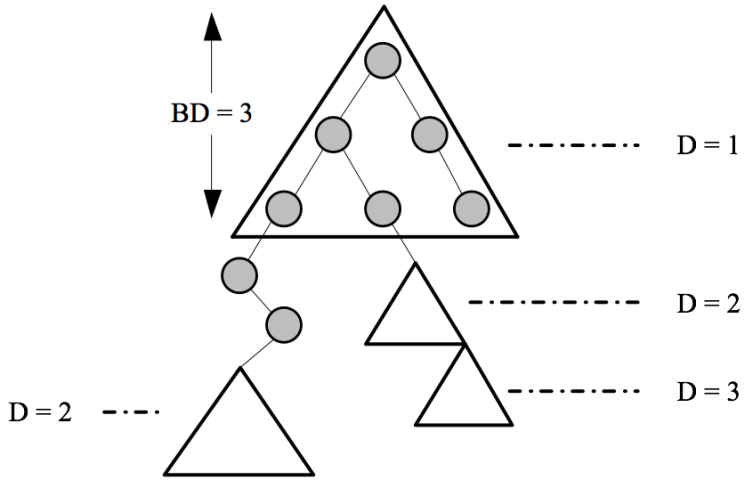


Figure 4.7: Balanced Tree Distance and Balanced Tree Depth

The full interval tree is characterized by D_{max}, the maximum value of *balance tree distance*, and BD_{max}, the maximum *balance depth*, calculated over all quasi-balanced subtrees. When the range coverage is non-uniform, the interval tree contains many quasi-balanced sub-trees with small height values, and its D_{max} will be large. On the other hand, a small D_{max} value means a more uniform coverage.

We show the complete measurement algorithm in Figure 4.8.

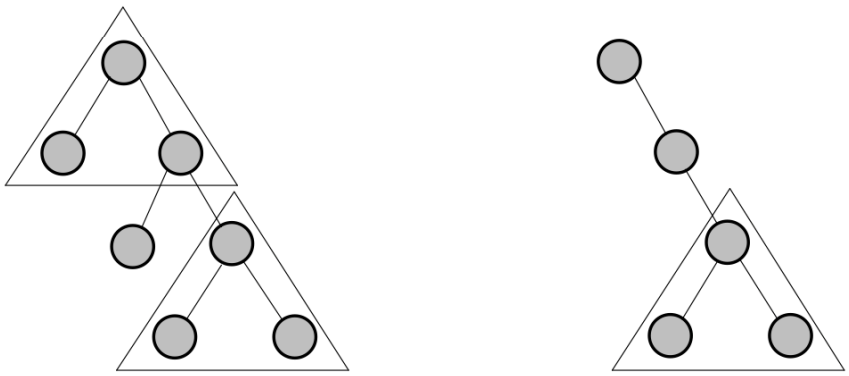
In Algorithm 4.8, the `getBalanceDepth(root)` computes the balance depth of the quasi-balanced tree root at `root`, the `getBTreeLeaves(root)` return a node set consisting of the child nodes of the leaf node of the quasi-balanced tree root at `root`.

Algorithm 1 Measure (root, *binth*, *step*)

```
global BD_max = 0;
global D_max = 0;
bd = getBalanceDepth(root);
{leaf} = getBTreeLeaves(root);
if bd > BD_max then
  BD_max = bd;
end if
if step > D_max then
  D_max = step;
end if
for leafNode ∈ {leaf} do
  if ChildrenCount(leafNode) == 1 then
    while ChildrenCount(leafNode) == 1 do
      leafNode ← getChildren(leafNode);
    end while
  end if
  if leafNode.numRule < binth or ChildrenCount(leafNode) == 0 then
    continue;
  end if
  Measure (leafNode, binth, step + 1);
end for
```

Figure 4.8: Measuring algorithm

Recall that the quasi-balanced tree contains at least 3 nodes. Here we give two measurement results shown in Figure 4.9 to illustrate the limitation.



(a) $BD_{max} = 2, D_{max} = 2$ (b) $BD_{max} = 2, D_{max} = 1$

Figure 4.9: the measurement results of two interval trees

4.4.3 Algorithm decision framework

In practice, we observed that small rulesets usually exhibits a non-uniform distribution of small ranges (non-uniform coverage), and therefore HyperSplit is suited to them. However, as the size of rulesets grows, the size of orthogonal structure, as well as the number of uniformly distributed ranges also grows. When our memory footprint model indicates that the size of the built DT is too large, one needs to split the ruleset into sub-rulesets and build a single DT for each set. However, due to the probable existence of the "coverage uniformity" in some of the subsets, rather than using HyperSplit algorithm on all the sub-rulesets, it is well worth checking whether one sub-ruleset is uniform enough to warrant an attempt to use the faster classifier (use HiCuts/HyperCuts algorithm) on each sub-ruleset or not.

Now that we have a metric for characterizing range coverage uniformity we can use this metric to decide if cut based algorithms should be used or split based one. Let us denote the height of an interval as H and $D_{\max}$ the maximum number of quasi-balanced trees from top to bottom.

If the height of each of the quasi-balanced tree is $h_1, h_2, \ldots, h_n$ we have therefore

$$\underbrace{h_1 + h_2 + \cdots + h_n}_{D_{max}} = \overline{h} \times D_{max} \leq H \tag{4.12}$$

where $\overline{h}$ is the average height of quasi-balanced trees. As quasi-balanced tree has at least a height of 2, we will have $\overline{h} \geq 2$, so that:

$$2 \times D_{max} \leq \overline{h} \times D_{max} \leq H \tag{4.13}$$

For matching a set of K non-overlapping small rules we need at best a binary decision tree of height at least $log_2 K$. When using the interval tree, all rules in leaves

are non overlapping and the overlapping rules are absorbed by rules in higher levels. As explained before we stop the growth of interval tree when there are *binth* rules in a node. Therefore the height of a balanced interval tree should be close to its lower bound that $\log_2(\frac{\#(non-overlapping\ rules)}{binth})$. On other hand if one wants make a partition of all rules using splits he will need a decision tree of height at least $\log_2 \frac{\#rules}{binth}$, so there is an interest in using a cut-based algorithm only if $H < \log_2 \frac{\#rules}{binth}$. This means that when an interval tree height is between $\log_2(\frac{\#(non-overlapping\ rules)}{binth}) \leq H < \log_2(\frac{\#rules}{binth})$, there is a benefit in term of tree height or equivalently memory access in using cut. The higher bound can be rewritten as $D_{\max} < \frac{1}{2}\log_2 \frac{\#rules}{binth}$. We will use this last criterion to decide to implement a DT with cut or with splits. Indeed, the closer is the tree height from its lower bound the more balanced will be the interval tree.

4.4.4 SmartSplit algorithm

Now we can describe the SmartSplit algorithm that builds a multiple DT similar to EffiCuts. We first categorize the rules in the ruleset into *small* and *large* based on source and destination IPs. We put aside $(large, large)$ rules and build a specific tree for them that will use HyperSplit as these rules should be separated by port fields that have generally non-uniform coverage.

Since $(small, large)$, resp. $(large, small)$, rules are mainly separated by source IP field, resp. by destination IP field, we build the interval tree for both source and destinationIP fields, and we calculate D_{max} for both trees. We merge the set of $(small, small)$ rules with the $(small, large)$ when $D_{max}(srcIP) \leq D_{max}(dstIP)$, and with $(large, small)$ rules when $D_{max}(dstIP) < D_{max}(srcIP)$. This results in two sub-rulesets, S1 containing $(small, large)$ and S2 containing $(large, small)$ rules. One of S1 or S2 will also contains $(small, small)$ rules.

Now, we build for each one S1 and S2 a separate DT that will disentangle orthogonal structures. For the sub-ruleset containing only small ranges on source IP *.resp.* destination IP field, we use $D_{max}(srcIP)$ *.resp.* $D_{max}(dstIP)$ for algorithm recommendation using the criterion we had $D_{\max} < \frac{1}{2}\log_2 \frac{\#rules}{binth}$.

The SmartSplit algorithm is different from the EffiCuts algorithm from two perspectives. First, the SmartSplit algorithm only considers the "orthogonal structure" on IP fields, and separates a ruleset into 3 sub-rulesets, while EffiCuts considers the existence of "orthogonal structure" on both IP and port fields, resulting in $5 \sim 9$ sub-rulesets. Large number of sub-rulesets results in a large number of memory access and therefore lower classification throughput. Second, SmartSplit algorithm tries to maximize the classification speed by using different algorithms on different sub-rulesets, while EffiCuts uses only a variant of HyperCuts on all the sub-rulesets.

Besides the above points, we applied a pruning trick in our implementation of SmartSplit. As we have multiple trees, each should be sequentially tested in order to find the most specific rules. However we store for each node in the decision tree the index of the rule with minimal priority rule among all rules managed by the node. After doing the search on the first tree we use the matched rule number resulting from this first search and compare it to the minimal priority rule index stored at the node and we pursue the search if and only if the index of minimal priority rule is less than the already matched rule index. If not we prune the search for the whole decision tree. As we observed that generally rules in the $(small, small)$ set are more specific than rules in the $(small, large)$ and the $(large, small)$ set, that are more specific than $(large, large)$ rules, we first check the decision tree containing the $(small, small)$ rules, and we continue by the remaining $(small, large)$ or $(large, small)$ tree and we finish with the $(large, large)$ DT. This pruning optimization reduces the unnecessary memory access in multiple decision trees, improving the look up performance significantly.

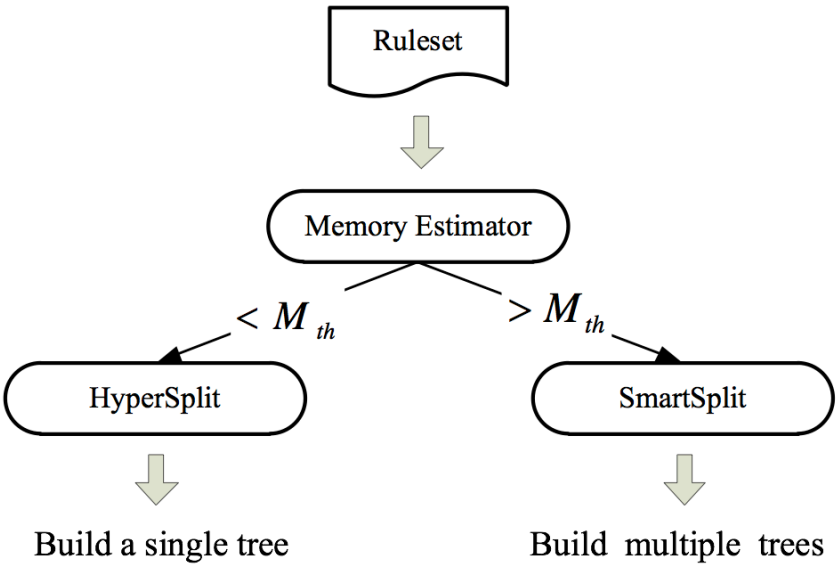


Figure 4.10: The AutoPC framework

4.5 The AutoPC framework

Combining all the algorithms described above, we propose *AutoPC*, a framework for autonomic construction of decision trees for packet classification. For a given ruleset, *AutoPC* first estimates the memory size requirements. If the estimate is less than a pre-defined threshold M_{th}, a single tree will be built using HyperSplit algorithm. Otherwise, the ruleset will be processed with the *SmartSplit* algorithm. The complete procedure of *AutoPC* is illustrated in Figure 4.10.

4.6 Experimental Methodology

In this section we will validate the analysis presented before. For this purpose we have implemented HiCuts, HyperSplit and EffiCuts algorithms in our experiments. In each node of HiCuts tree, we have used a pointer array instead of a bitmap to index child nodes, allowing more cuts per node (at most 65536 cuts in our implementation). However, in this case, each node needs 2 memory accesses (one for index array and

Table 4.5: Node data structure size in bytes

HiCuts	**1**	header information (the dimension to cut, leaf or internal node flag *etc.*)
	6	boundary information, **4** bytes are used to store the *min* value of the boundary of one dimension. **2** bytes are used to store the number of cuts.
	1	1 byte is used for storing the bit shift value.
	4	pointer to the children pointer array.
HyperSplit	**8**	4 bytes for the split point. Other bytes for the header information.

one for node). Our HiCuts implementation enables `Range Compaction` and `Node Merging` optimization however it disables the `Rule Move Up` for node size efficiency [22].

For HyperSplit algorithm, the code from [69] is used. To note, when calculating the memory size, the original source code does not account for the memory of rule pointers, we add this part of memory for a fair comparison. Each node of HyperSplit needs only one memory access.

For EffiCuts algorithm, we obtained the implementation from its authors and enable all its optimization techniques. The *spfac* of EffiCuts is set to 8 while the *spfac* of HiCuts is set to 4. The *binth* number is set to 16 for HiCuts, HyperSplit and EffiCuts .

For SmartSplit algorithm, we found that the number of $(large, large)$ rules are usually small compared to the size of the original ruleset, we therefore use $binth = 8$ for the HyperSplit tree built over the $(large, large)$ rules.

For all algorithms, we have stored each rule using 18 bytes [55]. Each rule needs one memory access. Note that EffiCuts has its own way of calculating the number of memory access (in their code, each rule needs less than one memory accesses). For a fair comparison, we use the results directly from the code of EffiCuts.

Table 4.5 shows the data structure of each node for HiCuts and HyperSplit. The header size of one node in HiCuts is 12 bytes while each node of HyperSplit needs only 8 bytes. The pointer size in all the algorithms is 4 bytes.

We use ClassBench [63] to generate synthetic rulesets. In our experiment, we have used all available types of rules including Accesses Control List (ACL), Firewall (FW) and IP Chain (IPC). For each type, we have generated rulesets containing from 1K to 100K rules.

Our experiments include performance comparison on both memory size and memory accesses observed from the built decision tree as well as real evaluation of classification speed on a commodity server. The speed is measured through averaging the lookup latency over a low locality traffic generated by ClassBench; Each trace contains 1 millions of 5 tuples. All experiments are run on Ubuntu machines, with 8 cores, Intel i7 processors, 4MB L3 Cache and 24GB of DRAM.

4.7 Experiment Results

4.7.1 Memory Size and Real Performance

In order to explore the relationship between the memory size and the real performance of packet classification on software based platform, we run the HyperSplit algorithm on 25 example rulesets, with memory footprint ranging from less than 10K to larger than 700MB. We measure the cache miss rate and the average memory access latency of the HyperSplit matching process on our experimental platform and we show in Figure 4.11 the relationship of memory size and memory access latency, and in Figure 4.12 the relationship of memory size and cache miss rate.

As can be seen in Figure 4.11, the memory access latency increases slowly with memory size varying from 10KB to 1MB. When the memory size becomes larger than 10MBytes, the latency explodes. The increasing memory access latency can be

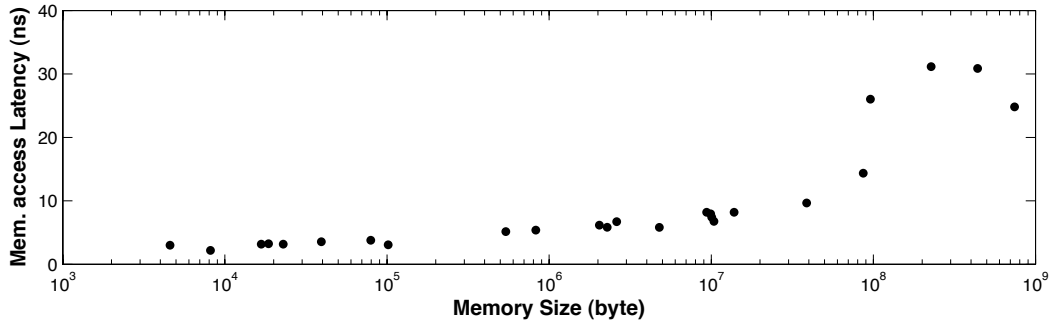


Figure 4.11: Average Memory Access Latency and Memory Size

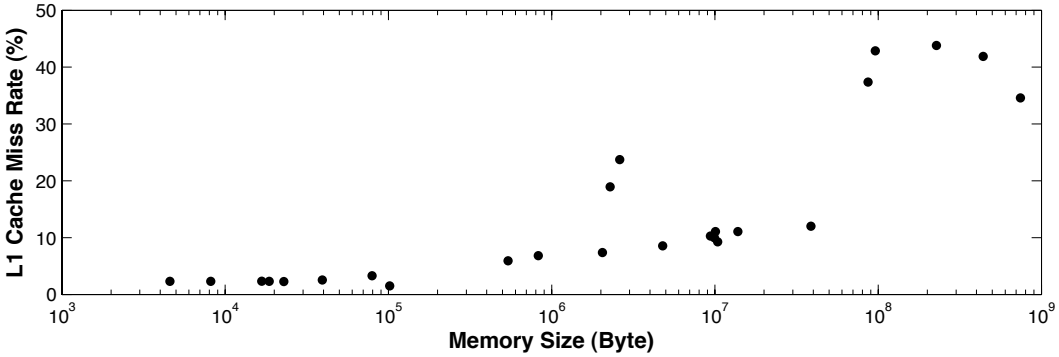


Figure 4.12: Cache Misses Rate and Memory size

explained by the fact that the memory footprint of the DT prohibits it to fit into the processor cache memory of our platform (4MB of L3 cache). As shown in Figure 4.12, the cache miss rate stays below 10% when the memory size is less than 10^7 Bytes, and it increases significantly to around 50% when the memory size goes beyond 10^8 Bytes. Based on this observation we set the memory threshold M_{th} in the AutoPC framework to 10MBytes to consider splitting rulesets when estimated memory size is larger than 10MB.

4.7.2 Memory estimation under different number of partitions

We present our estimation on large rulesets using different n in Figure 4.19 - 4.18. We see that, for ACL and IPC rules, increasing n will significantly reduce the memory estimate. In contrast, for FW rules, the estimation does not change too much for increasing n. This confirms that by dividing the space into n subspace, the memory model will overcome the over-estimating introduced by non-uniformly distribution of IP ranges.

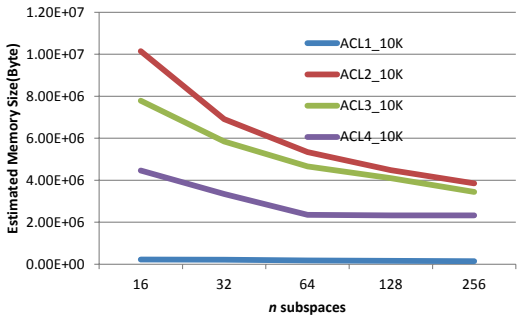


Figure 4.13: acl10k

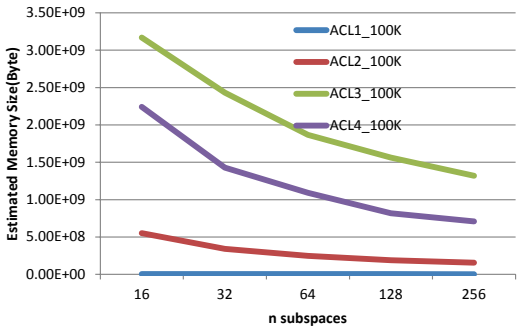


Figure 4.14: acl100k

4.7.3 Estimated and Actual Memory

We apply our memory consumption model on 60 rulesets of various size consisting of 1K, 5K, 10K, 20K and 50K rules. In the experiments, we first divide the source-destination IP space into 256 equal-sized rectangular sub-space, and perform memory size estimation in each sub-space to obtain a better estimate. We also set the *binth* to different values (16 and 8) to evaluate its impact on the memory size estimation. We present the estimated and observed memory footprint for $binth = 16$ in Figure **??** and for $binth = 8$ in Figure **??**.

Both Figure **??** and Figure **??**, show that the estimated and observed memory size remain aligned around a perfect prediction line in logarithmic scale, meaning

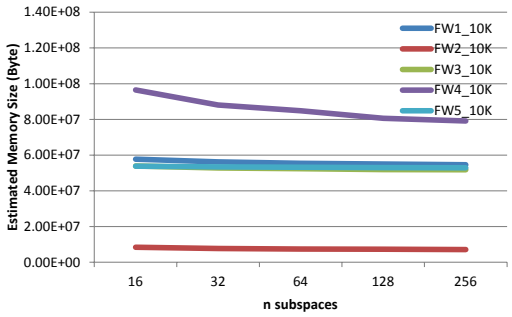


Figure 4.15: fw10k

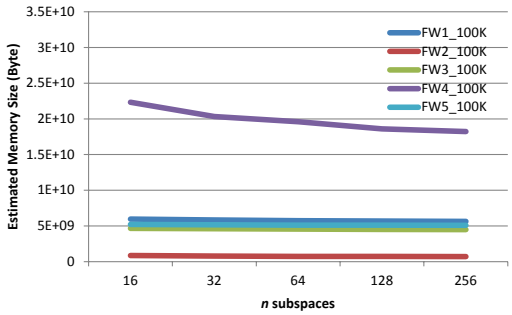


Figure 4.16: fw100k

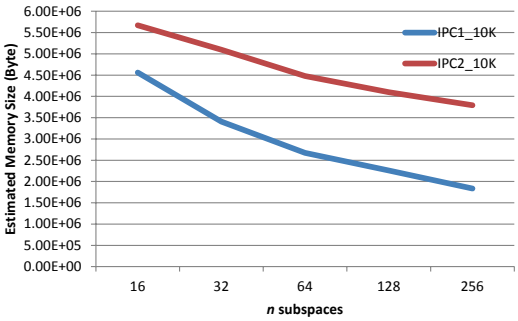


Figure 4.17: ipc10k

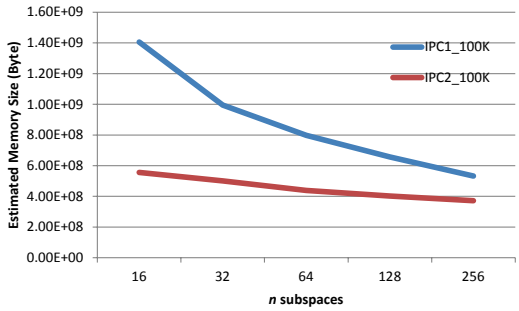


Figure 4.18: ipc100k

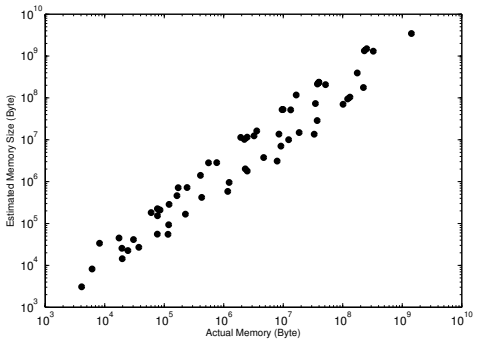


Figure 4.19: Estimated and Actual memory size with $binth = 16$

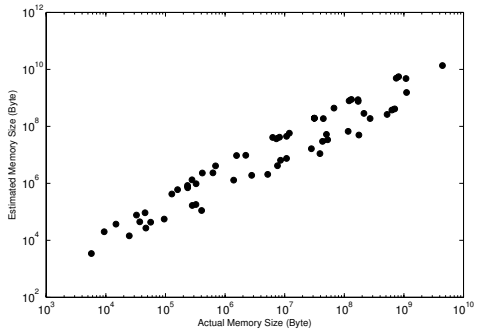


Figure 4.20: Estimated and Actual memory size with $binth = 8$

that the order of magnitude of the estimated memory is correct. As mentioned before, the memory access latency increases with the order of magnitude of memory size increases. Therefore, our memory consumption can be used to predict better the classification performance of a ruleset than using the number of memory access.

We show in Figure 4.21 the estimated and actual number of rulesets within the special memory size interval. We see that our consumption model is capable of identifying the rulesets into the right categories with small errors. In our experiment, the average memory size estimate error (mean($\frac{est}{actual}$)) with $binth = 16$ is 2.57, and with $binth = 8$ the error is 2.79.

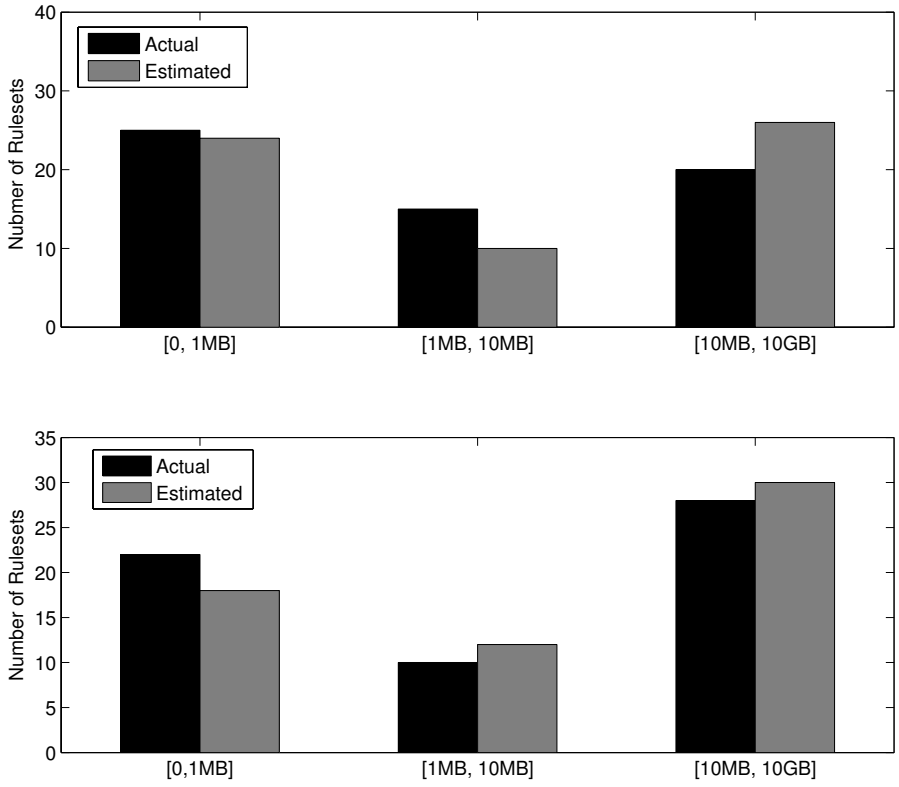


Figure 4.21: The estimated and actual number of rulesets for $binth = 16$(top) and $binth = 8$(bottom)

We present the estimate and actual memory size of all the 100K rulesets with $binth = 16$ in Table 4.6. The memory size varies from less than 1MBytes to several GBytes, and the time for building a HyperSplit trees varies from tens of minutes to several hours. In practice, HyperSplit algorithm has the smaller building time than

Ruleset	HyperSplit Mem	HyperSplit Time(s)	Estimate Mem	Estimate Time(s)
ac1_100K	834K	167	2.8M	0.4
acl2_100K	95M	234	150M	0.6
acl3_100K	250M	1794	1.2G	0.7
acl4_100K	104M	1061	674M	0.6
acl5_100K	498K	186	384K	0.4
ipc1_100K	836M	2424	508M	0.6
ipc2_100K	463M	1132	354M	0.6
fw1_100K	929M	2124	5.3G	1.7
fw2_100K	970M	2568	680M	0.8
fw3_100K	733M	1148	4.2G	1.9
fw4_100K	6.5G	6413	17.3G	10
fw5_100K	1.2G	1891	4.9G	2

Table 4.6: Estimated and Actual Memory size of Large rulesets

HiCuts and EffiCuts, *e.g.*, the HyperCuts code usually takes $1 \sim 2$ hours while the EffiCuts code usually takes $5 \sim 9$ hours to build a Decision tree.

Table 4.6 shows that our memory consumption model is able to detect in less than one second that the large ruleset (acl1_100K and acl5_100K) which has small memory footprint avoiding the application of SmartSplit and enabling fast classification with small memory size and few memory accesses.

4.7.4 Study the error of the memory estimation

We now present more experiments here to illustrate that the reason why our memory size model will over-estimate the memory size. Our experiments show that there are two reasons which results in the memory size over-estimation:

1. The cutting/splitting on port/protocol fields. Our model assume that all the cuts/splits are performed on IP field. However, the actual splits on IP fields is actually around 60%. This assumption leads to an over-estimate on the number of cuts required to separate the $(large, small)$ and $(small, large)$ rules. Because

splits on other fields, can also be viewed as a way to untangle these "orthogonal" rules, so the final memory size will be smaller than our estimate.

2. The non-uniformly distribution of IP ranges. For this, we proposed an improved memory size model and we illustrated the effectiveness of this improvement in Section 4.3 and Section 4.7.2.

We have already discussed the non-uniformity issues in Section 4.3.1. Here we will show that the cutting/splitting on the port/protocol fields is another main source of the memory size over-estimating.

We first present the splits distribution of each fields on *uniform* rulesets using HyperSplits. The parameter n is set to 16 for memory size estimation.

Ruleset	IP	non-IP	Actual	estimate
FW1_10K	60%	40%	9.6M	55M
*FW2_10K	94%	6%	9.1M	8.08M
FW3_10K	56%	44%	9.8M	51M
FW4_10K	59%	41%	38M	92.3M
FW5_10K	63%	37%	13M	51M
IPC2_10K	98%	2%	4.6M	5.41M
IPC1_100K	70%	30%	724M	1.3G
*IPC2_100K	99%	1%	427M	530M
*FW2_100K	95%	5%	970M	805M
FW4_100K	58%	42%	5.6G	21.3G

Table 4.7: split distribution of each fields

From Table 4.7, we can see that for all the ruleset except fw2_10K, ipc2_100K and fw2_100K, the number of splits on port are usually around half the total splits, and this causes at most 4 times (See the equation 4.3. Since the number of cuts on both source and destination IP field are over-estimated by 2 times, the memory over-estimation is at most 4 times) over-estimation. On fw2_10K and ipc2_10K(also ipc2_100K), since the splits on IP fields is nearly 100%, our estimation actually gives very accurate estimation (the error is around 20%).

Ruleset	Actual	estimate	error
FW1_10K	12M	8.69M	-25%
*FW2_10K	8.2M	6.60M	-19%
FW3_10K	12M	8.68M	-27%
FW4_10K	5.5M	4.26M	-29%
FW5_10K	14M	9.29M	-50%
*IPC2_10K	4.6M	5.40M	14%
IPC1_100K	102M	140M	27%
IPC2_100K	424M	530.57M	25%
FW2_100K	864M	657M	-25%
FW4_100K	596M	411M	-31%

Table 4.8: estimate errors after restricting the split fields, $n = 16$.

We now show that the over-estimation comes from the assumption that all the cuts or splits are performed on IP fields. We change the code of HyperSplit so that all the splittings are restricted only on the IP fields. Before running HyperSplit on the test rulesets, we need also to remove all the $(large, large)$ rules, since these rules can be only separated by port or protocol fields. Therefore, these rules will never be separated in this modified algorithm implementation. Note that after removing these $(large, large)$ rules, the M_{ll} in our memory model equals to 0.

From Table 4.8, we can see that, after restricting the splitting fields, our model is quite accurate, the error is just around $20\% \sim 50\%$, and in the most rulesets, our estimate is smaller than actual size. This confirms our proof shown in Section 4.7.2 that our memory model is actually estimating the low bound of the memory size.

We present in Table 4.9 the results of running the splitting restricted HyperSplit on non-uniform rulesets.

We see in Table 4.9, all the rulesets, except ACL1_100K, having large actual memory size than our estimate, and also, our estimate is closed to the actual memory size. This, again, proves that our memory model is a lower bound if all the splits are performed on IP fields. The average error is only 38% in Table 4.9.

Ruleset	Actual	estimate	$\frac{est-actual}{actual}$
acl1_10K	58K	50K	-13%
acl2_10K	1088K	724K	-33%
acl3_10K	1530K	885K	-42%
acl4_10K	1633K	784K	-51%
acl5_10K	38K	27K	-28%
ipc1_10K	1011K	640K	-36%
ipc2_10K	4615K	3687K	-20%
ipc1_100K	102640K	82477K	-19%
ipc2_100K	424699K	362845K	-14%
acl1_100K	606K	1.39M	135%
acl2_100K	21452K	20214K	-5%
acl3_100K	156750K	114819K	-26%
acl4_100K	100746K	65379K	-35%
acl5_100K	499K	384K	-23%
fw1_100K	778530K	1233161K	-36%
fw3_100K	977452K	724506K	-25%
fw5_100K	1517341K	912632K	-40%

Table 4.9: estimate errors after restring split field, $n = 256$.

However, the reason listed here can not explain why the over-estimation is usually 4×. We present a decision tree model shown in Figure 4.22 to show the reason. In the model, the upper levels (1st to $N-1$th level) contain nodes cuts/splits on IP fields, while the bottom level contains nodes cuts/splits on other fields. Assume that the decision tree is a balanced binary tree, we have observed that 50% cuts/splits are performed on IP fields and 50% on other fields. Therefore we actually over-estimate the cuts/splits on IP fields by 2 times (in the memory size model, we assume all the cuts/splits are on the IP fields, however, in our decision tree model, only 50% cuts/splits actually are). Recall that the M_{ll} is calculated by the multiplication of the cuts/splits on source IP and cuts/splits on destination IP (see Section 4.3 for details). The M_{ll} is therefore over-estimated by 4 times. Since M_{ll} contributes mostly to the whole memory estimate, our memory model is therefore over-estimated 4 times of the actual memory.

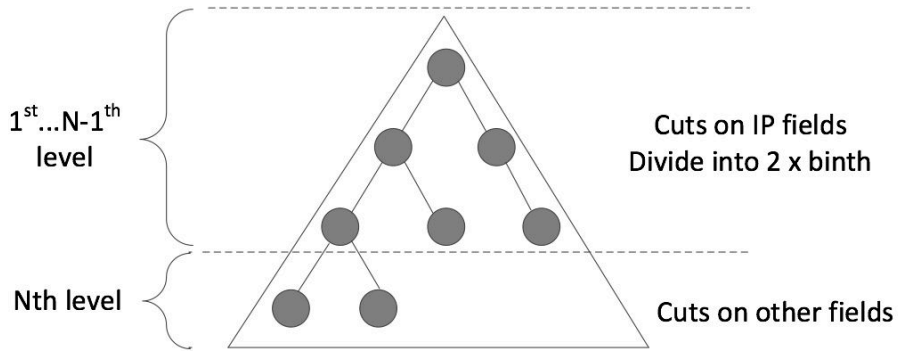


Figure 4.22: A decision tree model

Note that this decision model can also explain why most of cuts distribution is usually closed to 50%(IP fields) to 50%(other fields) in Table 4.7.

4.7.5 Comparing SmartSplit and EffiCuts

In this section we compare EffiCuts and SmartSplit that both use multiple trees. Figure 4.23 shows the memory size and number of memory accesses of EffiCuts and SmartSplit. As shown in Figure 4.23, SmartSplit outperforms EffiCuts both in memory size and in number of memory accesses. For example for fw5_100K ruleset, EffiCuts consumes 22.46MB of memory size, while the memory size of SmartSplit is only 1.98MB, about 11.3× smaller; for fw2_10K ruleset, the worst number of memory accesses for EffiCuts is 75, while the number of memory accesses for SmartSplit is only 18, about 4.1× less. These results show that using multiple algorithms for one ruleset, improve greatly the performance. Moreover this validates the fact that the "orthogonal structure" over IP fields is the main cause of high memory footprint for single decision trees. Through untangling the "orthogonal structure", the memory size decreases dramatically from several giga-bytes to less than 2 mega-bytes.

Detailed information about large rulesets is shown in Table 4.10. As mentioned above, the SmartSplit algorithm split rulesets into three sub-rulesets. We use S_m to denote the sub-ruleset resulting from merging $(small, small)$ rules with either

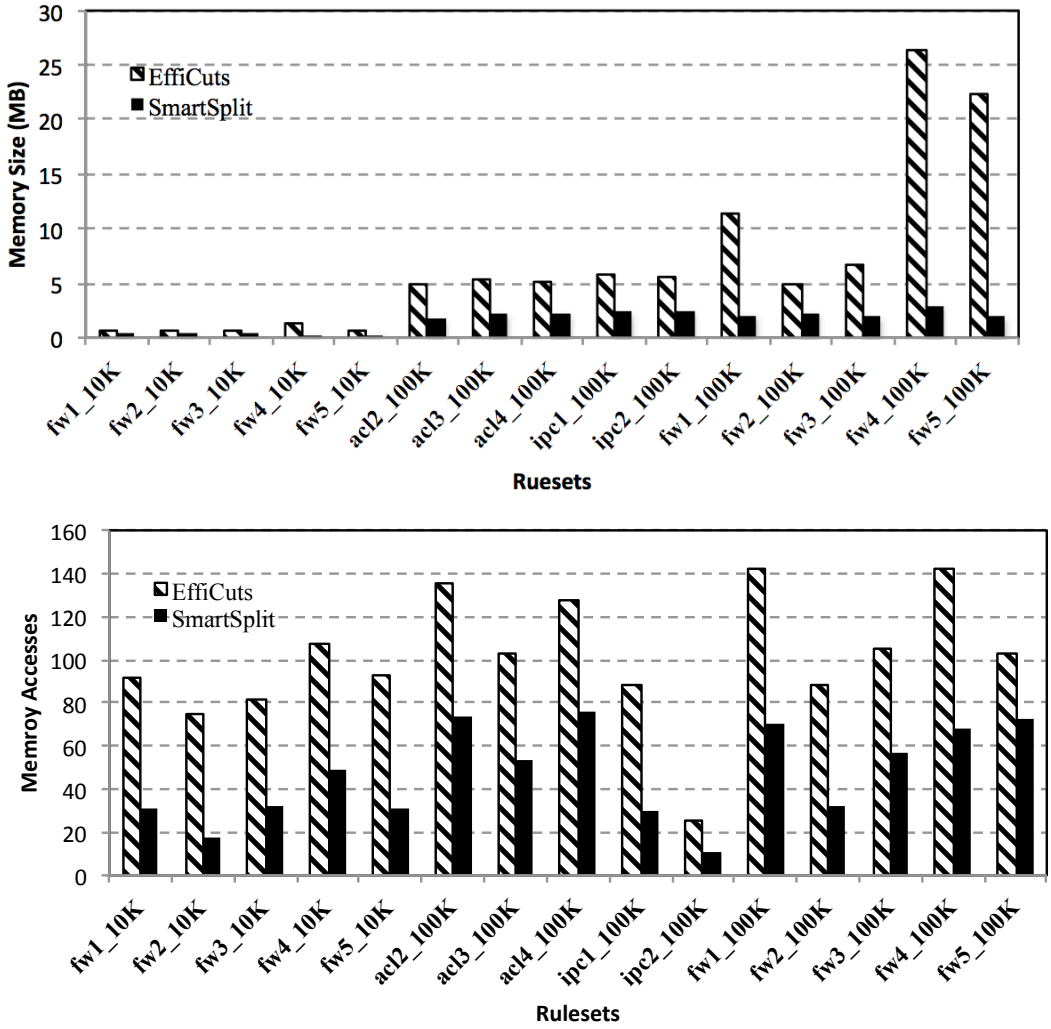


Figure 4.23: Memory and Accesses for EffiCuts and SmartSplit

$(small, large)$ or $(large, small)$ rules, S_{ll} to denote the sub-ruleset containing the $(large, large)$ rules and S_s for the other rules not merged with $(small, small)$.

Among all sub-rulesets, S_m and S_s contain more than 80% of the rules. The memory size and number of memory accesses of the decision trees built on S_m and S_s usually contribute the most in the total performance results. We therefore present the performance results of S_m and S_s in Table 4.10.

We observe in Table 4.10 that for all the FW 10K rulesets $D_{max}(srcIP)$ and $D_{max}(dstIP)$ is very small, *i.e.*, we have applied the HiCuts algorithm on both S_m

Ruleset	D_{max} srcIP	D_{max} dstIP	mem. S_m	acc. S_s	$\frac{1}{2}log_2\frac{\#rules}{binth}$	EffiCuts tree num.
fw1_10K	2	1	4	7	4	9
fw2_10K	2	2	4	4	4	7
fw3_10K	2	2	4	7	4	7
fw4_10K	4	2	10	18	4	9
fw5_10K	2	1	4	6	4	7
acl2_100K	13	13	32	25	6	7
acl3_100K	10	1	8	26	6	8
acl4_100K	10	12	31	27	6	8
ipc1_100K	1	1	8	6	6	9
ipc2_100K	2	1	6	5	6	3
fw1_100K	14	6	20	28	6	9
fw2_100K	4	2	4	18	6	7
fw3_100K	14	2	7	28	6	7
fw4_100K	5	4	27	18	6	9
fw5_100K	13	4	21	29	6	7

Table 4.10: Detailed Information of Large rulesets

and S_s. The large number of cuts per node makes the built tree "flat", reducing the total number of memory accesses of S_m and S_s from 8 to 28. Among 100K-rules rulesets, the IPC rulesets have uniform range distribution on both IP fields, therefore the total number of memory accesses of S_m and S_s is very small (only 11 and 14).

The FW 100K rulesets have uniform range distribution on destination IP field and non-uniform range distribution on source IP field, so that SmartSplit applies HyperSplit on S_s resulting in small memory size, from 100KB to 400KB in our experiments, and HiCuts on S_m for fewer memory accesses. We see the number of memory accesses of S_s increases to around 30 while this value for S_m is still small. However, since the SmartSplit algorithm only generates 3 sub-rulesets, the total number of memory accesses remains small, while, EffiCuts algorithm usually builds $5 \sim 9$ trees on large rulesets and yields more than 100 memory accesses.

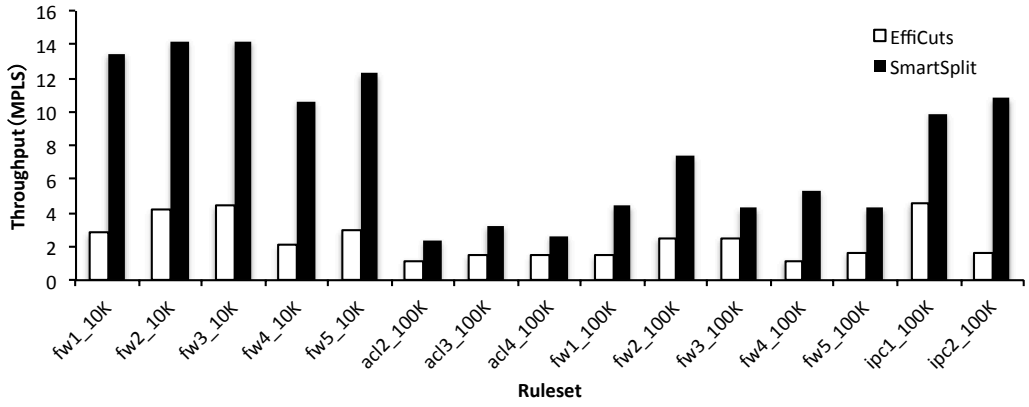


Figure 4.24: Comparing the measured performance of SmartSplit and EffiCuts

4.7.6 Real Performance Evaluation

We implement an optimized and fast packet matching program capable of loading the built tree data structure from multiple algorithms into memory and performing rule matching using the resulting DTs. We implemented HiCuts, HyperSplit and SmartSplit in the packet matching program, and used AutoPC framework to configure the program. We also implement EffiCuts, but disabling the `Node Co-lacation` and `Equal-dense Cuts` optimization tricks described in [77] to simplify the implementation. It is noteworthy that in Section 4.7.5, we compared SmartSplit with EffiCuts enabling all its optimizations.

We first compare the real measured performance of SmartSplit and EffiCuts on rulesets with large memory size in Figure 4.24. We see that SmartSplit runs significantly faster than EffiCuts. For all the FW 10K rulesets, SmartSplit achieves beyond 10 Millions of Lookup Per Second (MLPS) while EffiCuts only achieves $2 \sim 4$ MPLS. For larger rulesets, SmartSplit is usually 2 times faster than EffiCuts.

We present in Table 4.11 the lookup speed of AutoPC and EffiCuts in terms of millions of lookup per second (MLPS)[3]. The evaluation shows that the AutoPC

[3]the results marked with * means AutoPC builds a single tree on the ruleset

Type	Size	AutoPC(MLPS)	EffiCuts(MLPS)	speedup
ACL	1K	11.3*	4.5	2.4
	10K	6.9*	3.1	2.2
	100K	8.6*	2.2	3.9
FW	1K	9.8*	2.4	4.1
	10K	10.7	2.1	5.1
	100K	7.4	2.5	3.0
IPC	1K	12.6*	3.0	4.25
	10K	5.3*	1.48	3.6
	100K	9.91	1.63	6.1
Average Speedup: 3.8				

Table 4.11: Real Performance Evaluation of AutoPC and EffiCuts

framework is in average 3.8 times faster than using EffiCuts solely on different type of rulesets.

4.8 CONCLUSION

In this work, we identify the intrinsic characteristics of rulesets that yield the performance unpredictability issue in the decision-tree based algorithms. Based on these observations, we propose a memory consumption model, a "coverage uniformity" analysis algorithm and an framework capable of identifying which algorithm is suited for a given ruleset through combining the model and the analysis algorithm.

The experimental results show that our method is effective and efficient. Our SmartSplit algorithm is significantly faster and more memory efficient than the state-of-the-art work, and our AutoPC framework can automatically perform memory size and accesses tradeoff according to the given ruleset. In the experiments, compared to EffiCuts, the SmartSplit algorithm has achieved up to 11 times less memory consumption as well as up to 4 times few memory accesses. The real performance evaluation shows that SmartSplit is usually $2 \sim 4$ times faster than EffiCuts. The AutoPC framework achieves in average 3.8 times faster classification performance than using EffiCuts solely on all the rulesets.

Besides these performance improvements, we believe that the observations in this chapter provide a new perspective to understand the connection between ruleset features and the performance of various decision-tree based algorithms.

Chapter 5

Evaluating and Optimizing IP lookup on Manycore Processors

5.1 Introduction

We have discussed the multi-dimensional packet classification algorithms from algorithmic and ruleset feature perspective. In this chapter, we will discuss another type of packet classification problem: IP-lookup. Among all the forwarding tasks in the data-plane of a router, IP lookup is obviously a critical one. Routing tables in nowadays core routers can easily grow to several hundred thousand of prefixes, resulting in large FIB (forwarding information base) sizes and slow lookup speed.

In this work, we will evaluate the performance of algorithmic-based IPv4 lookup algorithms on a popular highly multi-core processor, the TILEPro64 processors. TILEPro64 processors contain 64 full programmable processing cores. A full TILEPro64 development board that can support up to 8×1 Gbps plus a 1×10 Gbps Ethernet Interface costs currently several thousand dollars, making this platform affordable for practical usage as a software router. Indeed more powerful processors

are available these days meaning that the results presented in this paper are just lower bound on potential IP lookup speeds.

For the software IP lookup we choose two simple and practical algorithms, DIR-24-8-BASIC [21] and Tree Bitmap [18]. DIR-24-8-BASIC is used in many software router prototypes, such as RouteBricks [16], PacketShader [24] *etc.*, while Tree Bitmap is a well-known IP lookup algorithm with low memory footprint and fast lookup speed. Other algorithms are either too complicated or not suitable for the multicore platform. For example, Bloom Filter based IP lookup needs hardware implementation of several hundreds of hash functions, that will need dedicated FPGAs, while our aim in this paper is to study a software only implementation on a many-core platform.

In order to get a full understanding of performance of different algorithms, we do our evaluation experiments on a routing table issued from RouteViews project [73] and containing about 358K prefixes. We've found that, in a single core environment, the DIR-24-8-BASIC algorithms run at least 3 times faster than Tree Bitmap on all IP traces. However, the FIB size generated by Tree Bitmap is almost 20 times lower than DIR-24-8-BASIC. In the parallel experiments, we have observed that the run-to-complete execution model is superior to the pipeline model. Our experiment shows that, by using only 18 cores out of 64 cores on TILEPro64, we can achieve a lookup throughput of up to 60Mpps (almost 40Gbps for 64 bytes per packet) with a power consumption of less than 20W [74], to be compared with 240W for GPU based PacketShader. Moreover as the packet processing is done directly on the TILEPro64 processor the lookup delay is very small compared to the delay needed for batching in PacketShader.

The contributions of this work can be summarized as follows: 1) we describe and evaluate how IP lookup algorithms can be implemented in practice on a many-core processor—TILEPro64. We implemented various optimization tricks, including both algorithmic refinements and architecture specific optimizations. 2) We measured the

performance of different IP lookup algorithms on many core chips using different traces. 3) Based on our evaluation results, we propose a hybrid scheme SplitLookup to combine the strengths of two algorithms. This hybrid scheme has the similar performance with DIR-24-8-BASIC on single-core but has a much smaller update overhead in the worst case.

The remainder of this paper is organized as follows. In Section 2, we will provide some background, including the two algorithms and the TILEPro64. In Section 3, we will present our implementation and the implemented optimizations. In Section 4, we will report our hardware setup and experimental evaluation. In Section 5, we will present a hybrid IP lookup scheme and evaluate its performances. We conclude this work in Section 6.

5.2 Background

5.2.1 The Tree bitmap algorithm

The *Tree Bitmap* algorithm is a multi-bit trie IP lookup algorithm using a clever encoding scheme. Fig. 5.1 shows an example of a 3-bit stride Tree Bitmap trie. In Fig. 5.1, we can see the whole binary trie is divided into several multi-bit nodes having two bitmaps, the internal bitmap (IBM) and the external bitmap (EBM). The IBM is used to represent the prefixes stored in this multi-bit node, and the EBM is used to represent the position of the child of this multi-bit node.

We use the Node A as an example to show the encoding scheme of the IBM and the EBM. A 3-bit sub-trie has 8 possible leaves. In Node A, only the first and fourth leaves have the pointers to children. Thus the EBM of this node is 10010000. The encoding scheme of IBM is a little bit complicated: we firstly turn this 3-bit sub-trie into a full binary tree with 7 nodes by adding "virtual nodes", then we traverse this tree in level oder. In each level, we traverse the nodes from left to right. If the node

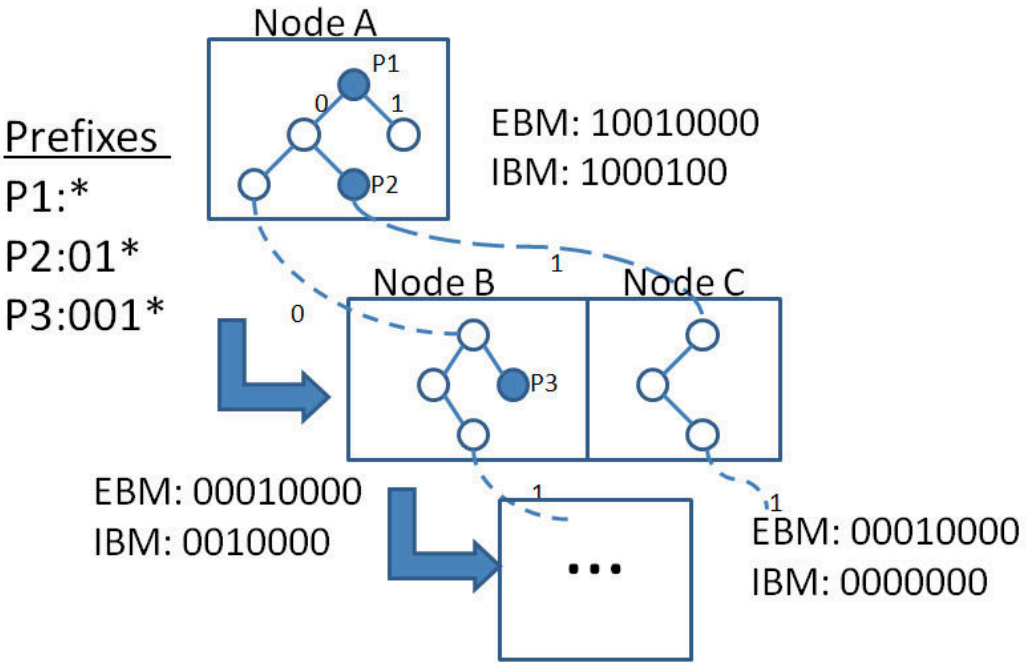


Figure 5.1: The Tree Bitmap Algorithm

stores one prefix, we set the bit otherwise we clear the bit. In Node A, we have two stored prefixes, P1= * and P2=01*. P1 is stored in the first node we traverse, so the first bit of IBM is 1, P2 is stored in the fifth node, so we set the fifth bit of IBM. The IBM of this node is 1000100. To note, a K stride node has a 2^K bit EBM and $2^K - 1$ bit IBM. These bitmaps provide a compact way to store the location information of the child node or prefixes. For example, if we search Node A with bits 011, we check the fourth bit of IBM and also count the number of bit set to the left of fourth bit. There is only one bit set, thus the child node's address can be retrieved by $P + 1 \times S$, where P is the pointer stored in Node A, and S is the node size.

While checking EBM is easy, checking IBM is a little complicated. For example, if we want to check IBM of Node A with bits 011, we need successively remove the right-most bits of the bit sequence, and check the corresponding bit position in IBM, until we find the bit set in that position. In the first iteration, we get 01 after remove the last bit. We walk the sub-trie following 01, and stop at the node to check if there

is one prefix stored. We firstly use the IBM traverse way to determine the number of this node (fifth), and then check the corresponding bit in IBM. We find the fifth bit of IBM is set, and we know there is a prefix matching 011. Same method described above can be used to retrieve the location of this prefix.

5.2.2 The DIR-24-8-BASIC algorithm

Compared to the Tree Bitmap algorithm, *DIR-24-8-BASIC* is much simpler. It uses two tables to store all the prefixes. The first table, *TBL24* which uses the first 24 bits of an IP address as an index, stores all the prefixes with length shorter than 25 bits. If more than one prefixes share the same first 24 bits, the corresponding entry of these prefixes in *TBL24* is filled with a pointer pointing to a 256 entries block in the second table, *TBLlong*, storing all the possible suffix of the left 8-bits. When there is only a single prefix with matching first 24 bits, *TBL24* contains the next hop information. However *TBLlong* always contains the next hop information.

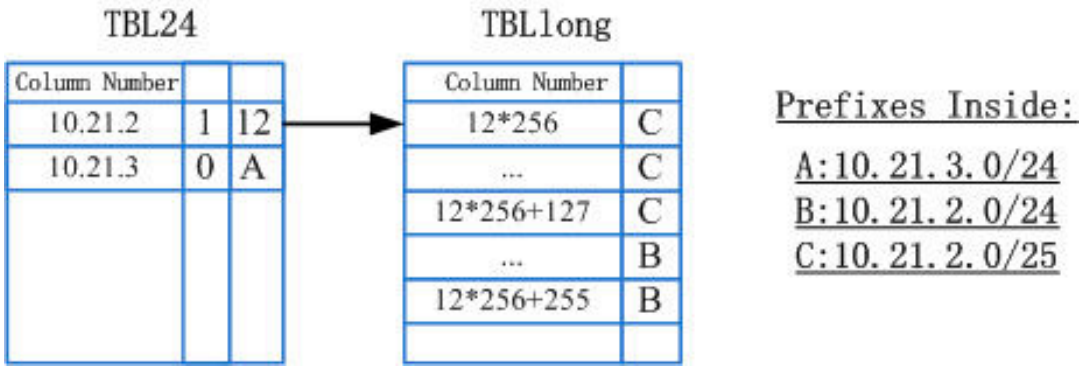


Figure 5.2: The DIR-24-8-BASIC Algorithm

An example is shown in Fig. 5.2, where no prefixes share the first 24 bits of Prefix 10.21.3.0/24, thus the egress A is directly stored in *TBL24*; Prefix 10.21.2/24 and 10.21.2.0/25 has the same first 24 bits, thus the corresponding entry in *TBL24* stores a pointer which points to the 12th block in *TBLlong*. When searching an IP address, we first use the first 24-bits of IP address as an index to read one entry of Table

Table 5.1: frequency and cache size

type	clock frequency	cache size
TILEPro64	700MHz	L2 64KB / L3 4MB
E5506	2133MHz	L2 1MB / L3 4MB

Table 5.2: cache system and cache miss penalty

type	cache system	penalty(cycles)
TILEPro64	distribute	L2 8 / L3 30 $\sim$ 80
E5506	distribute	L2 14 $\sim$ 15 / L3 $\sim$ 100

TBL24. Depending on the content of *TBL24*, the lookup is terminated or we proceed to table *TBLlong* following the pointer in *TBL24*.The leftmost 8-bit of the IP address are used to obtain the index of the prefix in Table *TBLlong* and access it with one more memory access. Since currently, most of prefixes have length less than 25 bits in the core routing table, it only takes one memory access to do any IP lookup.

5.2.3 The TILEPro64 architecture

TILEPro64 is a many-core processor based on Tile Architecture that consists of a 2D grid of homogeneous computing elements, called tiles or cores. Each tile is a full-featured CPU that can independently run an entire operating system. As the name implies, TILEPro64 consists of 8×8 cores, that is much larger compared to mainstream multi-core processors, which usually have only $4 \sim 8$ cores. However, TILEPro64 cores have differences with for example an Intel Xeon cores. Table 5.1 lists the differences between a TILEPro64 core and an Intel Xeon E5506 one.

As can be seen from Table 5.1 and Table 5.2, a TILEPro64 core is relatively weaker than one in an Intel Xeon E5506. Therefore, while we can assign heavy processing to a single Intel Xeon core, *e.g.* all the processing of a software router's dataplane, including the decoding, IP lookup, checksums, *etc.* in a single thread on a single core of a Xeon E5506, on TILEPro64 we split different dataplane activities between several cores. Following this, we have assigned entire cores of the TILEPro64 to only

do IP lookup. As the programmable on-chip network on TILEPro64 can be used to eliminate the communication overhead of adding to other cores other dataplane activities, and the distributed cache system ensures that the cache isolation, we can evaluate the IP lookup load independently of the other activities of the control plane that will be assigned to other cores.

5.3 IP Lookup on TILEPro64

In this section, we present our implementation and detail the optimization tricks we used.

5.3.1 Implementation

Tree Bitmap: We implemented two versions of this algorithm, *TreeU16* and *TreeU32*. The *TreeU16* implementation uses the built-in type *uint16_t* in TILE64 core to store the bitmaps inside the multi-bit node. This implementation is specially suitable for Tree Bitmap with 4 bits stride as it eliminates the overhead of querying the stride information during the lookup process. The *TreeU32* implementation is more general and uses array *uint32_t* type to store the bitmaps. This implementation can be tuned to any trie with 5 bits or more strides. In both implementations, a single 32-bit pointer is used to point to both the child and result arrays. Therefore each node of *TreeU16* needs $2 \times 2 + 4 = 8$ Bytes, and each node of *TreeU32* costs $2 \times 4 \times 2^{(stride-5)} + 4$ Bytes.

DIR-24-8-BASIC: We implemented *DIR-24-8-BASIC* using 32-bits integer for each entry of both *TBL24* and *TBLlong*. For each entry of Table *TBL24*, one bit is used as a flag to signal if this entry point to *TBLlong* or if it is a definitive prefix, 5 bits are used to store the prefix length, and 26 bits are used to store an index or a pointer to the next-hop information. The 26 bits index is necessary for lookup

and update. In each entry of table *TBLlong*, 5-bits are used to store the prefix length, and the leftover bits are used as a pointer to the next hop information. As table *TBL24* needs 2^{24} entries, our implementation of DIR-24-8-BASIC needs at least 64MB DRAM memory (4 bytes per entry in table *TBL24*).

5.3.2 Optimization tricks

Large page

Rather than using by default the 4KBytes page, we have used 16MB large page in our development. This optimization reduces the TLB misses during the IP lookup. Algorithms like DIR-24-8-BASIC which uses a large amount of memory can be benefit from this trick. In our experiment, we find this is even beneficial for the Tree Bitmap which uses much less memory. Our experiment shows that this implementation detail highly improves the performance the lookup by decreasing lookup time by almost 20%.

Initializing an array for trie

One way of improving the lookup speed is to implement a lookup table for the first consecutive bits in the trie-based IP lookup. For example, for the first 13 bits of an IP address, we build an initial array with 8K entries that enables fast access to the node storing these prefixes. The array speeds up the lookup, however it increases the update overhead. In our implementation of Tree Bitmap, we have used such an array both in *TreeU16* and *TreeU32*.

Counting the number of 1s in a bitmap

The Tree Bitmap algorithm needs to count how many 1s are in one bitmap. This task can easily be done in hardware. However, in software, it is more complex and one have to use a lookup table to get the number of 1s in one bitmap. This adds more memory accesses during the lookup and degrades the performance. We use

the special purpose arithmetic instruction *popcount* to count the number of 1s in a bitmap. This instruction is common in many off the shelf processors.

Lazy checking

As mentioned above, one single multi-bit node can have two operations: checking the EBM to find the "exit point" and checking the IBM for the prefixes inside the node. Since the IBM checking is time consuming, we perform a lazy checking, *i.e.* we only check the EBM of traversed node and we use an extra stack to store them; when the searching cannot proceed to the next node, we pop the nodes in the stack to perform IBM checking. As long as there is a single prefix match, the lookup process terminates. Our experiment shows this trick can save up 30 to 50 cycles per lookup.

Fast internal bitmap checking

The checking of internal bitmap is time consuming. In the original implementation of Tree Bitmap, for a k bit stride internal bitmap checking, one needs k clocks to check the specific bit positions in the internal bitmap. In order to accelerate the checking, we use a small lookup table and some specific bit instructions in our implementation. For k bit stride internal bitmap, we construct a lookup table with 2^k entries; each entry corresponds to one possible input. In each entry, we store a pre-computed bit string recording all the checking bit positions for the corresponding input. For example, in the sub-tree shown in Figure 5.3, for the input bit string 000, one needs to check the first, the second and the fourth bit in the internal bitmap, therefore the pre-computed bit string is 1101000.

When checking the internal bitmap, we perform the "AND" operation between the pre-computed bit string and the internal bitmap. The position of the first set bit is the checking result. For example, in Figure 5.3, the internal bitmap is 1000100, the

result bit string is 1000000, and the position of the first set bit is 0. Therefore the internal bitmap checking returns 0.

Fortunately, most CPU provides a single instruction clz for the position searching of the first set bit. Therefore, our implementation can finish checking using only a few instructions (one table lookup, one "AND" and one clz).

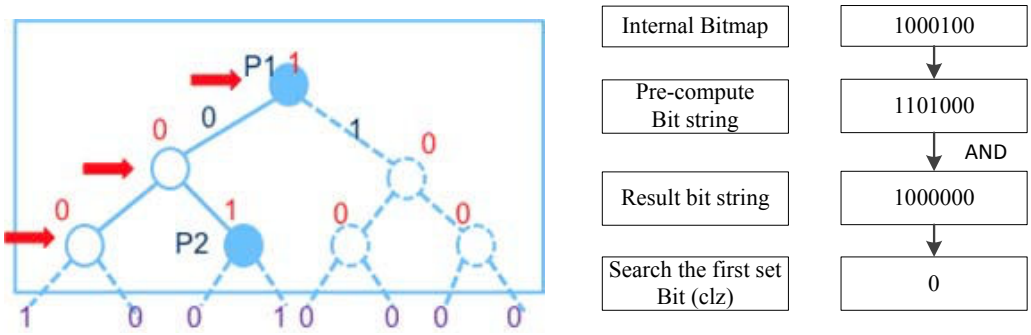


Figure 5.3: Fast internal bitmap checking

5.4 Performance Evaluation

In this section, we evaluate the performance of the two IP lookup algorithms with both synthetic and real world traces. We discuss our evaluation traces and the performance results in detail.

5.4.1 Evaluation Traces

The nature of the traces used to evaluate IP lookup is very important. An IP trace with high locality can lead to a very high performance result because many memory accesses cached in the L2 cache of CPU can be reused. Random IP traces, while having limited locality, contain too many "invalid" IP addresses (IP address does not match any prefix in one routing table), that usually have very short searching path in the trie-based IP lookup algorithm. Using these trace can also result into

"illusion" high performance. In order to get a full understanding of the performance of IP lookup in software, we use both synthetic and real world traces. We use three types of traces that are listed below:

Random Match: Let S be the set of all the prefixes in one routing table. We use the prefixes in S to construct a binary trie and we leaf push this trie, *i.e.* all the prefixes are stored only at the leaf nodes. We are representing as $L(p)$ the leaf nodes that stores the prefix $p \in S$. For any prefix $p \in S$, let us define a set, $P(p)$, containing all paths starting from the root node and ending at the leaf nodes that belong to $L(p)$. These paths can be viewed as the "leaf pushing" prefixes for the original prefix p. For any prefix $p \in S$, we collect the longest path in $P(p)$, and use this paths to form a new prefix set. We call this set the Random Match Set. Random Match traces are generated by repeating the following steps:

1. Choose randomly one prefix in the Random Match Set.

2. If the prefix is not 32-bit long, we use a random number to complement this prefix into a 32-bit IP address.

The Random Match trace has three characteristics: 1) it is unbiased for all prefixes in S; 2) it has low locality; 3) IP addresses in Random Match trace have the longest searching path. Thus, the performance result gathered on this trace can be considered as the worst case for all implementation. Fig. 5.4 shows how to construct a Random Match trace.

Realistic Random Match: We now add some locality to our evaluation trace thanks to realistic traces. We have used traces provided by CAIDA [30], and we have extracted all the destination IP addresses. Unfortunately, these IP addresses can not be used directly, because they are anonymized and many of them cannot match any prefix in a real routing table. However the anonymization maintains the

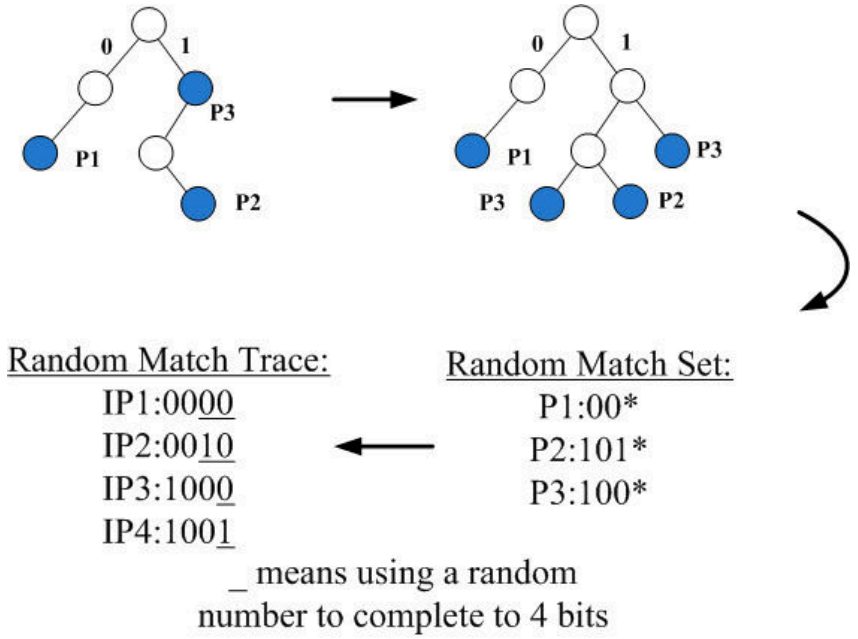


Figure 5.4: Random Match Trace Generation

prefix structure. Therefore, in order to generate a trace with realistic locality, we have replaced the anonymized IP addresses with "valid" IP addresses:

1. We define an association array H mapping anonymized addresses to the "valid" one.

2. For any anonymized IP p, if there exists $H[p]$, we replace it with $H[p]$

3. If not, we generate a "valid" IP address q using the method described in random match and we replace p with q, and let $H[p] = q$.

We can expect to have higher performance on this trace as realistic locality is enforced.

Realistic and Filtered: For this trace we directly used the anonymized realistic trace coming from CAIDA. We filtered out all "valid" IP addresses. This trace will have the highest locality among the three kinds of traces. However the trace will only match a small fraction prefixes in a routing table.

Name	unique IP addresses	Generated from
Random Match	353398	routing tables from [73]
Realistic Random Match A	24424	[30]
Realistic and Filtered A	17020	[30]
Realistic Random Match B	81811	[30]
Realistic and Filtered B	41654	[30]

Table 5.3: Evaluation Traces

We have generated five traces: one Random Matched, two for Realistic Random Matched and two Realistic and filtered ones. Each trace was containing 1 million IP addresses. The detailed information for these traces is listed in Table 5.3.

5.4.2 Single-core Performance Evaluations

In each experiment, we have used two cores: one core only for loading the traces and extracting the IP addresses, the other core receiving the IP addresses on the on-chip network and doing the IP lookup. We have used two configurations for Tree Bitmap, one using an initial array of 2^{13} entries, and 4 bit stride; the second using an initial array of 2^{11} entries, and 7 bits of stride. We name them respectively TBP 13-4-4-4-3 and TBP 11-7-7-7. The memory footprint of FIB generated by DIR-24-8-BASIC, TBP 13-4-4-4-4-3 and TBP 11-7-7-7 is respectively 69.1MB, 2.9MB and 4MB. Fig. 5.5 shows the performance results for a single core of TILEPro64.

From the Fig. 5.5, the following observations can be made. First, the lookup speed is highly related to the number of memory accesses. Although the FIB size of Tree Bitmap is almost 20 times less than DIR-24-8-BASIC, DIR-24-8-BASIC is still 3 times faster. Second, the time spent on processing instructions can not be ignored. Small stride leads to a faster IBM checking, which makes TBP 13-4-4-4-4-3 faster. Third, the locality of the trace determines the final performance. To note, we measure the lookup speed in cycles. The clock frequency of TILEPro64 is 700MHz,

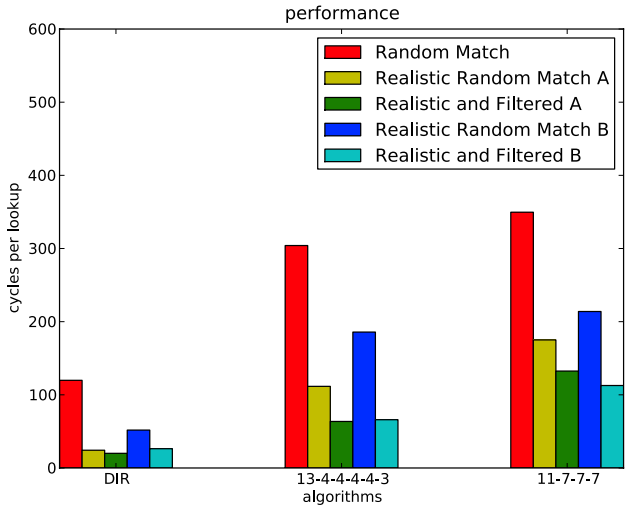


Figure 5.5: Single core Performance Results

which means that each cycle is 1.4 ns. So in the single-core environment, at least 168ns are required per lookup for the fastest case.

5.4.3 Parallel Performance Evaluations

We have used for the experiment in this section two parallel execution models: pipeline and run-to-complete model.

The pipeline model is only applied to the Tree Bitmap algorithm. In the pipeline model, for each IP lookup, each core only needs to do the processing of one multi-bit node (including both the IBM and EBM checking) in one level of the Tree Bitmap trie, then transfer the intermediate result to the next core. There are many proposed algorithms [28] [2] to balance the memory utilization of each pipeline stage. However these works assume that the IP lookup engine has multiple single port memories. For example, [28] splits the whole IP lookup into 24 stages, requiring 24 banks of single port memory. TILE64Pro only has 4 DRAM memory interface which does not conform this assumption. So we do not adopt any of these algorithms and simply divide the Tree Bitmap trie by its levels.

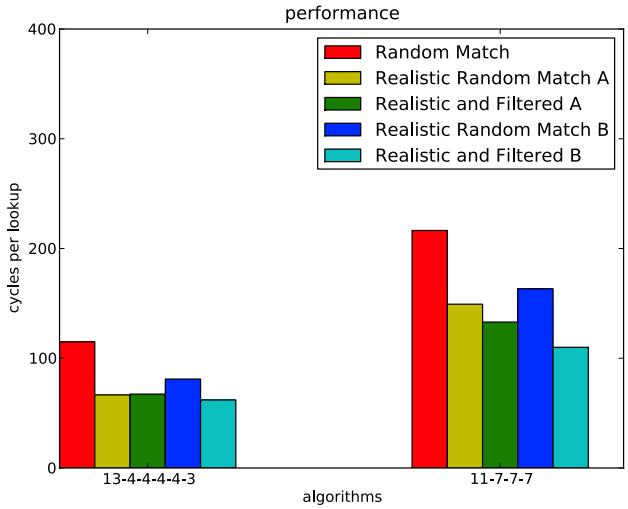


Figure 5.6: Pipeline Parallel Performance of Tree Bitmap

We use 5 cores for TBP 13-4-4-4-4-3 and 3 for TBP 11-7-7-7. It is noteworthy that in the pipeline model, we cannot perform the "lazy checking" optimization trick. We show in Fig. 5.6 the performance achieved by the pipeline. Compared to the Fig. 5.5, we can observe that the performance gain is about 3 fold. This can be explained as most of the IP addresses in the evaluation traces match prefixes that have length less than 25. Looking up these IP addresses only needs 3 to 4 memory accesses. So in average, the speed up rate is around 3 times. And once again, TBP 13-4-4-4-4-3 is faster.

In Fig. 5.7-5.8 we show the performance achieved by the run-to-complete model. In this approach, one core is used as a dispatcher that splits the workload by forwarding the IP addresses to the other cores in a round-robin fashion. Whenever a core finishes its lookup, a new IP address is forwarded to it and looked up. In this model all algorithms parts run in parallel. In both figures, *Limit* represents the average transfer time of the on-chip network, it also provides an upper bound on the performance we can achieve.

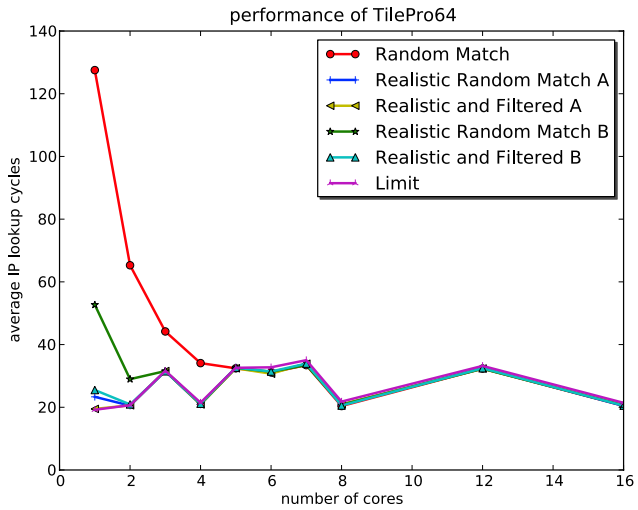
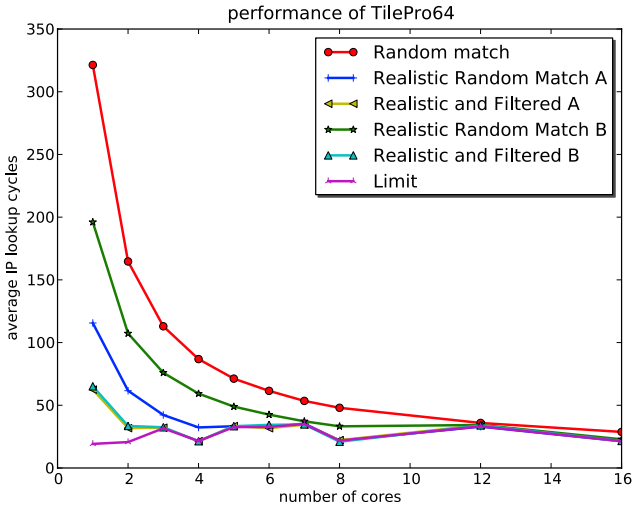


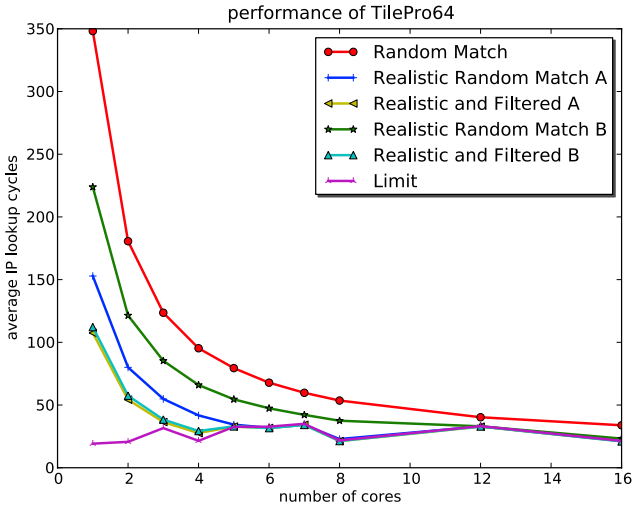
Figure 5.7: Run-to-complete Parallel Performance of DIR-24-8-BASIC

Fig. 5.7 shows that the highest performance, about 20 cycles per lookup, is achieved when the number of parallel cores reaches 8. This is equivalent to about 20Gbps of throughput when packets are 64 Bytes. TILEPro64 has four memory controllers and we only use one of them in our experiment. This means that, if necessary, the two memory controllers can be used to provide enough memory bandwidth to support 18 cores (2 for dispatching and 16 for lookup) reaching a 40Gbps lookup throughput.

In Fig. 5.8, as the number of lookup cores increased, the performance increased almost linearly (or the lookup time decreases). However, we achieve at best 28 cycles per lookup by using 16 cores, which is still slower than DIR-24-8-BASIC. This confirms that DIR-24-8-BASIC is superior in speed to tree bitmap (as it lookup time is 8ns less) at the cost of a memory footprint that is 20 times larger.



(a) TBP 13-4-4-4-3



(b) TBP 13-4-4-4-3

Figure 5.8: Run-to-complete Parallel Performance of Tree Bitmap

5.5 A Hybrid IP Lookup Scheme: SplitLookup

From the evaluation above, we can conclude that the DIR-24-8-BASIC runs faster than Tree Bitmap on average. However, this algorithm suffers from a high update overhead. Suppose we want to delete a /8 prefix, we need $2^{24-8} = 65536$ memory accesses. This worst case update overhead may become a performance bottleneck

in practice. In contrast, the update overhead of Tree Bitmap is much less. In this section, we propose a hybrid IP lookup scheme to combine the strength of both.

The root of high update overhead lies in the short prefixes ($<$ /17) stored in *TBL24*. These short prefixes overlap a large range in *TBL24*. Updating these prefixes need to modify all the entries in this range. In order to prevent the high overhead, one can put all these short prefixes in a Tree Bitmap trie. This will result as a side effect, reducing the number of memory access, because these prefix are all near the root node in the trie. In TBP 13-4-4-4-4-3, only one memory access is needed for such short prefixes. As mentioned above, the lookup speed is highly related to the number of memory access. So this hybrid scheme can also achieve high performance for these short prefixes.

The basic idea of our hybrid lookup scheme is as follows:

1. Store the short prefixes of length 1 to 16 in a Tree Bitmap trie.

2. Store the prefixes of length 17 to 24 in the Table *TBL24*.

3. For the prefixes of length 25 to 32, we use only one entry in Table *TBL24* to store a pointer and put the remaining 8-bit in a sub-trie.

A simple comparison of update overhead is listed in Table 5.4. We measure the average memory operation accesses in both DIR-24-8-BASIC and TBP 13-4-4-4-3 when adding and deleting all the /8 $\sim$ /16 prefixes in our routing table. There are 12894 prefixes in total. As mentioned, one entry of DIR-24-8-BASIC is 4 bytes; one node of TBP 13-4-4-4-3 is 8 bytes. From the table, we can estimate the update overhead of TBP-13-4-4-4-3 is about several hundreds times less than DIR-24-8-BASIC. Because our hybrid algorithm uses the TBP 13-4-4-4-3 to store these prefixes, we can conclude the update overhead of this hybrid algorithm is much less.

The lookup process on the hybrid scheme is similar to the lookup process in DIR-24-8-BASIC. We first perform the long prefix lookup ($>$ /16) using Table *TBL24*

Table 5.4: update overhead of two algorithms

Algorithms	Add/Del	Entry set	Node copy	Node alloc
DIR-24-8-BASIC	Add	586.67	null	null
	Del	553.95	null	null
TBP 13-4-4-4-3	Add	0.40	1.15	0.97
	Del	0.39	1.43	0.97

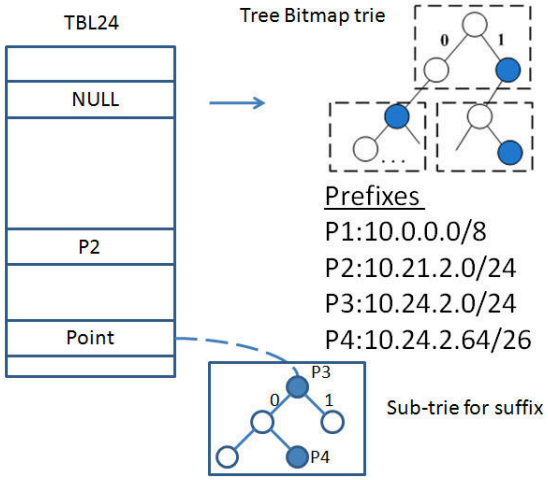


Figure 5.9: Data Structure of the hybrid algorithm

and the attached sub-tries. If there are not any prefixes matching this IP address, we perform the lookup process on the independent tree Bitmap trie which stores the short prefixes. The data structure of the hybrid algorithm is shown in Fig 5.9.

Fig. 5.10 shows the performance achieved by our proposed hybrid scheme on a single-core. We used the TBP 13-4-4-4-4-3 as the independent trie, and stride of 4 as the sub multi-bit tree attached to Table *TBL24*.

From the Fig. 5.10, we see that, as we expected from the design, our hybrid scheme achieves a performance similar with the DIR-24-8-BASIC. We now give a brief analysis of the worst case update overhead. As mentioned above, the worst case update overhead of our hybrid scheme is bound by the update overhead of Tree Bitmap. When updating happens in Tree Bitmap algorithm, the worst case is to reconstruct a full child array. In our case, we use a stride of 4, which means the largest child array has up to 16 multi-bit nodes. So the update overhead is bounded

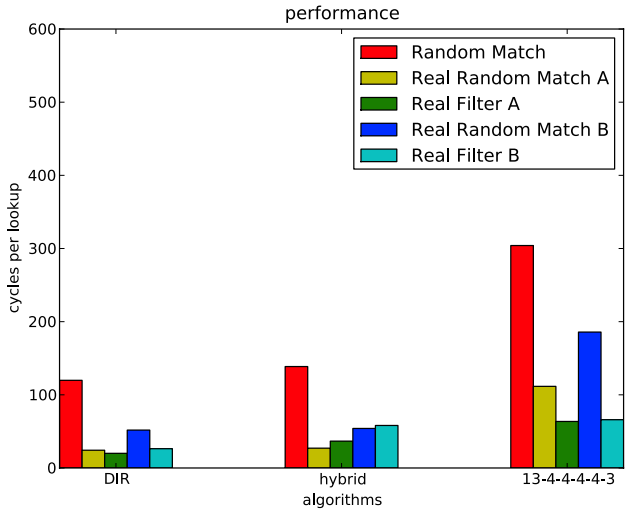


Figure 5.10: Performance of our hybrid IP lookup scheme

by $16 \times 8 = 256$ bytes memory copy. Compared to DIR-24-8-BASIC, which needs larger than 65536 memory accesses in the worst case, our scheme has a much less update overhead. However the memory footprint of the hybrid scheme and the DIR-24-8-BASIC are comparable as the *TBL24* is reused.

5.6 Conclusion

To summarize, in this work, we implemented two widely used IP lookup algorithms on TILEPro64 and evaluated the performance of them with both synthetic and real world traces. We have achieved a throughput of 40 Gbps by using 18 cores of TILEPro64. Compared to the work of PacketShader which uses GPUs [24] to do the IP lookup, the power consumption of our solution is much lower. We also found that, on our platform, the IP lookup speed is highly related to the number of memory accesses. Although the small sized FIB can be easily cached, IP lookup with less memory accesses is always faster. We evaluated the performance of different parallel model. Our experiments show that the run-to-complete model is more efficient on many core

chips. In the end of this paper, we propose a new hybrid IP lookup scheme which provides a low bound to the worst case update overhead for DIR-24-8-BASIC. Our work demonstrates the performance power of many core chips, and also gains some insight into the IP lookup on many-core processors.

Chapter 6

PEARL: A Programmable Virtual Router Platform

6.1 Introduction

Deploying, experimenting and testing new protocols and systems over Internet have always been a major issue. While, one could use simulation tools for the evaluation of a new systems aimed toward large-scale deployment, real experimentation in experimental environment with realistic enough settings, such as real traffic workload and application mix are mandatory. Therefore easy programmable platforms that can support high-performance packet forwarding and enable parallels deployment and experiment of different research ideas are highly demanded. Moreover, we are moving toward a future Internet architecture that seems to be polymorphic rather than monolithic, i.e., the architecture will have to accommodate simultaneous coexistence of several architecture (like the Named Data Network (NDN) [27], etc.) including the current Internet. Therefore future Internet could be based on platforms running different architectures in virtual slices enabling independent programming and configuration of functions of each individual slice. Current commercial routers, the most

important building blocks of the Internet, while attaining very high performance, only offer a very limited access to the researchers and developers to their internal component to implement and deploy innovative networking architecture. In contrast, open software based routers naturally facilitate the access and adaptation of almost all their components however, often with a low packet processing performance. As an example, recent OpenFlow switches [38] provides flexibility by allowing programmers to configure the 10-tuple of flow table entries, enabling to change the packet processing of a flow. OpenFlow switches are not ready for non-IP based packet flows, such as NDN. Moreover, while the switches allow a number of slices for different routing protocols through the FlowVisor, the slices are not isolated in terms of processing capacity, memory and bandwidth. Motivated by these facts, we have designed and built a ProgrammablE virtuAl Router pLatform, named PEARL that can guarantee high performance. The challenges are two-fold: first to manage the flexibility vs. performance trade-off that translates into pushing functionality to hardware for performance vs. programming them in software for flexibility, second to ensure isolation between virtual router instances both in hardware and software with low performance overhead. This chapter describes the PEARL routers architecture, its key components and the performance results. It shows that PEARL meets the design goals of flexibility, high performance and isolation. In the next section, we describe the design goals of the PEARL platform. These goals can be taken as the main features of our designed platform. We then detail the design and implementation of the platform, including both hardware and software platforms. In the next section, we evaluate the performance of a PEARL based router using the SPIRENT TestCenter by injecting into it both IPv4 and IPv6 traffic. Finally, we briefly summarize the related works, and conclude the paper.

6.2 Design Goals

In the past few years, several future Internet architectures and protocols at different layers have been proposed to cope with the challenges that the current Internet faces [6]. Evaluating the reliability and the performance of these proposed innovative mechanisms is mandatory before envisioning a real scale deployment. Besides theoretically evaluating the performance of a system and simulating these implementations, one needs to deploy them in a production network with real user behavior, traffic workload, resource distribution, and applications mixture. However, a major principle in experimental deployment is the *"No harm"* principle that states that normal services on a production network should not be impacted by the deployment of a new service. Moreover, no unique architecture or protocol stack will be able to support all actual and future Internet services and we might need specific packet processing for given services. Obviously, a flexible router platform with high-speed packet processing ability and support of multiple parallel and virtualized independent architectures is extremely attractive for both Internet research and operation. Based on this observation one can define isolation, flexibility, and high performance as the needed characteristics and the design goals of a router platform future Internet. In particular, the platform should be able to cope with various types of packets including IPv4, IPv6, even non-IP and be able to apply packet routing as well as circuit switching. Various software solutions like Quagga or XORP [25] have provided such flexible platform that is able to adapt their packet-processing components as well as to customize the functionalities of their data, control and management planes. However, these approaches fail to be fast enough to be used in operational context where a wire-speed is needed. Nevertheless, by adding and configuring convenient hardware packet processing resources such as FPGA, CPU cores and memory storage, one can hope to meet the performance requirements. Indeed, flexibility and high performance are in conflict in most situations. Flexibility requires more function-

alities to be implemented in software to maximize the programmability. On other hand, high performance cannot be reached in software and needs custom hardware. A major challenge for PEARL is to allocate enough hardware and multi-cores in order to achieve both flexibility and high performance. Another design goal is related to isolation. By isolation we mean a mechanism that enables different architectures or protocols running in parallel on separate virtual router instances without impacting each other performances. In order to achieve isolation, we should provide a mechanism that will ensure that one instance can only use its allocated hardware (CPU cores and cycles, memory, resources, etc.) and software resources (lookup routing tables, packet queue, etc.) and is forbidden to access resources of other instances even when they are idle. We need also a dispatching component that will ensure that IP or non-IP packets are delivered to specified instances following custom rules defined over MAC layer parameters, protocols, flow label or packet header fields.

The PEARL offers high flexibility through the custom configurations of both hardware data path and software data path. Multiple isolated packet streams and virtualization techniques enable the isolation among virtual router instances, while the fast lookup hardware provides the capacity to achieve high performance.

6.3 Platform design and Implementation

6.3.1 System Overview

PEARL uses commodity multi-core CPU hardware platforms that run generic software as well as specialized packet-processing cards for high performance packet processing as shown in Figure 6.1. The virtualization environment is build using the Linux-based LXC solution [4, 71]. This enables multiple virtual router instances to run in parallel over a CPU core or one router instance over multiple CPU cores. Each virtual machine can be logically viewed as a separate host. The hardware platform

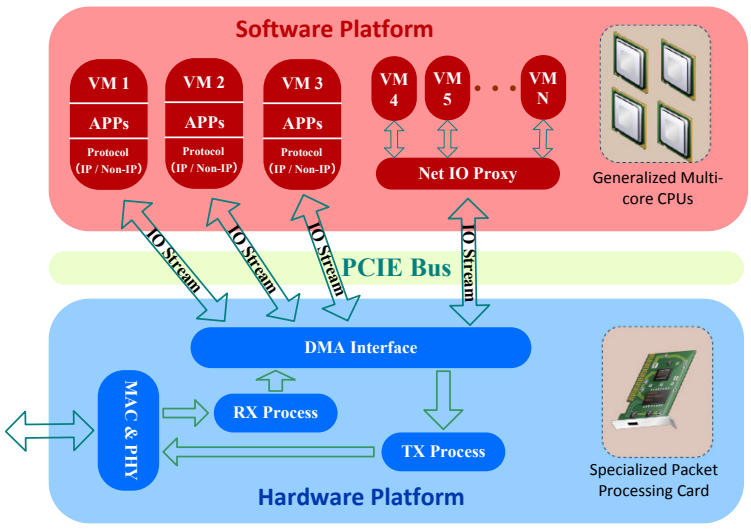


Figure 6.1: Overview of PEARL architecture

contains a FPGA-based packet processing card with embedded TCAM and SRAM. This card enables fast packet processing and strong isolation.

Isolation. PEARL implements multiple simultaneous fast virtual data planes by allocating separate hardware resources to each virtual data plane. This facilitates strong isolation among the hardware virtual data planes. Moreover, LXC takes advantage of a group of the kernel features (namespace, cgroup) to ensure isolation in software between virtual router instances. A multi-stream high-performance DMA engine is also used in PEARL which receives and transmits packets via high-speed PCI Express bus between hardware and software platforms. Each IO stream can be either assigned to a dedicated virtual router or shared by several virtual routers using a net IO proxy. **Flexibility.** We use TAP/TUN device as the network interface in each virtual machine. Each virtual machine could be considered as a standard Linux host containing multiple network ports. Thus, the IPv4, IPv6, OpenFlow, even Non-IP protocol stack can be easily loaded. Adding new functions to router is also convenient though programming Linux applications. For example, to load IPv4

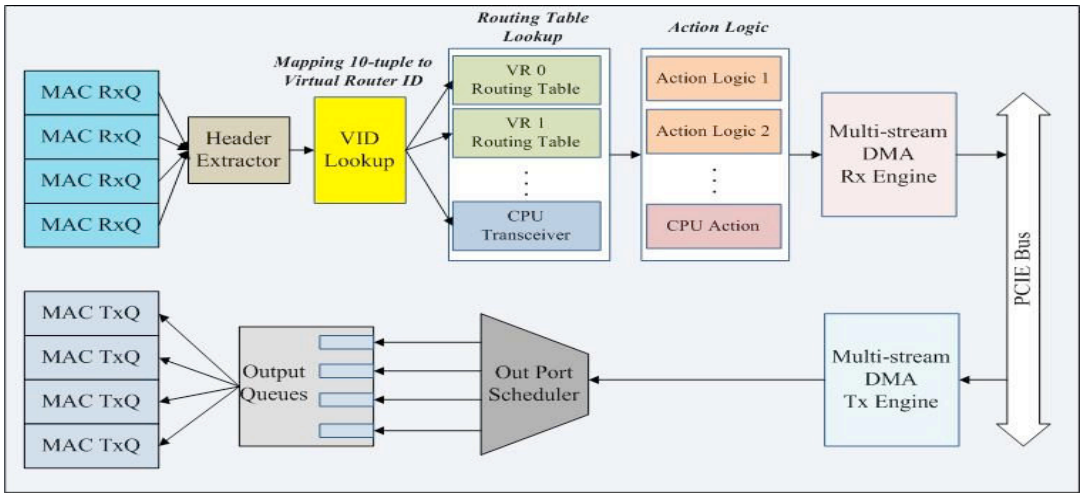


Figure 6.2: PEARL hardware data plane architectural

or IPv6, Quagga routing software suite can be used as the control plane inside each Linux container.

High performance. The operations of routing table lookup and packets dispatch to different virtual machines are always the performance bottleneck. PEARL offloads these two operations into hardware to achieve high speed packet forwarding. In addition, since LXC is a light weight virtualization technique with low overhead, the performance is further improved.

Hardware platform

To provide both high performance and strong isolation in PEARL, we design a specialized packet processing card. Figure 6.2 shows the architecture of hardware data plane. It is a pipeline-based architecture which consists of two main data paths: the transmitting and the receiving data path. The receiving data path is responsible for processing the ingress packets, and the transmitting data path, the egress packets. In what follows, the processing stages of the pipeline are detailed.

Header Extractor. For each incoming packet, one or many fields are extracted from the packet header. These fields are used for virtual router ID (VID) lookup

in the next processing stage. For IP-based protocols, a 10-tuple, defined following OpenFlow, is extracted, while for non-IP protocols, the MAC address is extracted.

VID Lookup. Each virtual router in the platform is marked by a unique VID. This stage classifies the packet based on the fields extracted in the previous stage. For the storage and lookup of the fields, we use a TCAM which can be configured by the users. Due to the special features of TCAM, each field of the rules in VID lookup table can be a wildcard. Hence, PEARL can classify packets of any kind of protocols into different virtual routers as long as they are Ethernet based, such as IPV4, IPv6 and non-IP protocols. The VID lookup table is managed by a software controller which enables users to define the fields as needed. The VID of the virtual router to which a packet belongs is appended on the packet as a custom header.

Routing Table Lookup. In a network virtualization environment, each virtual router should have a distinct routing table. Since there are no standards for non-IP protocols until now, we only consider the storage and lookup of routing tables for IP-based protocols in hardware. It is worth noting that routing tables for non-IP protocols can be accommodated through FPGA in the cards. Given limited hardware resources, we implement four routing tables in the current design. The tables are stored in TCAM as well. We take the VID combined with destination IP address as the search key. The VID part of the key is performing exact matching and the IP part is performing the longest prefix matching in TCAM. Once a packet matches in the hardware, it will be sent to the kernel for further processing, greatly improving the packet forwarding performance. For non-IP protocols or the IP-based protocols that are not accommodated in the hardware, we integrate a CPU transceiver module. The module is responsible for transmitting the packet to the CPU directly without looking up routing tables. Whether a packet should be transmitted by the CPU transceiver module or not is completely determined by the software. With the flexibility offered

by the CPU transceiver module, it is easy to integrate more flexible software virtual data planes into PEARL.

Action Logic. The result of routing table lookup is the next hop information which is stored in a SRAM-based table, including output card number, output port number, the destination MAC address and so on. It defines how to process the packet, so it can be considered as an action associated with each entry of the routing tables. Based on the next hop information, this stage performs some decisions such as forwarding, dropping, broadcasting, decrementing TTL and updating MAC address.

Multi-stream DMA Engine. To accelerate the IO performance and greatly exploit the parallel processing power of multi-core processor, we design a multi-stream high-performance DMA engine in PEARL. It can receive packets of different virtual routers from the network card to different memory regions in the host, and transmit packets in the opposite direction via the high-speed PCI Express bus. From a software programmers perspective, there are multiple independent DMA engines, and the packets of different virtual routers are directed into different memory regions, which is convenient and lockless for programming. Meanwhile, we make a tradeoff between flexibility and high performance of DMA transfer mechanism, and carefully redesign the DMA engine in FPGA. The DMA engine can transfer packets to the pre-allocated huge static buffer at contiguous memory locations. It greatly decreases the number of memory accesses required to transfer a packet to CPU. Each packet transported between the CPU and the network card is equipped with a custom header which is used for carrying processing information to the destination, such as the matching results.

Output Scheduler. The egress packets sent back by the CPU are scheduled based on their output port number, which is a specific field in the custom header of the packet. Each physical output port is equipped with an output queue. The scheduler puts each packet in the appropriate output queue for transmitting.

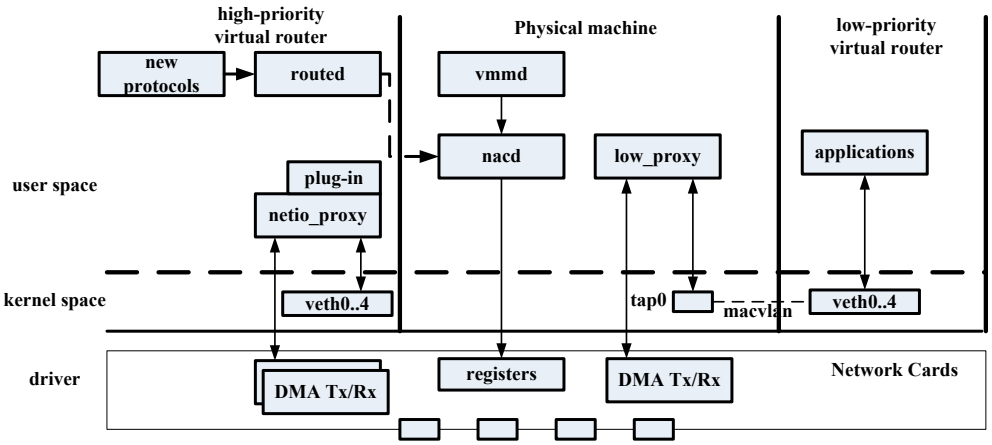


Figure 6.3: Packet Processing Path in software

6.3.2 Software Platform

Our software platform of PEARL consists of several components as shown in Figure 6.3. These include **vmmd** to provide the basic management functions for virtual routers and packet processing cards; **nacd**, to offer a uniform interface to the underlying processing cards outside the virtual environment; **routed** to translate the forwarding rules generated by the kernel or user applications into a uniform format in each virtual router, and install these rules into the TCAM of the processing cards; **netio_proxy** to transmit the packets between the physical interfaces and virtual interfaces, and **low_proxy** to dispatch packets into low priority virtual routers which share one pair of DMA Rx/Tx buffers. With different configuration and combination of these programs, PEARL can generate different types of virtual routers to achieve flexibility.

There are two types of virtual routers in PEARL: high and low priority virtual routers. Each high-priority virtual router is assigned with one pair of DMA Rx/Tx buffers and an independent TCAM space for lookup. With the high-speed lookup based on TCAM, and efficient IO channels provided by the hardware, the virtual router can achieve the maximum throughput in PEARL platform. For low-priority

virtual routers, all the virtual routers share only one pair of DMA Rx/Tx buffers and they cannot utilize TCAM for lookup. The **macvlan** kernel module is used to dispatch packets between multiple virtual routers. The applications inside the low priority virtual routers can use the socket APIs to capture packets from the virtual interfaces. Each packets needs to go through at least 2 times context switch (system calls) during the transmission to the user application, resulting in a relatively low IO performance.

We take IPv4 and IPv6 as two Internet protocols to show how the virtual routers can be easily implemented on PEARL. **High-priority IPv4 virtual router.** To create a high-priority IPv4 virtual router in our platform, the **vmmd** process first starts a new Linux container with several virtual interfaces, and collects the MAC address of each virtual interface, and installs these addresses in the underlying cards via the **nacd** process so that the hardware can identify the packets heading to this virtual router, and copy the packet into the certain DMA Tx buffer which is assigned to this virtual router. Then, the routed process is launched in the new container. It extracts the routes through the **NETLINK** socket inside the virtual router and installs routes and the forwarding action in hardware, so the hardware can fill a little structure in the memory to notify the **netio_proxy** process when a packet matches a route in TCAM. The **netio_proxy** process delivers the packets either to the virtual interface or directly to the physical interface according to the forwarding action in memory. For example, most of time, normal packets will match a route in the hardware. When the **netio_proxy** receives these packets, it will directly send them through a DMA Tx buffer. An ARP request packet will not match any rules in the TCAM, and the **netio_proxy** process will deliver this packet to the virtual interface, receive the corresponding ARP reply packet from the virtual interface, and then send it to the physical interface.

Low-priority virtual router. To create low priority virtual routers, a tap device is set up by the vmmd process (tap0). Low priority virtual routers are configured to share the network traffic of this tap device through the **macvlan** mechanism. The **low_proxy** process acts like a bridge, transmitting packets between DMA buffers and the tap device in both directions. Noting that the MAC addresses of the virtual interfaces are generated randomly, we can encode the MAC addresses to identify virtual interface where the packet comes from. For example, we can use the last byte of the MAC address to identify the virtual interfaces if the **low_proxy** process receives a packet with source MAC address 02:00:00:00:00:00, it knows that the packet is from the first virtual interface in one of the low priority virtual routers, and transmits the packet to the first physical interface immediately. We adopted this method in the **low_proxy** process and **vmmd** process, and use the second byte to identify the different low priority virtual routers. It not only saves the time consumed by inefficient MAC hash lookup to determine where the packet comes from, but also saves space in TCAM, because all the low priority virtual routes only need one rule in TCAM (02:*:00:00:00:*).

IPv4/IPv6 virtual router with kernel forwarding. In this configuration, the routed process does not extract the route from the kernel routing table; instead, it enables the *ip_forward* options of the kernel. As a result, all packets will match the default route in TCAM without the forwarding action. The **netio_proxy** process transmits all these packets into the virtual interfaces, so that the kernel will forward the packet instead of the underlying hardware. The tap/tun device is used as the virtual interface. Since the **netio_proxy** is a user space process, each packet needs two system calls to complete the whole forwarding.

User-defined virtual router. User-defined packet process process can be implemented as a plug-in loaded by the **netio_proxy** process, which makes the PEARL extensible. We opened the basic packet APIs to the users, such as *read_packet()*,

send_packet(). Users can write their own process module in C Language, and runs it in the independent virtual routers. For the light-weight applications which do not need to deal with huge amount of network traffic, users can also write a socket program in either high or low priority virtual routers.

6.3.3 Evaluation and Discussion

We implemented PEARL prototype using a common server with our specialized network card. The common server is equipped with an Xeon 2.5GHz 64-bit CPU and 16G DDR2 RAM. The OS-level virtualization techniques Linux Containers (LXC) is used to isolate the different virtual routers (VR).

In order to demonstrate the performance and flexibility of PEARL, our implementation is evaluated in three different configurations: high performance IPv4 virtual router, kernel forwarding IPv4/IPv6 virtual router, and IPv4 forwarding in low priority virtual router.

We conducted 4 independent sub-networks with SPRIENT TestCenter to measure the performance of the three configurations in 1-4 VRs. Three different lengths of packet (64, 512 and 1518) are used (for IPv6 packet, the packet length is 78, 512 and 1518. 78 bytes is the minimal IPv6 packet length supported by TestCenter).

Figure 6.4 shows the throughputs of an increasing numbers of VRs using Configuration 1. Each virtual data plane has been assigned with a pair of DMA RX/TX buffers and the independent routing table space in the TCAM, resulting in an efficient IO performance and a high speed IP lookup. The result shows that when the number of the VR reaches to 2, the throughput of minimal IPv4 packet of PEARL is up to 4Gbps, the maximum theoretical speed of our implementation.

Figure 6.5 illustrates the throughputs of an increasing number of VRs using Configuration 2. Each virtual data plane has the same configuration as Configuration 1, except that no virtual data plane has its own routing space in TCAM. In Figure 5,

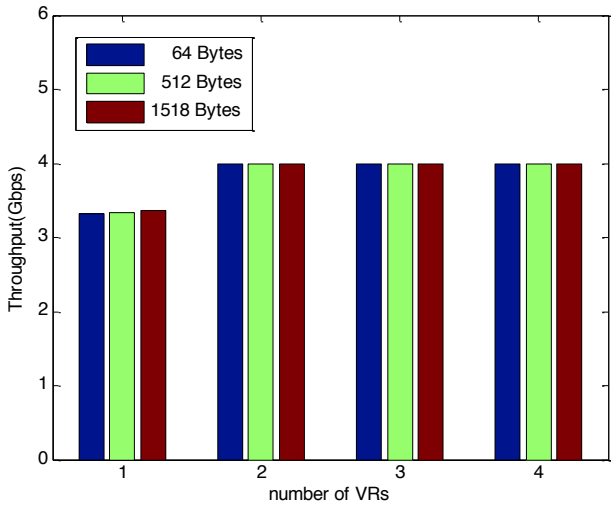


Figure 6.4: Throughput of high performance IPv4 virtual routers

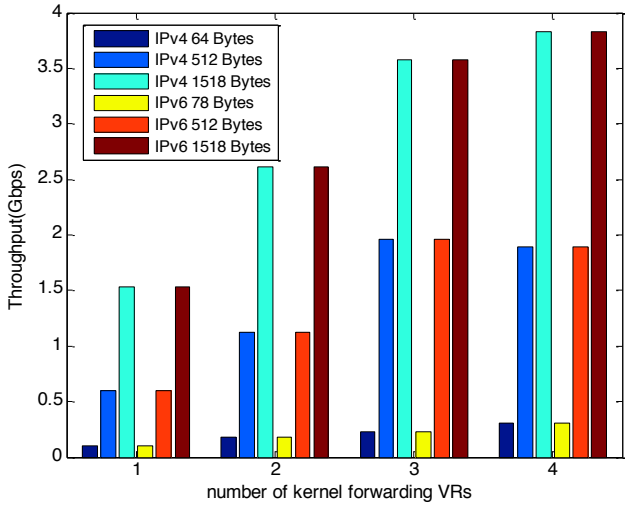


Figure 6.5: Throughput of high performance IPv4 virtual routers

the throughput of minimal IPv4/IPv6 packet forwarding is only 100Mbps when there is only one VR. It is because we used the original kernel for packet forwarding *i.e.* each packet needs to go through 2 times context switch and 3 times memory copy in our design. We can optimize this by re-implementing the forwarding functions as a plug-in in the **netio_proxy** process in VR.

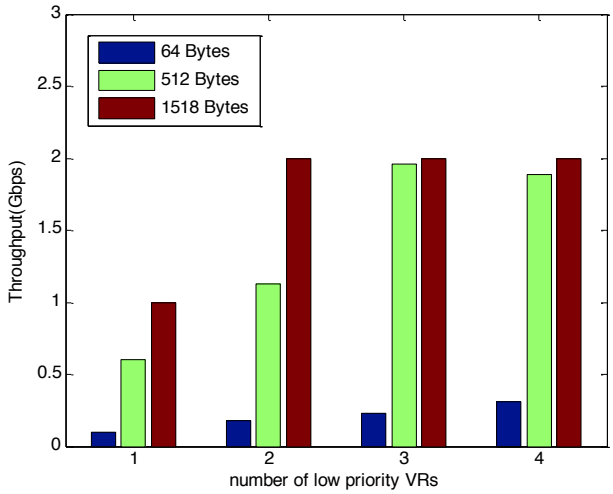


Figure 6.6: Throughput of high performance IPv4 virtual routers

Figure 6.6 shows the throughputs of an increasing numbers of low priority VRs using Configuration 3. Low priority VRs share only one pair of DMA TX/RX IO channel and cannot take advantage of TCAM to speed up the IP Lookup. It can be used to verify the applications which handle little traffic (new routing protocols etc).

We can see from the results that the total throughputs of the minimal IPv4 packet remain 60Mbps as the number of VR increases. The **macvlan** module used for the sharing of the network traffic between multiple VRs results in a long kernel path for the packet processing, so the total performance is even lower than the IPv4 kernel forwarding in the Configuration 2. We can improve the performance by developing a new mechanism that suit for our case.

6.4 Related Work

Recent researches, such as vRouter Project [19], RouteBricks [16] and PacketShader [24], have exploited the tremendous power of modern multi-core CPUs and multi-queue network interface cards (NICs) in building high-performance software routers on commodity hardware. However, the commodity hardware is not good

enough for such applications. They pointed out that memory latency or IO performance becomes the bottleneck for small packets in such platforms. Due to the complexity of DMA transfer mechanism in commodity NICs, the performance of high speed PCI Express bus has not been fully exploited. Meanwhile, the traditional DMA transfer and routing table lookup result in multiple memory accesses, limiting the forwarding performance for small packets. To address both the IO and memory access bottleneck in PEARL, we make a balance between the flexibility of commodity hardware and the high performance of FPGA-based specialized hardware. We simplify the DMA transfer mechanism and redesign the DMA engine to accelerate IO performance, and offload the routing table lookup operation to the hardware platform. OpenFlow [38] enables rapid innovation of various new protocols, and divides the function into control plane and data plane. PEARL has the similar idea to divide the function between software and hardware. However, PEARL does host multiple virtual data plane in hardware itself, which could offer both strong isolation and high performance, while OpenFlow does not. Another work, SwitchBlade [1] has presented a modular architecture of virtualized data plane in FPGA-based hardware platform. However, PEARL makes some important improvements and has two unique features in hardware data plane. First PEARL dispatches packets into different virtual routers based on the 10-tuple, but SwitchBlade only classifies packets based on MAC addresses. With the flexibility of the 10-tuple and wildcard, PEARL has the capability to classify packets based on the protocol of any layer. For example, PEARL can specify the packets of a critical application with a dedicated fast path based on its TCP flow information, which makes it easy to guarantee QoS for critical applications. In this case, it resembles a predefined virtual link in a circuited-switch network. Second, all the packets, even those matched in hardware, are transmitted to the CPU in PEARL. This is especially beneficial for scalability in a single physical

machine with more FPGA cards. However, it is not clear how to forward packets between different ports in different cards in SwitchBlade architecture.

6.5 Conclusion

We aim at using flexible routers to bridge the gap between new Internet protocols and the practical test deployment. To this end, this work presents a programmable virtual router platform, PEARL. The platform allows users to easily implement new Internet protocols and run multiple isolated virtual data planes concurrently. A PEARL router consists of a hardware data plane and a software data plane with DMA engines for packet transmission. The hardware data plane is built on top of a FPGA based packet processing card with TCAM embedded. The card facilitates fast packet processing and IO virtualization. The software plane is built by a number of modular components and provides easy programmable interfaces. We have implemented and evaluated the virtual routers running on PEARL.

Our future work includes designing a hybrid algorithm which can take advantage of TCAMs to improve the performance of packet classification on PEARL.

Chapter 7

Conclusion

Packet classification is a core concern for network services. With the development of cloud computing and Software Defined Networking, efficient software-based packet classification has become a new research interest.

In this thesis, we have reviewed typical packet classification algorithms, such as IP lookup algorithms and multi-dimensional packet classification algorithms. We find that most algorithms exploit the "sparseness" feature to achieve fast algorithmic packet classification. In IP lookup algorithms, different algorithms exploit the different ruleset features, such as non-uniform prefix length distribution, and the binary tree built from the FIB table has a lot of single child nodes (the sparseness in the shape of the binary tree), *etc.* Similarly, in multi-dimensional packet classification algorithms, the rulesets exhibit some features such as few prefixes combinations, non-uniform range distribution. Different from IP lookup which has a bounded lookup time complexity, the performance of multi-dimensional packet classification algorithms rely heavily on the ruleset features. Therefore when evaluating typical packet classification algorithms on different rulesets, we find a wide variation of performance results.

In order to understand the root cause of high performance variation, we have studied the impact of both algorithm design and ruleset features on the overall performance. In the algorithm design, we have reviewed four typical packet classification algorithms, and presented an algorithm design framework. In this algorithm framework, we view each decision-tree based algorithm as a combination of three types of meta methods, and present two new variants – HyperSplit-op and HiCuts-op by mixing the meta methods from different algorithms. Through experiments, we find that the memory footprint of these two variants is $1 \sim 2$ orders of magnitudes smaller than those of the previous ones, and their classification speed is $1\times$ faster. On the low locality traffic, the two variant algorithms achieve $4 \sim 9$ Gbps throughput. More importantly, these results reveal that part of the reason of the wide performance variation is due to the use of unsophisticated field choosing methods or a lack of necessary optimization tricks. These observations provide a solid foundation for the study of the connection between ruleset features and the performance of different algorithms.

In our research, we find that the ruleset feature usually determines the final performance of majority algorithms. Our research results show that the "coverage-uniformity" of the rulesets determines the number of memory accesses, while the "orthogonal structure" inside the rulesets, the memory footprint of different algorithms. For "coverage-uniformity", we present a method capable of quantifying this uniformity and choosing the right algorithm. We also propose a memory footprint model based on the feature of "orthogonal structure" which can roughly estimate the memory footprint.

The memory footprint model can be used to estimate the memory footprint of large rulesets (100K) in seconds. And the quantify method can be used to reveal the "coverage-uniformity" of the rulesets. These features are powerful to guide the design of efficient packet classification algorithms.

Then we design the SmartSplit multi-decision tree algorithm and AutoPC framework based on the analysis of these two features. Compared to the state-of-art algorithms, the memory accesses of SmartSplit are reduced by one time, and the memory footprint is reduced by up to 10 times. For a given ruleset, the AutoPC framework is capable of choosing the "right" algorithm for the ruleset. Compared to using only one algorithm, the lookup speed is increased by 3.8 times.

Besides studying the feature of multi-dimensional rulesets, we have also studied the connection between prefix length and update cost in IP lookup. We observe that the number of memory accesses is linear with the prefix length in Tree Bitmap; the update cost is small if the prefix length is short in DIR-24-8 algorithm. Based on this observation, we propose a hybrid algorithm SplitLookup. SplitLookup achieves a lookup speed close to DIR-24-8 while its update cost is 2 orders of magnitude smaller than DIR-24-8. On the Tilera many-core chip, SplitLookup can achieve 40Gbps of 64B packets.

At last, we design PEARL, a flexible and easy-to-program platform for network applications. In PEARL, the whole packet processing can be reprogrammed, and different network applications can be easily isolated using LXC containers. Our experiment shows that while flexible, PEARL can also provide 4×1 Gbps packet forwarding of 64B packets.

Packet classification has been extensively studied in the past decades. However, the challenges remain as the needs of flexible network architecture supporting more and more new network applications. This thesis reviews a lot of typical packet classification algorithms, including IP lookup algorithm and multi-dimensional packet classification algorithms, and conclude that particular ruleset features usually determine the performance of many existing algorithms. The thesis has proposed several methods for analyzing the ruleset features and guiding the design of new algorithms or the choice of existing algorithms. The methods and algorithms proposed in this

thesis have been extensively evaluated on software-based platform. experiments show that our approach is effective and efficient. Future work includes exploring the rule-set features which are related to the TCAM based packet classification solutions, and designing hybrid algorithms for TCAM and algorithm based packet classification, *etc.*

Bibliography

[1] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. Switchblade: a platform for rapid deployment of network protocols on programmable hardware. *ACM SIGCOMM Computer Communication Review*, 40(4):183–194, 2010.

[2] F. Baboescu, D.M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 123–133. IEEE, 2005.

[3] F. Baboescu and G. Varghese. Scalable packet classification. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 199–210. ACM, 2001.

[4] Sapan Bhatia, Murtaza Motiwala, Wolfgang Muhlbauer, Yogesh Mundada, Vytautas Valancius, Andy Bavier, Nick Feamster, Larry Peterson, and Jennifer Rexford. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, page 72. ACM, 2008.

[5] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve ip lookups. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1454–1463. IEEE, 2001.

[7] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM Computer Communication Review*, 37(4):1–12, 2007.

[8] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: a retrospective on evolving sdn. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 85–90. ACM, 2012.

[9] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

[10] Bernard Chazelle. Lower bounds for orthogonal range searching: part ii. the arithmetic model. *Journal of the ACM (JACM)*, 37(3):439–463, 1990.

[11] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 333–346, 2001.

[12] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[13] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. *Small forwarding tables for fast routing lookups*, volume 27. ACM, 1997.

[14] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E Taylor. Longest prefix matching using bloom filters. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212. ACM, 2003.

[15] Sarang Dharmapurikar, Haoyu Song, Jonathan Turner, and John Lockwood. Fast packet classification using bloom filters. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 61–70. IEEE, 2006.

[16] M. Dobrescu, N. Egi, K. Argyraki, B.G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *ACM SOSP*, volume 9. Citeseer, 2009.

[17] Richard P Draves, Christopher King, Srinivasan Venkatachary, and Brian D Zill. Constructing optimal ip routing tables. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 88–97. IEEE, 1999.

[18] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software ip lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[19] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, page 20. ACM, 2008.

[20] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang. Parasplit: A scalable architecture on fpga for terabit packet classification. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, pages 1–8. IEEE, 2012.

[21] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1240–1247. IEEE, 1998.

[22] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*, pages 34–41, 1999.

[23] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 147–160. ACM, 1999.

[24] S. Han, K. Jang, K.S. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 195–206. ACM, 2010.

[25] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: An open platform for network research. *ACM SIGCOMM Computer Communication Review*, 33(1):53–57, 2003.

[26] Guy Joseph Jacobson. Succinct static data structures. *Phd thesis*, 1988.

[27] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[28] W. Jiang and V.K. Prasanna. A memory-balanced linear pipeline architecture for trie-based ip lookup. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*, pages 83–90. IEEE, 2007.

[29] Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 51–62. ACM, 2008.

[30] Paul Hick kc claffy, Dan Andersen. The caida anonymized 2011 internet traces 20110217. http://www.caida.org/data/passive/passive_2011_dataset.xml.

[31] TV Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 203–214. ACM, 1998.

[32] Yanbiao Li, Dafang Zhang, Alex X Liu, and Jintao Zheng. Gamt: a fast and scalable ip lookup engine for gpu-based software routers. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 1–12. IEEE Press, 2013.

[33] Alex X Liu, Chad R Meiners, and Yun Zhou. All-match based complete redundancy removal for packet classifiers in tcams. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 111–115. IEEE, 2008.

[34] Alex X Liu, Eric Torng, and Chad R Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.

[35] L. Luo, G. Xie, Y. Xie, L. Mathy, and K. Salamatian. A hybrid ip lookup architecture with fast updates. In *to appear in INFOCOMM'12.* IEEE, 2012.

[36] Y. Ma and S. Banerjee. A smart pre-classifier to reduce power consumption of tcams for multi-dimensional packet classification. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 335–346. ACM, 2012.

[37] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging parallelism for multi-dimensional packetclassification on software routers. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 227–238. ACM, 2010.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[39] Chad R Meiners, Alex X Liu, and Eric Torng. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. In *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, pages 266–275. IEEE, 2007.

[40] Chad R Meiners, Alex X Liu, and Eric Torng. Topological transformation approaches to optimizing tcam-based packet classification systems. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 73–84. ACM, 2009.

[41] Chad R Meiners, Alex X Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in tcams. *IEEE/ACM Transactions on Networking (ToN)*, 20(2):488–500, 2012.

[42] Donald R Morrison. Patriciapractical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[43] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. vcrib: virtualized rule management in the cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 23–23. USENIX Association, 2012.

[44] Stefan Nilsson and Gunnar Karlsson. Ip-address lookup using lc-tries. *Selected Areas in Communications, IEEE Journal on*, 17(6):1083–1092, 1999.

[45] He Peng, Guan Hongtao, Laurent Mathy, Kavé Salamatian, and Xie Gaogang. Toward predictable performance in decision tree based packet classification algorithms. In *The 19th IEEE LANMAN Workshop.* IEEE, 2013.

[46] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[47] Lucian Popa, Norbert Egi, Sylvia Ratnasamy, and Ion Stoica. Building extensible networks with rule-based forwarding. In *OSDI*, pages 379–392, 2010.

[48] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *INFOCOM 2009, IEEE*, pages 648–656. IEEE, 2009.

[49] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 9–9. USENIX Association, 2012.

[50] Luigi Rizzo and Matteo Landi. netmap: memory mapped access to network devices. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 422–423. ACM, 2011.

[51] Miguel Á Ruiz-Sánchez, Ernst W Biersack, and Walid Dabbous. Survey and taxonomy of ip address lookup algorithms. *Network, IEEE*, 15(2):8–23, 2001.

[52] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224. ACM, 2003.

[53] Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, volume 1991, pages 93–99, 1991.

[54] H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster ip route lookup. In *ICNP 2005. 13th IEEE International Conference on Network Protocols, 2005.*, pages 10–pp. IEEE, 2005.

[55] H. Song and J.S. Turner. Abc: Adaptive binary cuttings for multidimensional packet classification. *IEEE/ACM TRANSACTIONS ON NETWORKING*, 2012.

[56] Haoyu Song, Fang Hao, Murali Kodialam, and TV Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *INFOCOM 2009, IEEE*, pages 2518–2526. IEEE, 2009.

[57] Song.H. Design and evaluation of packet classification systems. http://www.arl.wustl.edu/∼hs1/publication/thesis-main.pdf.

[58] Ed Spitznagel, David Taylor, and Jonathan Turner. Packet classification using extended tcams. In *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, pages 120–131. IEEE, 2003.

[59] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 135–146. ACM, 1999.

[60] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999.

[61] V Srinivasan, G Varghese, S Suri, and M Waldvogel. Fast scalable algorithms for level four switching. In *Proceedings of ACM SIGCOMM98*, pages 191–202, 1998.

[62] David E Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[63] D.E. Taylor and J.S. Turner. Classbench: A packet classification benchmark. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2068–2079. IEEE, 2005.

[64] The amazon ec2 outage no one noticed. http://www.praxicom.com/2008/04/the-amazon-ec2.html.

[65] Aryaka wan optimization. http://www.aryaka.com.

[66] Cisco cloud firewall. http://blogs.cisco.com/datacenter/announcing-the-cisco-asa-1000v-cloud-firewall/.

[67] Embrance. http://www.embrance.com.

[68] Evaluation of packet classification algorithms. http://www.arl.wustl.edu/∼hs1/PClassEval.html.

[69] Hypersplit source code. http://security.riit.tsinghua.edu.cn/share/index.html.

[70] Intel data plane development kit. http://dpdk.org.

[71] Linux container. `https://linuxcontainers.org`.

[72] Prefixes update. `http://bgpupdates.potaroo.net/instability/bgpupd.html`.

[73] Routeviews. `http://www.routeviews.org`.

[74] Tilepro64 power consumption. http://tilera.com/products/processors/TILEPRO64.

[75] Why nicira abandoned openflow hardware control. `http://searchnetworking.techtarget.com/news/2240174517/Why-Nicira-abandoned-OpenFlow-hardware-control`.

[76] Zscaler cloud security. http://www.zscaler.com.

[77] B. Vamanan, G. Voskuilen, and TN Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 207–218. ACM, 2010.

[78] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. *Scalable high speed IP routing lookups*, volume 27. ACM, 1997.

[79] Yi Wang, Yuan Zu, Ting Zhang, Kunyang Peng, Qunfeng Dong, Bin Liu, Wei Meng, Huicheng Dai, Xin Tian, Zhonghu Xu, et al. Wire speed name lookup: A gpu-based approach. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 199–212. USENIX, 2013.

[80] T.Y.C. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1213–1222. IEEE, 2000.

[81] Gaogang Xie, Peng He, Hongtao Guan, Zhenyu Li, Layong Luo, Jianhua Zhang, Yonggong Wang, and K Salamatian. Pearl: a programmable virtual router platform. *Communications Magazine, IEEE*, 49(7):71–77, 2011.

[82] Francis Zane, Girija Narlikar, and Anindya Basu. Coolcams: Power-efficient tcams for forwarding engines. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 42–52. IEEE, 2003.

[83] Marko Zec, Luigi Rizzo, and Miljenko Mikuc. Dxr: towards a billion routing lookups per second in software. *ACM SIGCOMM Computer Communication Review*, 42(5):29–36, 2012.

[84] Jin Zhao, Xinya Zhang, Xin Wang, and Xiangyang Xue. Achieving o (1) ip lookup on gpu-based software routers. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 429–430. ACM, 2010.

[85] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.