A Scalable Algorithm for Packet Classification

by

Ruoyuan Zhang

A thesis submitted to the

Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Masters of Applied Science

at the

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

May 22, 2003

©Ruoyuan Zhang, 2002

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-83488-3

Our file Notre référence

ISBN: 0-612-83488-3

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

The undersigned recommend the Faculty of Graduate Studies and Research the acceptance of the thesis, "A Hierarchical Packet Classification Scheme", submitted by Ruoyuan Zhang, B.Eng., in partial fulfillment of the requirements for the degree of Master of Applied Science.

A handwritten signature in black ink, appearing to read "D. Paul", written over a horizontal dashed line.

Thesis Supervisor

A handwritten signature in black ink, appearing to read "S. Paul", written over a horizontal dashed line.

Chairman, Department of Systems and Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Abstract

In order to support the more flexible Internet applications, such as firewalls, differentiated services, QoS routing etc., the routers are expected to have the ability to differentiate the traffic, which is called *Packet Classification*. Different traffics are termed as different flows. A flow, defined by a rule, is a set of packets having the same packet header characteristics. A classifier has many rules and defines what kinds of flows the router should differentiate. With the development of the Internet, fast classification algorithms are required in the routers. The existing algorithms in the literature have poor scalability performances in either memory storage or searching time, as the sizes of classifiers are growing. Especially in the Bit Vector (BV) algorithm[1], both the storage and searching speed are not scalable. And in Aggregate Bit Vector (ABV) scheme[5] the searching speed is much improved, but the storage is still same as the BV algorithm. In this thesis, based on some observations of the real classifiers and the above two schemes, we propose a packet classification scheme called Hierarchical Bit Vector (HBV). By using the multi-dimensional hierarchical architecture and ruleset pre-grouping, the HBV scheme is scalable in both storage and query time. We test the scheme with synthetic 2-dimensional classifiers and show that in the HBV, while the query speed is slightly better than the ABV, the required storage is one or two orders of magnitudes smaller than the BV algorithm.

Acknowledgements

I would like to thank the following people who helped me to complete the thesis. Without their advice, time, encouragement and love, this thesis would not be possible.

First of all, I want to thank my thesis supervisor Professor Ioannis Lambadaris. Thank you for your support that helps me a lot, for your patience, for your advices, ideas and discussions that inspire me all the time.

I am thankful to my colleagues in the Broadband Network Lab, including Ashraf Mawlawy, Yunhao Li, Fengjie Yuan, Mark Francisco, Zhenjiang Han, Wei Yi, and others. Thank you all for your encouragements and precious suggestions. Especially I thank Jacky Tuinstra, who spent a lot time proofreading my thesis.

I am also deeply grateful to my parents in China. Thank you for your encouragement and concern that help me to get through all the difficulties.

I thank my friend Sue Bao who invigorated me when I felt much pressure. As well, I thank Jean Wang, Hong Lu, Jun Yan, Wenyan Hong, Jenny Zhu and other friends in Ottawa, China and United States. Thank you all for your concern and all sorts of help.

And I must thank granny Chen, who gave me the feeling of sweet comfortable home in this city far from my home-town.

Contents

1	Introduction	1
1.1	Differentiating the Traffic	1
1.2	Requirements of Classification Algorithms	4
1.3	Objectives and Contributions	6
1.4	Organization of the Thesis	8
2	Problem Statement	9
2.1	Packets and Rules	9
2.2	Ruleset and Best Matching	10
2.3	Packet Classification Problem	11
3	Review of the Packet Classification Algorithms	13
3.1	Linear Search	13
3.2	TCAM	14
3.3	Bit Vector Intersection	16
3.4	Cross-producting	18

3.5	Hierarchical Tries	19
3.5.1	Basic Trie of Tries	20
3.5.2	Set Pruning Tries	22
3.5.3	Grid of Tries	23
3.6	Segment Tree	25
3.7	Area-based Quadtree	27
3.8	Tuple Space Search	29
3.9	Recursive Flow Classification	30
3.10	Prefix-based Searching	33
3.11	Composite Schemes	34
3.11.1	Hierarchical Intelligent Cuttings	34
3.12	Summary	36
4	Hierarchical Bit Vector Scheme	37
4.1	Concept	37
4.1.1	Bit Vector Algorithm	38
4.1.2	Aggregate Bit Vector Scheme	41
4.1.2.1	Aggregation	41
4.1.2.2	False Match and Sorting the Rules	43
4.1.2.3	Pros and Cons	45
4.1.3	Hierarchical Bit Vector Scheme	45
4.1.3.1	Node Space	46
4.1.3.2	Partitioning the Space	46

4.1.3.3	Hierarchical Search	50
4.1.3.4	Improvement for the BV Algorithm	52
4.1.3.5	Rules Splitting	57
4.2	Algorithm	60
4.2.1	Partition Algorithm	60
4.2.1.1	Cut Operation	60
4.2.1.2	Partition Metrics	62
4.2.1.3	Algorithm	64
4.2.2	Search Algorithm	66
4.2.3	Pre-group	68
4.3	Evaluation	71
4.3.1	Analysis of HBV	71
4.3.1.1	Storage	71
4.3.1.2	Access Time	74
4.3.2	Experiments	74
4.3.2.1	Different Wildcards Percentages	75
4.3.2.2	Different Partition Metrics	80
4.3.2.3	Different Vector Lengths	81
5	Conclusions	84
A	Measurements of Partitions of Synthetic Classifiers	90

List of Figures

1.1	An example of applications of packet classification.	2
2.1	Relations of rules: (a) irrelative; (b) overlap; (c) fully covered	11
3.1	Classification with TCAM	15
3.2	Bit Vector intersection algorithm.	17
3.3	Basic trie of tries.	21
3.4	Set pruning tries.	23
3.5	Grid of tries.	24
3.6	Segment tree.	25
3.7	Splitting the space recursively.	27
3.8	A quadtree with CFS for the example classifier. The rules in brace are the CFS.	29
3.9	Tuple space search for 2-dimensional classification.	30
3.10	The mapping of packet headers to flow IDs (assuming the length of packet header is 128 and the length of flow ID is 12).	31
3.11	The data flow in RFC algorithm.	32

3.12	Prefix-based searching with the parallel architecture.	33
3.13	One of the possible data structure of HiCuts for the example classifier. . . .	35
4.1	Two tries for each of the fields in the simple example of Table 3.1. The long bit vectors are for BV algorithm, and the short vectors are for ABV algorithm with aggregate size 4.	39
4.2	The aggregation of long bit vectors. (the aggregate size is 8)	42
4.3	Ranges and rectangles represented by the trie nodes.	47
4.4	Rectangles of rules in our simple example.	47
4.5	Some nodes spaces in the destination trie of the example classifier in Table 3.1.	48
4.6	hierarchical partition of rule space	51
4.7	One partition of the space of the simple example classifier, the first level classifier of the auxiliary-rules and the corresponding searching tries. Only two levels of partition is used in this case. In each auxiliary-rule there are no more than 4 real rules. AR_i represents the auxiliary-rule and the sequence of the bit vectors is $AR_0-AR_1-AR_2$	53
4.8	Level 2 classifiers of all the subspaces and the corresponding searching tries.	55
4.9	Another partition of the space of the simple example classifier, the first level classifier of the auxiliary-rules and the corresponding searching tries. Still there are only two levels of partition.	58

4.10 Level 2 classifiers of all the subspaces and the corresponding searching tries.	59
4.11 Hierarchical partition algorithm to partition the space and ruleset into mul- tiple subspaces so that the size of the ruleset of each subspace is less than N_0	65
4.12 The algorithm of searching best matching rule in HBV scheme.	67
4.13 The wildcards cause serious duplications of rules in the subspaces. Each rule appears twice in the subspaces.	69
4.14 Group the ruleset into two subsets according to the wildcards.	70
4.15 The pre-group algorithm	70
4.16 Hierarchical classifiers.	71
4.17 Storage of BV and HBV scheme with different percentages of wildcards, without the pre-group and the vector length is 64-bit. The HBV scheme outperforms BV scheme by a factor of one or two magnitudes.	76
4.18 λ s for different classifiers without the pre-group. The length of bit vectors is 64.	77
4.19 Storage of BV and HBV scheme with different percentages of wildcards. The pre-group is used and vector length is 64-bit.	79
4.20 Storage of BV and HBV scheme with different cutting metrics. The length of bit vectors is 64 and there are 30% wildcards. The pre-group is not used.	80
4.21 Storage of BV and HBV scheme with different cutting metrics. The length of bit vector is 64 and there are 0% wildcards.	81

4.22	Storage of BV and HBV scheme with different length vectors and 0% wild-cards.	82
4.23	Storage of BV and HBV scheme with different length vectors and 30% wildcards.	83

List of Tables

1.1	An example classifier at router A of Figure 1.1.	3
3.1	A simple example classifier of two fields.	14
3.2	Projection tables on each dimension.	18
3.3	Cross-producting table.	19
3.4	Storage and query time complexities of some algorithms.	36
4.1	Classifier with false matches.	44
4.2	Sorted classifier with zero false match.	45
4.3	Worst case access times of ABV and HBV schemes. ABV0 and HBV0 denote the ABV and HBV algorithms with 0% wildcards, respectively. Here the bit vector length in HBV is 64 and the aggregation size in ABV is also 64. And the memory word is 64-bit wide. For the classifiers of 1K, 2K and 5K, ABV of 2 levels is used; and for classifiers of 10K and 20K ABV of three levels is used.	78

4.4	Worst case access times of ABV and HBV schemes with the pre-group. ABV0 and HBV0 denote the ABV and HBV algorithms with 0% wildcards, respectively. Here the bit vector length in HBV is 64 and the aggregation size in ABV is also 64. And the memory word is 64-bit wide.	79
4.5	Query time of HBV for 0% wildcards with different vector lengths. No pre-group is used.	83
4.6	Query time of HBV for 30% wildcards with different vector lengths. No pre-group is used.	83
A.1	m_l of partitioned sub-classifiers. There is no pre-group and no wildcards. The bit vector length is 64.	90
A.2	α_l of partitioned sub-classifiers. There is no pre-group and no wildcards. The bit vector length is 64.	91
A.3	m_l of partitioned sub-classifiers. There is no pre-group and 10% wildcards. The bit vector length if 64.	91
A.4	α_l of partitioned sub-classifiers. There is no pre-group and 10% wildcards. The bit vector length if 64.	91
A.5	m_l of partitioned sub-classifiers. There is no pre-group and 20% wildcards. The bit vector length if 64.	91
A.6	α_l of partitioned sub-classifiers. There is no pre-group and 20% wildcards. The bit vector length if 64.	92
A.7	m_l of partitioned sub-classifiers. There is no pre-group and 30% wildcards. The bit vector length if 64.	92

A.8	α_l of partitioned sub-classifiers. There is no pre-group and 30% wildcards.	
	The bit vector length is 64.	92
A.9	m_l of partitioned sub-classifiers. There is no pre-group and 40% wildcards.	
	The bit vector length is 64.	92
A.10	α_l of partitioned sub-classifiers. There is no pre-group and 40% wildcards.	
	The bit vector length is 64.	93
A.11	m_l of partitioned sub-classifiers. There is no pre-group and 50% wildcards.	
	The bit vector length is 64.	93
A.12	m_l of partitioned sub-classifiers. There is no pre-group and 50% wildcards.	
	The bit vector length is 64.	93
A.13	m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 16.	94
A.14	α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 16.	94
A.15	m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 32.	95
A.16	α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 32.	95
A.17	m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 128.	95
A.18	α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 128.	95

A.19	m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 256.	96
A.20	α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards.	
	The bit vector length is 256.	96

List of Acronyms and Symbols

Acronyms

ABV — Aggregate Bit Vector

BV — Bit Vector

HBV — Hierarchical Bit Vector

HiCut — Hierarchical Intelligent Cuttings

IP — Internet Protocol

RFC — Recursive Flow Classification

TCAM — Ternary Content Addressable Memory

TCP — Transmission Control Protocol

Symbols

p — a packet

$p^{(j)}$ — the j th field of packet p

$\Re$ — the ruleset of a classifier

R or R_i — a rule

$R^{(j)}$ — the j th field of rule R

N — total number of rules in the ruleset

d — dimensions of the classifier

W — the length of one field of rules

w — the width of memory data bus

N_0 — the maximum number of rules in each sub-space in the HBV scheme

m_l — the number of rules in the hierarchical level l in the HBV scheme

α_l — the duplication ratio in the hierarchical level l in the HBV scheme

L — the total number of levels in the HBV scheme

Chapter 1

Introduction

1.1 Differentiating the Traffic

Currently most of the routers in the Internet only provide the best-effort service, in which all the packets are forwarded based on the destination address field. The destination address of the incoming packet is looked up in a table in the router and the packet is sent to the output link corresponding to the search result. However, with the development of the Internet, the routers are demanded to provide more and more new services on some subset of packets, such as QoS routing, packet filtering, Virtual Private Networks (VPN), traffic billing, service level agreement (SLA), etc. All of these functions require the routers to differentiate the traffic not only on the destination address but on other header fields. Figure 1.1 illustrates some of these examples that can be used in Internet service provider. Subnetworks CN1 and CN2 belong to one corporation and are in one VPN. CN3 is a subnet of CN1. The traffic between CN1 and CN2 have some reserved bandwidth. And there

are voice traffics from subnetwork X flowing into CN1 through router A and should have higher priority than the normal email data. The subnetwork CN3 is more sensitive and only the users from CN2 have privileges to access the resources in network CN3, and other traffic will be refused and dropped.

Figure 1.1: An example of applications of packet classification.

This traffic differentiation is called *packet classification* or *Layer 4 Switching* [7], whose name comes from the fact that the decision of forwarding and other actions is also made based on layer 4 information. The packets are classified to different *flows* according to one or more fields in the packet header. If only one field is used it is called 1-dimensional classification and if multiple fields are used it is multi-dimensional classification. Regarding IP packets, usually five header fields are concerned: source address, destination address, source port number, destination port number and protocol type. The flows are defined by *rules*. In the example of Figure 1.1, the packets from the source of CN2 to the destination of CN3 will be viewed as a flow. Associated with the specification of a flow there is an action which is applied on that flow, such as dropping or passing the packet in the firewall,

adding the packet to some priority queue in QoS routing, etc.

Destination	Source	Traffic Type	Action
CN3	not CN2	*	dropped
CN1	CN2	*	forwarded in reserved bandwidth
CN2	CN1	*	forwarded in reserved bandwidth
CN1	X	voice	higher priority

Table 1.1: An example classifier at router A of Figure 1.1.

The database of a set of rules is called *classifier*, which defines the types of flows that the router should deal with differently. The rules in a classifier could be configured manually or automatically by some software. The header of the incoming packet is looked up in this database to find out the matching rule. If more than one rule matches it, we must have a criteria to select the best one. Normally each rule is assigned a cost or priority and in the case that multiple matching rules are found the best rule should be the one with lowest cost or highest priority. In this thesis we use *best* instead of the lowest cost or highest priority. Table 1.1 shows an example classifier in router A in the Figure 1.1. Notice that the first and fourth rule overlap with each other. Here we arrange the rules in the decreasing priority sequence. If a packet from X is going to CN3, at router A it matches both first and fourth rule and we take the first rule as the best one.

The traditional IP lookup on the destination address is a special case of 1-dimensional packet classification, in which all the packets to one destination prefix is in the same flow and the forwarding to the corresponding output port is the the action. In the case of multiple matching prefixes, the longest one is the best result. So we can take the length of the prefix as the priority of the rule.

It is an issue of which header fields should be used in rules and what should be the

associated actions for some specific applications. However, this thesis only focuses on the general problem: developing an algorithm which can find the best matching rule in a given classifier *efficiently*, both in speed and storage.

1.2 Requirements of Classification Algorithms

Searching Speed

Classification speed can be termed as the number of packets processed per unit time (second). With the development of the Internet, the speeds of the backbone link have increased from OC-48 (2.4Gbits/s) to OC-192 (10Gbits/s) and are marching toward OC-768 (40Gbits/s). The speed of the classification processor should catch up with the wire speed so that it can finish the searching in the time that the minimum size packet takes to arrive on a link. For example, assuming the minimum IP packet length is 68 bytes, in a 10Gbits/s link up to 31.25 million packets arrive in per second. Thus the processor should carried out 31.25 million queries in one second.

Memory Size

Memory is needed to store the rules in some kind of data structure. The minimum size of memory for the rules themselves is $N \bullet d \bullet W$ bits, where N is the number of rules; d is the number of dimensions; W is the length of each field. Moreover the actions are coded and stored with the rules, which cost more storage space. For many algorithms the memory space is larger than the minimum value because of the complex data structure. And if the required memory size is very small, the fast memory technologies, such as static random

access memory (SRAM), could be used on chip to further increase the searching speed. Generally the storage size can be tradeoff-ed with the query time. The query time could speed up with larger storage space.

Dimensions

Applications of packet classification differ on the number of the header fields involved. For the current routing lookup only the destination address is used. For the multicast routing problem the source and destination addresses are needed. More complex applications, such as the firewall and policy routing, may use more fields. Moreover, the algorithms should be scalable in dimensions.

Size of Classifier

Currently typical multi-field applications such as firewalls may have a few hundred or dozens of hundred rules. While it is expected that the size will scale up to dozens of thousands with the emergence of new services and the rise of the throughput of the routers. The classification processor should support the larger classifiers with reasonable storage and fast query speed.

Updating Time

The classifier could be changed by the manager or automatically by the software. If it is changed by the manager, the database is relatively static, which means it will not be modified often. If it is changed automatically, the classifier could be updated frequently and dynamicly. For some applications, dynamic updating of the classifier is needed. For

example, some TCP connections request packet could initialize a TCP flow, which should be added in the classifier quickly. The basic two operations of changing the classifier are deleting and inserting rules incrementally. The classification solutions can not sacrifice too much performance for the updating operations. If each deleting or inserting operation will cause the major part of the whole data structure to be reconstructed, it takes too much time on building the data structure, which is not efficient.

1.3 Objectives and Contributions

The general multi-field classification is a hard problem. There are theoretical bounds for the *general multidimensional searching* problem that require either huge storage or long searching time in the worse case [7]. There are many papers in the literature that propose some algorithms to address the packet classification problem, such as [1] [6] [7] [8] [9] [10] [14] and their references. But some are only suitable for the 2-dimensional classification; some only work when the classifiers are small; some only support the static classifiers, etc. None of them meet all of the requirements of this problem.

Since the general multidimensional searching problem is extremely complex, we can focus on the specific case of packet classification. It is expected that there are some structures in the classifiers to explore. For example, the specifications of the destination and source address fields of the rules can't be any format; they are prefixes in different length. And in [9] [5] the authors investigated some real classifiers and found more characteristics. Thus we can exploit the structure of the rulset to design a classification scheme with good performances, such as small storage or fast query speed.

We present an algorithmic scheme for solving the packet classification problem, which aims at the following goals:

- **Small Storage:** The algorithm should take as little memory as possible. If the algorithm is implemented in hardware, it should be efficient and applicable to fit the small data structure in the on-chip memory.
- **Fast Query Speed:** To catch up with the progress of the speed of the links of Internet, the classification should be fast enough for the current OC-48 and OC-192 links.
- **Supporting Multiple Fields:** The algorithm should allow the rules with multiple fields, from link layer fields to application layer fields.
- **Suitable for Both Hardware and Software:** In order to get high query speed the classification scheme should be capable to be implemented in hardware with reasonable cost. It also should be suitable to be implemented in software to get more flexibility.
- **Scalable:** The storage space and query time should have good performance when the number of rules scale up.

Based on the Bit Vector (BV) algorithm , proposed by T.V. Lakshman and D. Stiliadis in [1], we developed an improved packet classification algorithm called Hierarchical Bit Vector (HBV), which significantly reduces the storage space and query time. It can scale up to support dozens of thousands of rules with reasonable memory. And it is suitable to be implemented in hardware to take the advantages of parallelism and pipelining to speed up the searching, while it is also amenable to software to perform in general CPU.

1.4 Organization of the Thesis

This thesis is organized as follows. We describe the packet classification problem in Chapter 2, and review the previous works in Chapter 3. In Chapter 4 we start to analyze the two related algorithms and develop our proposed approach with some experiments. At last, we conclude in Chapter 5.

Chapter 2

Problem Statement

2.1 Packets and Rules

Since we only focus on the packet classification based on the packet header, we refer to a packet by describing the fields in its header. For a packet p , $p^{(j)}$ denotes the j th field in its header, here $j \in [1, d]$ and d is the number of fields in the header. Similar with a packet p , a rule R also has d fields. $R^{(j)}$ denotes the j th components of a rule R , and it is an expression of a condition. In practice this expression of condition varies according to different fields. It could be prefix for IP destination and source address fields, or operator/number for port number fields. In prefix specification the expression is a string of bits representing the prefix, and an example is 0101^* , where ‘*’ is wildcard and means the later are don’t-care bits. The operator/number specification consists of an operator (‘=’, ‘>’, ‘<’, etc.) and a number. An example is ‘>1023’, which is often used in firewalls to restrict the incoming packets based on the port numbers. Commonly the expression of a component always defines a

range. For prefix specification, 0101^* is equivalent to the range $[01010000, 01011111]$, if assuming the total length of prefix is 8. A rule R is said to *match* a packet p , if $\forall j, p^{(j)}$ satisfies the expression $R^{(j)}$, or in another word, falls in the range defined by $R^{(j)}$.

From the view of geometry, a packet can be viewed as a point in a space. If $d=2$ the space is a 2-dimensional plane and if $d>2$ it is a hyper-space. Since one component of a rule defines a range in that dimension, the whole rule can be viewed as a rectangle or hyper-rectangle in the space. If a rule R matches a packet p , it means that in the space the point p falls in the rectangle represented by R , or the rectangle covers the point.

2.2 Ruleset and Best Matching

For a classifier there are N rules and we use $\mathcal{R}$ to denote the set of rules (ruleset), $\mathcal{R}=\{R_1, R_2, \dots, R_N\}$, where $R_i, i \in [1, N]$, is the i th rule. Also the ruleset can be considered as a set of rectangles in the space. Note that it is possible that some rectangles overlap with each other in some areas. For a point in those areas multiple rectangles cover it, which means more than one rule match the packet. And the best matching rule should be selected as the final result.

From a geometric view there are three kinds of relations between two rules, irrelative, overlap and full covered, which are shown in Figure 2.1. For the situation (b) when the point falls in the intersection area, both the rules match it, but only the best one is returned as the query result. For the situation (c), when the point falls in the inner rectangle, the inner rule, rather than the outer rule, is the best matching rule. Otherwise the inner rule is not necessary.

Figure 2.1: Relations of rules: (a) irrelative; (b) overlap; (c) fully covered

2.3 Packet Classification Problem

The packet classification problem is to *efficiently* build the data structure from the ruleset and *quickly* find out the best matching rule in the data structure for the given packets. ‘Efficiently’ means the data structure should be small to save storage space, and simple to support the dynamic incremental update of the ruleset. It is not good if, when a new rule is added to the classifier, the whole data structure is required to be built again. ‘Quickly’ means the query for a given packet header should be as fast as possible. There are two aspects for the query time: average time and worstcase time. Usually the query time is measured by the number of memory access times.

The general multi-field packet classification is a hard problem. It can be viewed as the point location problem in a multi-dimensional space. There is a theory conclusion for the general point location problem about the storage space and query time in the worstcase [1]: when $d > 3$ the query time complexity is $O(\log^{d-1} N)$ with $O(N)$ storage space, or $O(\log N)$ query time with $O(N^d)$ storage space.

However, for the packet classification problem in the real world, we believe that there

are some structures and characteristics in the ruleset[5], which can be exploited heuristically to improve the performances. In this thesis we focus on developing the classification scheme for the practical classifiers and evaluating our method based on synthetic classification databases, which may be encountered in real world applications.

Chapter 3

Review of the Packet Classification Algorithms

In the past several years there were many algorithms proposed to solve the packet classification problem. We will briefly introduce these algorithms in this chapter. In order to explain the algorithms we use a simple example classifier in Table 3.1 to illustrate the data structures and performances.

3.1 Linear Search

Linear search is the simplest way to search a database. For the packet classification problem all the rules are stored in a linear array. When a packet comes, its header is compared with each rule sequentially until a matching rule is found. If there are multiple matching rules, we can arrange the rules in decreasing order of priorities and the first matching rule is the

Filter	Destination Field	Source Field
R_0	0*	10*
R_1	0*	01*
R_2	0*	1*
R_3	00*	1*
R_4	00*	11*
R_5	10*	1*
R_6	01*	00*
R_7	11*	10*
R_8	11*	0*
R_9	*	00*

Table 3.1: A simple example classifier of two fields.

best one. Otherwise, all rules must be examined and the best rule is selected according some criteria.

Linear searching algorithm is efficient in terms of storage of rules, which is $O(NdW)$. However, since the matching rule could be in the bottom of the array, the searching time is poor, which is proportional to the classifier's size N , and has poor scalability when N increases. So it is only suitable for small classifiers.

3.2 TCAM

Content Addressable Memory (CAM)[16] is a special kind of memory: given a number, the correspondent address, where this number is stored, is returned. While in normal RAM, the address is the input and the output is the content in that address. CAM is often used in table lookup. In ternary CAM (TCAM) each cell can take three valued: '0', '1' or wildcard '*'. Each element of TCAM can have a string of '0', '1' and '*', such as 001101*. So

TCAM is suitable to lookup the IP address prefixes. During the search in TCAM, the input is broadcasted to all elements and each element checks if the input data matches the content in its own storage. Note that all the elements are doing the comparison in parallel, which means the searching could be very fast. After the comparison it is possible that more than one elements match the input value. Thus an encoder is used to find the highest matching elements.

Figure 3.1: Classification with TCAM

With the rules in the TCAM, it can be used for packet classification. The basic architecture is shown in Figure 3.1. The rules are stored in the TCAM in the decreasing order of priorities. When a packet comes, the memory array gives out an N -bit long vector showing the matching results for every element. The encoder indicates the address of the highest matching element. Feeding the address to the RAM, the corresponding flow ID or action code can be found.

Although the TCAM is very fast (one clock lookup), it still has some disadvantages.

First it is less dense than the RAM. More transistors are used for storage of one bit in TCAM. So the cost of TCAM is much higher than the similar size RAM. Furthermore, TCAM has much larger power dissipation because each element compares the input in the same time. There are some detail investigations about the searching with TCAM in [6].

Because in TCAM the rules are stored in some order, the updating of rules is a problem. When a new rule is added or deleted, the rules with the lower priorities should be shifted, which has the worst time complexity $O(N)$. For the prefix searching problem, a fast updating algorithm CAO_OPT for TCAM is proposed in [16], whose operation times is less than $D/2$, where D is the length of the longest prefix.

3.3 Bit Vector Intersection

The Bit Vector intersection (BV) algorithm, proposed in [1], decomposes the d -dimensional classification problem into d 1-dimensional subproblems. Figure 3.2 illustrates the basic idea of the BV algorithm in a 2-dimensional plane. There are five rules and they are projected on the two dimensions, leading to some intervals on each dimension. Every interval has a 5-bit long bit vector showing which rule covers this interval. If i th rule does, the i th bit is set to '1', otherwise it is '0'. Given a point, we first find out on each dimension in which interval the point is. The bit vectors of intervals indicate what rule(s) *possibly* cover the point. As the example point in Figure 3.2, the two intervals are X3 and Y6. Then we can calculate the intersection of bit vectors of these two intervals. The '1' bit(s) in the intersection bit vector indicates the final matching rules, which match the point on all dimensions.

Figure 3.2: Bit Vector intersection algorithm.

Since there are N rules, the length of bit vectors is N bits and there are at most $2N - 1$ intervals in each dimension. So the storage complexity is $O(dN^2)$. Note that when N is very large the bit vectors could be very long, say, 10,000 bits. In order to calculate the intersection, we need to read out all the bit vectors. The access time for a bit vector is N/w , where w is the data bus width of the memory. The total query time can be reduced by parallel lookup in each of the dimensions. If the 1-dimensional lookup time is t_1 , the query time complexity is $O(t_1 + N/w)$. This algorithm works well for a small classifier. However, when the N is very large, the storage increases quadratically and the reading of bit vectors is stressful.

We will continue to describe the BV algorithm in detail, and its improved ABV algorithm, in Chapter 4. There we will develop our classification scheme based on the basic

BV and ABV algorithms.

3.4 Cross-producing

In [7] another algorithm called cross-producing is discussed for multiple dimensional classification. In this algorithm we first consider each field separately and build a single column table for each dimension, by projecting all the rules on this dimension, which means only the corresponding field of rules is considered. The projections of our simple example classifier in Table 3.1 are shown in Table 3.2. The cross-producing table includes all the combinations of entries from the projection tables. For each entry in the cross-producing table, the matching rule is precomputed. The example cross-producing table is shown in Table 3.3. For a given packet, after the independent lookups in all dimensions' projection tables, we get a range for each dimension. This string of ranges is then used to look for the best matching rule in the cross-producing table.

destination field	source field
00*	00*
01	01*
10*	10*
11*	11*
0*	0*
*	1*

Table 3.2: Projection tables on each dimension.

Although the cross-producing table is very simple and suitable for arbitrary dimensions, it has the memory explosion problem. If there are N rules, the size of cross-producing table is $O(N^d)$. Moreover, when a rule is inserted or deleted many entries

cross-products	matching rule
(00*, 00*)	R_9
(00*, 01*)	R_1
(00*, 10*)	R_0, R_2, R_3
(00*, 11*)	R_3, R_4
...	...
(*, 00*)	R_9
(*, 01)	NULL
(*, 10*)	NULL
(*, 11*)	NULL
(*, 1*)	NULL
(*, 0*)	NULL

Table 3.3: Cross-producing table.

in the cross-producing table should be recomputed. Thus it is not suitable for applications with frequent update.

3.5 Hierarchical Tries

A trie is a binary branching tree and every node has two branches labeled with '0' or '1'. There is a prefix associated with each node, which is the concatenation of all the bits met from the root to the node. It is usually used to search for the longest IP address prefix in the traditional address lookup problem. The ranges can be translated into several prefixes[6]. Thus we can only consider the rules with prefixes.

3.5.1 Basic Trie of Tries

Intuitively, the 1-dimensional trie data structure for the longest IP address lookup can be extended to multiple levels of hierarchical tries for multi-field classification[6]. The hierarchical data structure is built as follows. First, we construct the first level trie, L_1 -trie, which is based on the first fields of the whole ruleset $\mathcal{R}$. Each node in L_1 -trie is linked to a trie of next level, L_2 -tries, by a pointer. Specifically for node u_1 in L_1 -trie, its L_2 -trie is built based on the second fields of the ruleset $R(u_1)=\{R_i \mid R_i^{(1)} = u_1\}$. And in this L_2 -trie there is a node u_2 , whose link points to a L_3 -trie which is based on the third fields of the ruleset $R(u_2)=\{R_i \mid R_i \in R(u_1), R_i^{(2)} = u_2\}$. Generally for a node u_k in a L_k -trie, it has a pointer to a L_{k+1} -trie, which is based on the k th fields of the ruleset $R(u_k)=\{R_i \mid R_i \in R(u_{k-1}), R_i^{(k)} = u_k\}$. Here u_{k-1} is the node of a L_{k-1} -trie, which points to the root of u_k 's trie. We repeat building the hierarchical tries until $j = d$. And in the leaf nodes of L_d -tries not only the prefixes but the best matching rules are stored, which are the final query results.

Finding the best matching rule involves searching all possible paths in the hierarchical tries. Given a packet $p = [p^{(1)}, p^{(2)}, \dots, p^{(d)}]$, searching starts from the root of L_1 -trie and traverses according to the bits in $p^{(1)}$. In the traversing path whenever a node is encountered, the searching continues to traverse the next level tries following the pointer in that node, if it exists. The traversing will go on until the leaf node of the last level trie is met. Since in one dimension there are possibly many matching rules, the backtracking in each trie is needed to make sure all the possible nodes are checked. After scanning the trie of tries, a set of rules is obtained and from them we can find out the best matching rule. For instance in our simple example classifier in Table 3.1, the incoming packet is (001, 100) and searching

paths are shown in Figure 3.3. First we start from the root of the destination trie (L_1 -trie) and there is an associated source trie (L_2 -trie), so we try to search the fourth source trie in that figure. But we fail in the first bit of the second field. Then we backtrack to the second node “0” in the destination trie. Again we try to scan its source trie, which is the second trie in the bottom level in the figure. This time there are two matching rules, R_2 and R_0 . Similarly we check the source tries of node “00” and “01” in the destination trie and find the third matching rule R_4 . From these three rules, the best matching rule is selected by some criteria.

Figure 3.3: Basic trie of tries.

In the basic trie of tries data structure the query time complexity is $O(W^d)$. Because in each trie there are at most W queries and there are d levels of tries, thus the maximum number of total queries is W^d . The storage complexity of this algorithm is $O(NdW)$, since each rule is only stored once.

3.5.2 Set Pruning Tries

In order to avoid the backtracking in the basic trie of tries, a data structure called Set Pruning Tries is proposed in [7], in which we only check the longest matching prefix in each trie, which means reduced searching time. However, it doesn't come without sacrifice. In set pruning tries the rules are replicated to avoid the backtracking. The construction of the L_1 -trie is the same with the basic trie of tries. However, it is different in the following tries. For a node u_1 in L_1 -trie, its associated trie is built based on all the rules whose first fields match u_1 , rather than *exactly* u_1 . Figure 3.4 shows the set pruning trie of our simple classifier. For the node "00*" in the destination trie, rules R_4, R_3 are included since they have "00*" in the first field. Moreover rules R_0, R_1, R_2, R_9 also match it in the first field and they are also duplicated to the source trie pointed by node "00*" in the destination trie. In another word, this source trie should contain all the rules whose destination fields are the prefixes of "00". Thus some rules will be duplicated many times into the longest nodes, such as $R_9=(*, 00^*)$, which is duplicated five times into all the leaf nodes in the destination trie.

Because of the duplications of the rules the query only needs to traverse each level of tries, find the longest matching node and follow the pointer to the next level trie, until the best rule is found. The searching time complexity for set pruning tries is $O(dW)$ with the expense of storage requirement of $O(N^d dW)$, because each of the rules can be at most copied $O(N^{d-1})$ times. The updating needs $O(W^d)$ queries to copy a new rule into many nodes.

Figure 3.4: Set pruning tries.

3.5.3 Grid of Tries

Grid-of-tries [7] is proposed for 2-dimensional classification, which avoids the memory blowup and also improves the search time. It is still based on the structure of basic trie of tries. Moreover, the key point is to use switch pointers to speed up the searching process and avoids backtracking the ancestors without the duplications of rules.

In two levels basic trie of tries, when we finish searching one node in the destination trie and its linked source trie, we backtrack to an ancestor node and follow the pointer to search another source trie. And we do not use the result from the previous searching in the source trie. While the switch pointers make it possible to skip searching some nodes in the source tries. For instance in Figure 3.5, the coming packet is (000, 001) and the longest matching prefix “00” in destination trie is found. Then the source trie containing R_3 and R_4 is searched. But at the first bit of source address it fails. In the algorithm of basic trie

Figure 3.5: Grid of tries.

of tries, we should go back to node “0” in the destination trie and start the search from the root of the corresponding source trie, which contains R_0 , R_1 and R_2 . However, now with the help of switch pointer we can jump directly to the node “0” in the next source trie (containing R_0 , R_1 and R_2) and skip over the root node. Since the second bit in source address of this packet is still ‘0’, the comparison fails again. Still we go along the switch pointer to the next source trie (containing R_9 only) and find the node “00”, in which R_9 is stored. There is no nodes and switch pointers any more. Thus R_9 is the result of this searching.

The switch pointer improves the search time to $O(W)$ since we only need to check each bit once. The storage complexity is same as the basic trie of tries, $O(NdW)$. However, the updating is more complex because inserting or deleting one rule can cause a lot of

recomputing of switch pointers. Thus it is suitable for 2-dimensional static classification.

3.6 Segment Tree

In [2] C. Su proposed a scheme based on segment tree, which focuses on the 2-dimensional classifications. In this scheme each address prefix can be considered as a segment in the whole address space, which can be encoded as two endpoints. For example, prefix 128.255.0.0/16 represents the segment from 128.255.0.0 to 128.255.255.255. If there are N prefixes, the $2N$ endpoints could partition the whole address space to at most $2N + 1$ non-overlapping intervals, which are called *element intervals*. We can construct a segment tree whose leaves are the elementary intervals.

Figure 3.6: Segment tree.

An example segment tree is shown in Figure 3.6, which is a balanced binary tree. How-

ever, it is not necessary to be binary and multiway branching can be used to shorten the height of the segment tree, which can reduce the query time in the tree. A leaf node corresponds to one elementary interval and a node covers some elementary intervals. For instance, node u_2 covers the first half of the whole space: u_8, u_9, u_{10} and u_{11} . The ranges in the rules also cover some elementary intervals, which are shown as lines labeled with $p_1 \sim p_6$. The ranges can be stored in the nodes in some way. Let's first define a set of elementary intervals $I(u)$, which are covered by the node u . A segment p is stored in node u , if $I(u) \subseteq p$ and $I(u\text{'s parent}) \not\subseteq p$. All the segments stored in node u are called the canonical set of u . In this example, the canonical sets of u_4 and u_5 are $\{p_3\}$ and $\{p_1, p_5\}$, respectively. And another number $dp(u)$ is defined as the point which divides $I(u)$ as two subsets corresponding to the two children of node u .

In order to classify the packets, first we build a segment tree based on the $\mathbb{R}^{(1)}$, which means the first fields (destination address field) of the rules. Given a packet, the matching path can be determined by searching the segment tree from the root to the leaves. When node u is encountered the corresponding field of packet is compared with the $dp(u)$. If $dp(u)$ is larger we continue to the left child and if $dp(u)$ is smaller we continue to the right child, until a leaf node is reached. Next we search for the proper rule in the canonical sets of the nodes in the matching path, which is a classic routing lookup problem. Although there are a lot of existing solutions, the author suggests to still use the segment tree on the source address field to find the result. We can pre-compute the best rule for each elementary interval and store it in the right endpoint of that elementary interval. Thus when the search in the segment tree of source address is over, an elementary interval is obtained and the best rule is also found. And if we have the matching rule in each canonical set, we can compare

them and get the final best result.

3.7 Area-based Quadtree

In [4] the authors proposed the Area-based Quadtree (AQT), which is also for the 2-dimensional classifications. In the 2-dimensional case, quadtree is another way to explore the address space and find the best result. Quadtree is a kind of data structure representing the division of the whole space into subspaces recursively. A simple quadtree is a 4-way branching tree, in which the space or subspace is divided into four equal size squares recursively. Each node corresponds to a subspace and its four children nodes correspond to the four divisions of that subspace. We use two bits to represent the four divisions. The first bit shows the X (horizontal) position of the small square. ‘0’ means the square is in the lower part along the X axis. ‘1’ means it is in the upper part along the X axis. Similarly we use the second bit for the Y (vertical) position of the square. Figure 3.7 illustrates this decomposition of the address space.

Figure 3.7: Splitting the space recursively.

Simply, the space is decomposed recursively until all points in the last subspace belonging to one rule and the rule is stored in the the leaf node. Given a packet, we interleave the bits of X coordinate (destination address field) and Y co-ordinate (source address field)

of the packet to form a string of bits. Starting from the root node, we use this bit-string to make the branching decision and traverse the quadtree until a leaf node is reached. With the information stored in that node we can find the final query result. This basic scheme has the query time $O(h)$, where h is the height of the quadtree. While the space complexity is $O(N^2)$ in the worst case.

The concept of the Crossing Filter Set (CFS) is introduced to reduce the storage requirement. The set of rules crossing the square represented by node v is called $\text{CFS}(v)$, which is stored in the node v . $\text{CFS}(v)$ consists of two parts, $\text{CX}(v)$ and $\text{CY}(v)$. $\text{CX}(v)$ is the set of rules that cross the square perpendicular to the X axis. And $\text{CY}(v)$ is the set of rules that cross the square perpendicular to the Y axis. The quadtree with CFS of our simple classifier in 3.1 is shown in Figure 3.8. Because all the rules in $\text{CFS}(v)$ cross at least one axis, we only need to search $\text{CX}(v)$ and $\text{CY}(v)$ in one dimension to find the best rule using the existing longest matching prefix algorithms. So when a node is visited, a best matching rule in corresponding CFS is found by some 1-dimensional algorithm. Along the traversing path we compare these best matching rules from each node and get the final best rule when the leaf node is reached.

Because each rule is stored only once, at the highest node where it crosses a subspace, the storage complexity is $O(N)$. If the searching time for 1-dimension is $O(\log W)$, the total query time is $O(h \log W)$. The authors also proposed an efficient update algorithm with $O(\alpha W)$ query time, $O(N)$ space and $O(\alpha^\alpha \sqrt{N})$ update time.

Figure 3.8: A quadtree with CFS for the example classifier. The rules in brace are the CFS.

3.8 Tuple Space Search

Tuple search[8] algorithm partitions the rule space into many small subspaces according to the length of the prefixes in each dimension. The tuple space $T(a, b)$ represents the set of rules whose rules have a -bit long prefixes in the first field and b -bit long prefix in the second field. Because in a tuple space all rules have the same prefix length in each field, the hash table can be used to search the exact matching rule with the concatenation of prefixes bits as the hash key. For example, if the incoming packet is $p=[p^{(1)}, p^{(2)}]$ we can use the first a bits of $p^{(1)}$ and b bits of $p^{(2)}$ as the hash key to search the hash table for tuple space $T(a, b)$. Note that in a tuple space if there is a matching rule, there is only one.

Finding the best rule involves several hash table probes in all possible tuple spaces. Query time is the total time of a number of hash table accesses. While the number of possible tuple spaces is large (32×32 for two dimensions), some precomputations of marks are used to guide the probe sequence and reduce the number of probed tuple spaces. Figure 3.9 shows a guided route of probing the tuple spaces. Refer to [8] for the details of this algorithm.

Figure 3.9: Tuple space search for 2-dimensional classification.

The storage complexity of tuple search is $O(N)$ since one rule is only stored in one hash table. The access time is $O(M)$, where M is the number of probed tuples. For the guided tuple search $M=2W$ for the 2-dimensional case. Incremental update is supported. And just one hash table, which is associated with the tuple of the involved rule, is needed to update.

3.9 Recursive Flow Classification

Since the general packet classification problem is very difficult and has very poor theoretical worst case bound, Gupta *et. al.* examined some real filter databases [9] and found considerable structures which can be exploited by some heuristic algorithms.

As mentioned above, the rules for multi-field classification can be viewed as the hyper-rectangles in the hyper-space. The goal of the classification is to find the rectangle that matches the most for a point in the space. Intuitively the more overlapping regions the database has, the more difficult the search is. In general there are as many as N^d regions

for an N -rule database. However, Gupta *et.al.* found that in the practical classifiers the number of these overlapping regions are much smaller than the worst case. For example, the number of distinct overlapping regions in a four-field filter database of 1734 rules is 4316, while the worst case number is about 10^{13} . Instead of finding an algorithm working well in the worst case, another direction is to aim at solving the problem for real databases with heuristic algorithms.

The multi-field packet classification problems can be viewed as finding a mapping from the space of the header to the much smaller space of flow ID, which is the result of classification. The simplest and fastest way is using an array of 2^S elements, where S is the total number of bits in the packet header. Each element holds the pre-computed best matching flow ID corresponding to its address. When a packet is given, the flow ID can be indexed by the whole header in one memory access. But this scheme is unrealistic because an extremely large memory is required. The Recursive Flow Classification (RFC) algorithm [9] takes the same mapping principle but it has more stages of mapping, shown in Figure 3.10.

Figure 3.10: The mapping of packet headers to flow IDs (assuming the length of packet header is 128 and the length of flow ID is 12).

Figure 3.11 illustrates the architecture of the RFC algorithm. In order to reduce the size

of memory, the long packet header is splitted to a few chunks and each chunk feeds on the index of a relatively small table. The outcome of each lookup is shorter than the index of the table, which means the input data space is reduced. After the first stage the outcomes are concatenated and go to the index of next stage tables and shorter values are returned. This procedure can go on recursively until we get the flow IDs.

Figure 3.11: The data flow in RFC algorithm.

The RFC algorithm requires a lot of pre-computations of the content of the memory blocks, detailed in [9]. So the dynamic incremental update is difficult, because the insertion or deletion causes the reconstruction of all the tables. It is only suitable for relatively static classifiers. It works well in the experiments in [9]. But when the number of rules is too large, say, greater than 6000, the algorithm has the storage problem. Because this scheme exploits the structure of the classifier the query time and memory space are not predictable

and vary with different databases.

3.10 Prefix-based Searching

In [14] a prefix-based algorithm is proposed for the multi-field classifications. In this scheme the searching consists of two stages. In the first stage each field of the incoming packet is looked up in parallel to find the matching ranges. In those 1-dimensional range searches each range is assigned a code (*range vector* in [14]) to represent it. After the first stage of range search, d codes are concatenated and are fed to an second stage table, in which the rules are stored. With some smart designing for the codes, finding the best matching rule in the second stage table is a longest-prefix searching problem. The incremental insertion is supported but there is no detail about the deletion. Figure 3.12 shows the parallel architecture of this prefix-based packet classification algorithm.

Figure 3.12: Prefix-based searching with the parallel architecture.

3.11 Composite Schemes

Since it is impossible to find one algorithm addressing the problem with very good performance on all aspects, some researchers [10] [11] [12] [3] tried to explore the classifiers and integrate several algorithms together to perform the classification. The main idea is to search the ruleset in multiple stages and after each stage the candidate ruleset shrinks. In different stages different algorithms are used for the better performance within this stage. Here we only briefly introduce the Hierarchical Intelligent Cuttings (HiCuts) in [10].

3.11.1 Hierarchical Intelligent Cuttings

A heuristic algorithm called Hierarchical Intelligent Cuttings is proposed in [10], which uses tree structure and linear tables to store the rules. In this algorithm the multi-dimensional hyper-space is decomposed according to some criteria until the number of rules in each sub-space is smaller than a pre-defined parameter. Searching for the best matching rule starts from the root node of the tree and traverses to a leaf node and lookup the matching rules in the small linear table stored in the leaf node.

We know that a node u represents a sub-space $S(u)$ and the root node represents the whole hyper-space. For a node u we assign a set of rules $R(u)$ to it, which includes all the rules that overlap or fully cover the sub-space of u . At node u if the size of $R(u)$ is larger than a pre-defined value, say $binth$, the u is further divided to np child nodes. If $R(u)$ is less than $binth$, no cutting will be performed and this node u actually is a leaf node in the tree with $R(u)$ rules stored in a linear table, which can be searched by linear algorithm.

Generally, there are an unlimited number of ways of decomposing a space, while the

authors choose to split the sub-space in one dimension during one cutting. There are various metrics that can be used to select the suitable dimension in one node, such as minimizing the maximum size of ruleset in the child nodes or select the dimension with the largest number of ranges in that dimension. More metrics and the tuning of the parameters can be found in [10].

Figure 3.13: One of the possible data structure of HiCuts for the example classifier.

For our simple example classifier, one of the possible data structures is shown in Figure 3.13, where *binth* is 2 and $(Y, 4)$ means Y dimension is partitioned to 4 intervals. We can see that in the heavily overlapping region (such as the region with R_0, R_2, R_3) even the size of $R(u)$ is larger than *binth* it is useless to decompose it further to child nodes.

Given a packet the query first traverses the tree according to the proper bits in the header, until a leaf node is reached. The best matching rule is obtained by linear search. In the experiments in [10] the heights of trees could be very small (≤ 4), making for fast searching.

This heuristic algorithm of RFC has the worst case storage complexity $O(N^d)$. Because in the partition of one dimension a rule can be duplicated at most N times and N^d copies

could be stored in the worst case. However, in the real classifiers the storage performance is relatively good according the experiments in [10]. The query time depends on the height of the tree and the value of *binth*. The update is fast and the complexity is the same as the query time.

3.12 Summary

Here is the table for the storage and query time complexities of all the above algorithms.

algorithms	storage complexity	query time complexity
Linear Search	N	N
Ternary CAM	N	1
BV	dN^2	$dW + N/w$
Cross-producting	N^d	dW
Basic Trie of Tries	NdW	W^d
Set Pruning Tries	$N^d dW$	dW
Grid of Tries	NdW	W^{d-1}
Segment Tree	$(l + 1)W$	$l \times N^{1+1/l}$
Area-based Quadtree	N	$\alpha^\alpha \sqrt{N}$
Tuple Space Search	N	N
RFC	N^d	d
HiCut	N^d	d

Table 3.4: Storage and query time complexities of some algorithms.

Chapter 4

Hierarchical Bit Vector Scheme

In this chapter we will introduce the idea of hierarchical search and then describe our packet classification scheme. Then we will evaluate the scheme with some synthetic classifiers and compare the performance with other similar algorithms.

4.1 Concept

First we review the Lucent Bit Vector (BV)[1] and Aggregation Bit Vector (ABV)[5] algorithms. From the investigation of those two algorithms we develop our idea of Hierarchical Bit Vector search and describe the data structure with the simple example classifier in Table 3.1.

4.1.1 Bit Vector Algorithm

As mentioned before, Bit Vector (BV)[1] algorithm divides the d -field classification problem into d 1-dimensional subproblems and the final result comes from the combination of d 1-dimensional searching results. For each dimension there is a searching trie associated with the corresponding field of the ruleset. If in some fields the range is not based on the prefix format, such as the source and destination port numbers and the protocol type, it can be translated into multiple prefixes [7] [8], or we can use range trees for those fields. In the tries a N -bit long bit vector is associated with a node, if this node is corresponding to a prefix of some rules. Here N is the number of rules in the ruleset. In the bit vector every bit represents a rule in the ruleset. If i th bit is set to '1', it means the i th rule matches the node associated with this bit vector, otherwise it is '0'. Thus from the bit vectors we can figure out what rules match the incoming packet in each dimension. By finding the common bits in the d bit vectors we can get the matching rules on all dimensions and select the best one with highest priority or lowest cost as the final result.

Figure 4.1 shows the data structure of the basic BV algorithm for the simple example in Table 3.1. The short bit vectors are for Aggregated bit Vector algorithm which is described in the next section. Here we only consider the long vectors for the BV algorithm. In each dimension the trie is built with the distinct prefixes in the corresponding fields of all the rules. For instance, the destination trie has 6 nodes: *, 0*, 00*, 01*, 10* and 11*. In each node the associated bit vector shows which rules match the prefix of this node. For example, the destination field of R_3 and R_4 is 00*. So the node of 00* in destination trie has the 3th and 4th bits set to '1'. Moreover, 00* is also matched by * and 0*, which

Figure 4.1: Two tries for each of the fields in the simple example of Table 3.1. The long bit vectors are for BV algorithm, and the short vectors are for ABV algorithm with aggregate size 4. .

are corresponding to R_0, R_1, R_2, R_9 . Thus the bit vector of 00* node in destination trie is 1111100001.

When a packet comes, each of the fields of the header is extracted and queried in the corresponding trie. In each of the tries we search for the longest matching node and get the associated bit vector. With the vectors in all the dimensions we take the intersection operation to find the set of rules matching the incoming packet and then the best matching rule is selected. If the sequence of the rules in the bit vectors is in decreasing order we only need to find the index number of the first '1' bit in the intersected bit vector. If the rules are arranged arbitrarily, all the costs or priorities of rules identified by bit '1' should be read off and compared to find the best one.

Since there are N rules, each bit vector is N -bit long and in each of the tries there are N prefixes in the worst case. Thus, in one trie the storage for the bit vectors is $O(N^2)$. The whole storage complexity is $O(dN^2)$, for there are d dimensions. We need to read off d bit vectors to compute the intersection and find the best matching rule. When the data bus width of memory is w , then we need $\frac{N \times d}{w}$ memory accesses for all the bit vectors. When the number of rules N is large, the reading of bit vectors is stressful, which limits the classification speed. For example, if there are 10k 5-field rules, 50k bits of bit vectors are needed to be read off for the intersection operation. If the data bus width is 32, we may need 1600 memory accesses to get the result, which is obviously not practical.

In a summary although BV scheme is very simple for implementation, there are mainly two disadvantages with the BV algorithm: a) memory explosion of the static storage for the rules; b) stressful memory access time for the dynamic reading of bit vectors.

4.1.2 Aggregate Bit Vector Scheme

In [5] an algorithm called Aggregate Bit Vector(ABV) is proposed to reduce the access time of bit vectors by aggregation. The key point is the assumption that the set bits in the bit vectors are very sparse. For example in a bit vector of 10,000 bits long there might be only 4 bits set to '1' and all others are '0's. Our target is to locate the positions of the '1's. Why do we need to read all of the 9,996 '0's? Thus aggregate vectors are introduced to guide locating the set bits with a much smaller number of memory accesses.

4.1.2.1 Aggregation

In BV algorithm the searching of '1's is linear, in which every bit is read out and checked, whether it is '0' or '1'. In ABV the searching is in a hierarchical manner with the help of shorter *aggregate bit vectors*, which is used to indicate the rough locations of '1's in the original bit vectors. The aggregate bit vector is generated as follows: A long bit vector is divided into some *bit sections* and for each section we use one bit to represent the appearance of the inside set bits, which is called an *aggregate bit*. If there is one '1' bit in one bit section, the corresponding aggregate bit is '1'. The concatenation of the aggregate bits forms the aggregate vector. In order to generate the aggregate bit vector, we need to first fix an aggregate size A , which is the amount of bits in one section and can be tuned to optimize some performance goal. For a bit vector v of N bits long, the length of aggregate bit vector v_a is $\lceil \frac{N}{A} \rceil$. And the i th bit of v_a is '1' only if any of the bit b_j of v is set to '1', $j \in [i \times A, (i + 1) \times A)$; otherwise this bit is '0'. The building of aggregate bit vector is illustrated in Figure 4.2. This aggregation can be repeated at more than two levels, which

can further improve the memory access performance.

Figure 4.2: The aggregation of long bit vectors. (the aggregate size is 8)

With a given packet the longest matching searches are carried out on each of the dimensions, similarly with the BV scheme. However, when we get the d leaves in d tries, we first only read out the aggregate bit vectors of those d leaves and find the common bits in the vector of intersection, from which we can find the indices of the bit sections possibly having the matching rules. Then we dig into those sections in each dimension and read out the corresponding groups of bits of the original bit vectors. Finally we do the intersection and find the common bits representing the rules. Assuming in each dimension the number of set bits in the long original bit vector is very small, for example $n^{(j)}$, the number of set bits in the intersection of aggregate vectors is also small $n \leq \min(n^{(j)})$. We only examine n sections of bits in the long original bit vectors, which are indicated by the set bits in the intersection of aggregate vectors. The memory access time is reduced by skipping reading the rest of the sections. For two levels of aggregation the total number of memory access time is $(n \times A + \lceil N/A \rceil) \times d/w$.

The data structures of the ABV scheme is shown in Figure 4.1 based on the example of

Table 3.1, in which the aggregate size is 4 and the short bit vectors are aggregate vectors. For instance, if the incoming packet header is $(10^*, 11^*)$, in destination trie the longest matching node is 10^* , in which the aggregate bit vector is 011; in source trie the longest matching node is 11^* , in which the aggregate vector is 110. By calculating the intersection of the two vectors, the matching rules must be located in the second bit sections. Then we only read out the second bit section in each of the dimensions, which are 0100 and 1100, respectively. Further, we calculate the intersection again and find the second bit is '1' in both of the bit sections. At last, the matching rule is the second one in the second section, which is R_5 . In this query, we access two 3-bit long aggregate vectors and two 4-bit long bit sections. Assuming the word width is 4-bit, the total memory access time in the ABV scheme is 4, while in the original BV scheme we need 6 memory accesses to get all the vectors to get the result.

4.1.2.2 False Match and Sorting the Rules

In some cases memory access time of the ABV scheme is even worse than the original BV algorithm because of the false match, in which it takes more time to access the aggregate vectors and the original vectors. For example for the classifier in Table 4.1 two bit vectors for packet (X, Y) are '10101010' and '01010111', respectively on the first field and second field. And the aggregate vectors with $A = 2$ are '1111' and '1111'. For all the set bits in the aggregate vector the algorithm assumes there might be a matching rule. However for the first three bits this assumption is wrong, which is called a *false match*. Only in the last bit is there a real match. In this example false match happens in all the bits of the aggregate vectors, which requires to access all the bits in the original bit vectors. Thus when there

rules	field 1	field 2
R_0	X	B_0
R_1	A_0	Y
R_2	X	B_1
R_3	A_1	Y
R_4	X	B_2
R_5	A_2	Y
R_6	X	Y
R_7	A_3	Y

Table 4.1: Classifier with false matches.

are a lot of false matches, the aggregation doesn't reduce the memory access, instead the number of memory accesses could be slightly higher because of the time to retrieve the aggregation bit vectors.

Sorting the rules can help to reduce the false matches. The intent is that the multiple rules which match a specific packet are placed close to each other, so that these matching rules are in the same aggregation group. If we sort the rules according to the first field, the classifier in Table 4.1 is changed to Table 4.2. Still for the packet (X, Y) , the two bit vectors are '00001111' and '11110001', and the corresponding aggregate vectors are '0011' and '1101', with $A = 2$. There is no false match in this case and only the last bit of the intersection is '1', which means only one aggregate group is needed for accessed.

In case we deploy sorting the rules, we assume the positions of the rules are not considered as the priorities. In [9] [5] it is found that the overlap of the rules is very rare. In other words, the case that multiple rules match a packet is very rare and the number of matching rules is very small. Thus it is possible to find all the matching rules if there are multiple matches and compute the best rule by the comparison.

rules	field 1	field 2
R_1	A_0	Y
R_3	A_1	Y
R_5	A_2	Y
R_7	A_3	Y
R_0	X	B_0
R_2	X	B_1
R_4	X	B_2
R_6	X	Y

Table 4.2: Sorted classifier with zero false match.

4.1.2.3 Pros and Cons

With the aggregation and the rearrangement of rules the ABV scheme is faster than the BV by an order of magnitude[5]. But we still need to store the long bit vectors, and this storage is still huge, even worse than the BV because of the addition of aggregation vectors.

4.1.3 Hierarchical Bit Vector Scheme

Assuming the set bits in the bit vectors are very sparse, the ABV algorithm can skip reading a large part of zeroes in the vectors, which only reduces the memory access time. We still need to store the very long bit vectors in the nodes of the searching tries, which takes a large amount of memory when the number of rules N is large. For example, for a 2-dimensional classifier with 10,000 rules, the memory consumed for the bit vectors is roughly 200Mbits. Can we further reduce the storage space for the long bit vectors? Since the set bits are sparse, why do we store the large number of '0's in the bit vectors? Can we only store the segments containing the '1's? The answer is yes. By introducing the hierarchical architecture of the bit vectors, we can shrink the length of the vectors to a preferred value

and the total memory for the vectors is significantly smaller than that in the BV and ABV algorithms.

4.1.3.1 Node Space

In the search trie each node can be viewed as a prefix or a 1-dimensional range. For example, in the space [0000, 1111] the node with prefix 010* represents the range [0100, 0101]. Moreover in the 1-dimensional search trie for a multi-field packet classification problem, a node with a prefix represents a (super-)rectangle. For example, in the 2-dimensional case the multidimensional space is ([0000, 1111], [0000, 1111]) and the node 010* in the destination trie actually represents the rectangle ([0100, 0101], [0000, 1111]). So in the multidimensional view each node in the tries is a subspace of the whole space, which is called *node space*. Especially in the 1-dimensional case the node space is just the range represented by the prefix. Figure 4.3 illustrates an example of the node spaces in one dimensional and two dimensional cases.

4.1.3.2 Partitioning the Space

In the view of multidimensional space, the rules are rectangles included in the whole space. For our simple example, Figure 4.4 shows the rectangles of the rules in a 2-dimensional plane. In BV algorithm the 1-dimensional searching is actually a step by step narrowing down the space along one dimension, which ends with a bit vector indicating the possible rules overlapped or crossed with the leaf node space. Figure 4.5 illustrates some nodes of the destination trie and their corresponding spaces based on the simple example.

When we go along the tries node by node, the node space under investigation is smaller

Figure 4.3: Ranges and rectangles represented by the trie nodes.

Figure 4.4: Rectangles of rules in our simple example.

Figure 4.5: Some nodes spaces in the destination trie of the example classifier in Table 3.1.

and smaller. When we reach a leaf node all the candidate rules are indicated in the associated bit vector. In the node 10^* in Figure 4.5, for example, the candidate matching rules are R_5 and R_9 . We note that in the middle of this procedure some of the rules are already excluded from the candidate matching rules. In the destination trie of our example, when the query process stops at the node 1^* , rules $R_0, R_1, R_2, R_3, R_4, R_6$ definitely can not be the matching rules and the candidates are R_5, R_7, R_8, R_9 , which can be shown in Figure 4.5. In the BV and ABV scheme, the nodes in destination trie following 1^* still store the information about $R_0, R_1, R_2, R_3, R_4, R_6$, the '0' bits in the corresponding positions in the bit vectors. We think it is a waste of storage space since at node 1^* we already know these six rules are excluded and we don't need to keep these positions in the vectors of the following nodes.

Our idea is to locate the query point gradually and hierarchically with the help of space partitioning. The whole space can be partitioned into some subspaces and each subspace covers or overlaps with some rules. The partitioning means that the union of the subspaces is the whole space and the intersection of any pair of subspaces is empty. With the partitioning of the whole space, the whole ruleset is also divided into sub-rulesets. Each subspace has a sub-ruleset. If a rule is overlapped partially or fully with a subspace, this rule belongs to the sub-ruleset of this subspace. Because some rules may be overlapped with more than one subspace, those rules will be copied to several sub-rulesets. Thus the sub-rulesets may be not exclusive with each other. While the rules can be viewed as subspaces, those subspaces from partitioning can also be considered as the *auxiliary-rules*. We use this term because the subspaces are treated as rules to be searched for in some stages of our classification scheme.

4.1.3.3 Hierarchical Search

With the auxiliary-rules, the classification problem is divided into several hierarchical sub-problems and the classifier is divided into several hierarchical sub-classifiers. First, we build a classifier with the subspaces, or auxiliary-rules, as the rules. Each subspace includes some real rules and it can be viewed as another classification problem. Thus in each subspace we again build a sub-classifier based on the included real rules.

With a given point, the query consists of two searching steps. We first search for the matching auxiliary-rule and ignore the real rules included in it. Because no two auxiliary-rules overlap each other, there is one and only one matching auxiliary-rule. Then, hierarchically, in the included real rules corresponding to the matching auxiliary-rule, we again search for the best matching real rule.

Moreover we can repeat the partitioning of the subspaces at multiple levels, forming a hierarchical auxiliary-rules and real rules architecture. The searching is decomposed into hierarchical multi-level searches. In level i , sub-classifier is called *level i sub-classifier* and the ‘rules’ (auxiliary-rules or real rules) are called *level i rules*. In the above description there are only two levels. In the level 1 classifier the ‘rules’ are subspaces and in the second level the ‘rules’ are real rules. When we consider the searching in some level, it doesn’t matter if the rules are auxiliary-rules or real rules. The general idea of hierarchical search is shown in Figure 4.6. Intuitively if each auxiliary-rule can contain up to 32 auxiliary-rules or real rules, an architecture with three levels could support up to $32 \times 32 \times 32 = 32768 \approx 32K$ real rules.

In the ABV algorithm, the authors present a similar statement about the hierarchical

Figure 4.6: hierarchical partition of rule space

levels. However, the ABV approach uses the aggregation in one dimension for the bit vectors, while we use aggregation in the multidimensional space for all the rules. We will show later that if the BV algorithm is used in each level, the multidimensional hierarchy can reduce the length of the bit vectors stored in the nodes. However, the ABV adds an extra bit vector to each original bit vector.

4.1.3.4 Improvement for the BV Algorithm

In the searching of each level we can choose an algorithm from the literature. Some algorithms can benefit from this hierarchical architecture, some can not. Here we only investigate the improvement based on the BV scheme in each level, which is called the Hierarchical Bit Vector (HBV) scheme. With the simple example in Table 3.1, we will show the improvement of HBV in storage and access time.

Storage For our simple example, Figure 4.7 shows one possible partition of the space and the level 1 classifier consisting of the auxiliary-rules. Because we assume the destination and source address fields of rules are all prefixes, our partition is also prefix-based, so that subspaces can be described by the prefixes on each dimensions. Because there are 3 subspaces in our example, the length of the bit vectors in the level 1 classifier is 3, which is much smaller than the number of rules, which is 10.

In level 2 inside one auxiliary-rule, the included ‘rules’, which are real rules in this example, compose the level 2 classifier. The reader should observe however that the specifications of these rules have been changed. In both destination and source address dimensions, certain bits of the incoming packet have been already examined, and only the rest of bits in

Figure 4.7: One partition of the space of the simple example classifier, the first level classifier of the auxiliary-rules and the corresponding searching tries. Only two levels of partition is used in this case. In each auxiliary-rule there are no more than 4 real rules. AR_i represents the auxiliary-rule and the sequence of the bit vectors is AR_0 - AR_1 - AR_2 .

those fields should be checked against the classifier of the current subspace. Therefore, for all the included rules some of the bits in the beginning will be removed in each dimension, and the number of bits removed depends on the specific including subspace. Figure 4.8 shows the three level 2 classifiers. For instance, In the first auxiliary-rule, AR_0 , there are four (real) rules included: R_0, R_2, R_3, R_4 . While $(0^*, 1^*)$ represents AR_0 , if the incoming packet falls in AR_0 in each of its fields, one bit has already been checked. So in the classifier of this subspace, we only need to examine the rest bits and, moreover, we also should ignore the first bit in each field of the included rules. For consistency, the first 1 bit in each field of the included rules will be taken off from the original specifications: R_0 is changed from $(0^*, 10^*)$ to $(^*, 0^*)$; R_2 from $(0^*, 1^*)$ to $(^*, ^*)$; R_3 from $(00^*, 1^*)$ to $(0^*, ^*)$; R_4 from $(00^*, 11^*)$ to $(0^*, 1^*)$. In the subspace of the third auxiliary-rule AR_2 , the included rules are taken off 1 bit in the destination address field but no bit in the the source address field, because AR_2 is $(1^*, ^*)$.

Inside the level 2 classifier, the searching tries are also built based on BV scheme. And the bit vectors are much shorter than N -bit long, here N is 10. Since there are only 4 rules in the AR_0 , 4-bit long bit vectors are enough. Moreover, AR_1 has 3 rules and their bit vectors are 3-bit long; AR_2 has four rules and the length of vectors is 4. In this partition we make sure in each subspace (auxiliary-rule) there are no more than 4 rules, which is called *hierarchical size*. This number can be tuned to adapt to the physical memory word size, such as 16, 32, 64 or higher. For the classifier of AR_0 , in the destination dimension 3 bit vectors are stored and each bit vector is 4-bit long. In the source dimension also has three 4-bit long bit vectors. Thus the storage space for bit vectors in AR_0 is $3 \times 4 + 3 \times 4 = 24$ bits. Similarly for AR_1 it is $2 \times 3 + 2 \times 3 = 12$ bits; for AR_2 it is $3 \times 4 + 4 \times 4 = 28$ bits.

Classifier of rules
in Auxiliary-rule 0

R0	(**, 0*)
R2	(**, **)
R3	(0*, **)
R4	(0*, 1*)

R0-R2-R3-R4
destination trie

source trie

Classifier of rules
in Auxiliary-rule 1

R1	(**, 1*)
R6	(1*, 0*)
R9	(**, 0*)

R1-R6-R9
destination trie

source trie

Classifier of rules
in Auxiliary-rule 2

R5	(0*, 1**)
R7	(1*, 10*)
R8	(1*, 0**)
R9	(**, 00*)

R5-R7-R8-R9
destination trie

source trie

Figure 4.8: Level 2 classifiers of all the subspaces and the corresponding searching tries.

Intuitively, in our HBV scheme the N rules are distributed to a number of subspaces and in each subspace the classifier has some shorter bit vectors, compared with the N -bit long bit vectors in the BV algorithm. Note that the number of rules doesn't change however the bit vectors for each node is shortened. While this shortening doesn't come free, the cost is to perform a previous search to find out the correct subspace. We will show later that we can save a large amount of storage space with a small cost. For our simple example, with the original BV algorithm, the total size of bit vectors is $5 \times 10 + 6 \times 10 = 110$ bits. With the HBV algorithm the size of all the bit vectors in level 2 classifiers is $24 + 12 + 28 = 64$ bits, which is around the half of that in the BV algorithm. That is because we reduce the bit vector length from 10 to 4, almost by half. And as for the cost of searching in the first level classifier, the storage space for bit vectors is $2 \times 3 + 3 \times 3 = 15$ bits. The total size of bit vectors in the HBV algorithm is $15 + 64 = 79$ bits, which is still better than the BV algorithm and the ABV algorithm.

Access time As for the memory access time, the HBV algorithm outperforms the BV algorithm and is as good as the ABV algorithm. Since, in the subspaces the bit vectors are short, the access time for each single bit vector is also reduced a lot. Even with the cost of multiple levels of searching, the total access time is still smaller than that in BV. For example if a 2-dimensional classifier has 1024 rules and the space can be partitioned into 32 subspaces. In each subspace there are exactly 32 rules included. For each level of searching, we only need to read 64 bits of bit vectors. Thus ultimately 128 bits of reading is enough for finding the final best matching rule. While in the BV algorithm at least 2048 bits are needed to be read off.

For our example classifier, in the BV scheme we need to read two 10-bit long bit vectors thus 20 bits in total. In the ABV scheme, the worst case happens if the incoming packet is located in (00*, 01*). The first level searching result is 111 and 101 in destination address dimension and source address dimension, respectively. Then the first and third section of the original bit vectors are needed to be read out, which results in $3 \times 2 + 8 \times 2 = 22$ bits in total. In our HBV algorithm, in the first level classifier two accesses of bit vectors are required to get the first level matching auxiliary-rule. And in the level 2, another two accesses are required to find the final real matching rule. In the first level, the length of bit vectors is 3 and in the second level the maximum length is 4. Thus the maximum total number of bits to be read off is $3 \times 2 + 4 \times 2 = 14$.

Similar with the ABV algorithm, the HBV algorithm searches the classifier hierarchically. But the ABV has the problem of false matching, which sometimes can lead to digging into multiple sections of original bit vectors and severely lower the performance. While in HBV, with the partitioning of the space, no two subspaces overlap each other. Thus we only perform one search for the matching auxiliary-rules in each level, which means the searching time is small and deterministic.

4.1.3.5 Rules Splitting

Although the partitioning of space saves the storage space and reduces the query time, it can lead to the rules splitting problem, which means some real rules are so big or so long that they reside in more than one subspaces. In other words, they are split into multiple sub-rules. In Figure 4.7, note that R_9 appears in AR_1 and AR_2 , because these two subspaces both overlap part of R_9 , and the rule R_9 has two copies in both classifiers of subspace AR_1

and AR_2 . This duplication of rules makes the total size of all last level classifiers larger than the original classifier. That means the storage space is not so efficient as the ideal case, in which one rule only resides in one subspace.

Figure 4.9: Another partition of the space of the simple example classifier, the first level classifier of the auxiliary-rules and the corresponding searching tries. Still there are only two levels of partition.

The ways of partitioning can have different results of rules splitting. Figure 4.9 and Figure 4.10 illustrate another partition and the corresponding searching tries. In this partition we still have no more than 4 rules included in one subspace, and there are no rules split. As a result, the total storage of bit vectors in this partition is 71, which is less than the number of the previous partition, 79. In the terms of total storage space for the bit vectors it is possible to find the *best* partition with smallest storage for a given classifier.

Classifier of rules
in Auxiliary rule 0

R0	(**, 0*)
R2	(**, **)
R3	(0*, **)
R4	(0*, 1*)

Classifier of rules
in Auxiliary rule 1

R5	(0*, **)
R7	(1*, 0*)

Classifier of rules
in Auxiliary rule 2

R1	(0**, 1*)
R6	(01*, 0*)
R8	(11*, **)
R9	(***, 0*)

Figure 4.10: Level 2 classifiers of all the subspaces and the corresponding searching tries.

4.2 Algorithm

In this section we formally describe our new Hierarchical Bit Vector algorithm. We start by describing the partition and search algorithms. Then we describe some variations of the basic scheme.

4.2.1 Partition Algorithm

4.2.1.1 Cut Operation

Since the subspaces look like the real rules, each field of an auxiliary-rule also has the format of a^* . Here a is the string of '0' or '1'. When we partition the space and get subspaces, we actually cut the space along each dimension repeatedly. For example in Figure 4.7, in order to get $AR_2=(1^*, *)$, we should cut the space $(*, *)$ in the middle on the destination dimension then take the upper (right) part subspace as AR_2 . Then we continue to cut in the middle of the subspace of lower (left) part, $(0^*, *)$, on the source dimension. The new upper part subspace $(0^*, 1^*)$ is AR_0 , the lower part subspace $(0^*, 0^*)$ is AR_1 . If we only make the cut in the middle of a space on some dimension, the two outcome of subspaces still have the prefix format. If the space is $(..., a^{(j)}*, ...)$, which means in j dimension the prefix is $a^{(j)}*$, the two subspaces from cutting on dimension j is $(..., a^{(j)}0*, ...)$ and $(..., a^{(j)}1*, ...)$. Thus a partition is equivalent to a serial cut operations on the space and subspaces.

For a subspace (auxiliary-rule) R , let $R.ruleSet$ represent the rules colliding with the space and $R.numRules$ represent the cardinality of the $R.ruleSet$. For a cut on R , $C_j(R)$, it is performed on dimension j . The left (lower) and right (upper) child of the cut are

$LeftChild_j(R)$ and $RightChild_j(R)$, respectively.

Given a space, we can use the repeat of cut operations to partition it into subspaces. First for the whole space we select a dimension and cut the space on that dimension. Then we get two subspaces. For each of them, we again cut it on some dimension and get more subspaces. Generally we check all the subspaces one by one and repeat the cut operations until the subspaces are not required to be partitioned further.

What can make the cut stop? We stop the cutting until the classifier of a subspace is smaller than the hierarchical size. Normally 16, 32, 64 or 128 is reasonable value in order to fit the actual memory word width. And in different subspaces, we even can choose different values for this maximum limit, while for simplicity we use a constant N_0 as this threshold for all subspaces.

The cut operation is similar with the Hierarchical Intelligent Cuttings (HiCut) algorithm proposed in [10], in which the cut takes place in each node of a decision tree. Although our HBV scheme has similar cuttings, we don't use the decision tree and linear arrays for the query. Instead, we build separate tries based on the BV scheme. Actually the HiCut algorithm also cuts the whole space into lots of small subspaces and in each subspace there are a small number of rules so that the linear search can be used. The query in the decision tree is to find the matching subspace, which is corresponding to our searching for the auxiliary-rule. And in the subspaces HiCut uses linear search, while we still use the BV algorithm. Since our HBV scheme can have larger N_0 for each subspace, the tries leading to the subspaces have a much smaller depth than the decision tree. Further, the BV tries for the sub-classifiers can take advantage of the parallelism and increase the query speed.

4.2.1.2 Partition Metrics

Given a classifier there are possibly many ways to partition it, however, there exists the *best* one. What is best? The ideal partition is that there are totally $\lceil N/N_0 \rceil$ subspaces, and each rule only resides in one subspace. Then in $\lfloor N/N_0 \rfloor$ subspaces there are exactly N_0 rules and the rest of rules fall in the last subspace. In this case we don't waste any bit in the bit vectors and the storage for the bit vectors is minimum. On the contrary, if in some subspace there are $n < N_0$ included rules, only n bits in the bit vectors are valid. While the bit vectors are still N_0 -bit long and the rest of bits are wasted. However, in fact it is almost impossible to find that ideal partitioning because in each cut operation the ruleset is divided into two rulesets, whose sizes are possibly not integral times of N_0 .

We use a heuristic to perform the cut operations in each step. There are various metrics for cut operations, for example:

1. Select the dimension to minimize

$$LeftChild_j(R).numRules + RightChild_j(R).numRules$$

in all dimensions to reduce the duplications of the rules.

2. View

$$\frac{LeftChild_j(R).numRules}{LeftChild_j(R).numRules + RightChild_j(R).numRules}$$

and

$$\frac{RightChild_j(R).numRules}{LeftChild_j(R).numRules + RightChild_j(R).numRules}$$

as the possibility distribution and maximize the entropy of the distribution, to make the rules most uniformly distributed among the child nodes.

3. For a ruleset if the cut operation is not carried out we don't know how many rules are in each child node. However, we can assume the rules are distributed uniformly and the expectation of number of rules in each child node is the half, if ignoring the rules' split. So for a ruleset $\mathcal{R}$ with N rules, the expectation of number of cut operations

$$E(M(\mathcal{R})) = \min_m (m \geq \log_2 \frac{N}{N_0})$$

, which also represents the number of subspaces in the end, where m is an integer. So in order to get the minimum number of subspaces, we try the cut operations in all the dimensions and minimize the sum,

$$E(M(LeftChild_j(\mathcal{R}))) + E(M(RightChild_j(\mathcal{R})))$$

With the metrics in each cut operation, we start the partitioning from the whole space until we get a set of subspaces (auxiliary-rules), *subspaceSetLvl2*¹. Each of the subspaces covers some real rules and it forms a *level 1 classifier*, and the size of each level 1 classifier is not larger than N_0 . The *subspaceSetLvl2* itself also can be viewed as a classifier based on the whole space. We call it the *top level classifier*, since it is on the top of the hierarchical architecture. Moreover if the size of *subspaceSetLvl2* is larger than N_0 , we can treat these subspaces as 'rules' and repeat the above partition procedure to produce a set of higher

¹We view the real ruleset $\mathcal{R}$ as the *subspaceSetLvl1*.

level subspaces, *subspaceSetLvl3*. Each element of *subspaceSetLvl3* covers some of the auxiliary-rules (subspaces) in *subspaceSetLvl2*, and it is called *level 2 classifier*. Then the top classifier consists of *subspaceSetLvl3*. We can continue the partitioning to more levels if necessary.

4.2.1.3 Algorithm

Figure 4.11 shows the algorithm partitioning the original classifier and building the hierarchical sub-classifiers. In that algorithm *wholeSpace* denotes the whole space and *subspaceSet* denotes the set of subspaces, or auxiliary-rules. In the beginning the *subspaceSet* only has one element, *wholeSpace*, whose associated ruleset is $\mathcal{R}$. Then we check every element in the *subspaceSet* and for each subspace if its number of rules in the associated ruleset is larger than N_0 , it should be partitioned further, which is performed by calling function **Partition**(*subspaceSet*[*i*]). After the partitioning we get two new subspaces, *leftChild* and *rightChild*. And we update the *subspaceSet* by deleting the obsolete subspace *subspaceSet*[*i*] and add *leftChild* and *rightChild* to the *subspaceSet*. We partition the subspaces until each of the elements of *subspaceSet* has no more than N_0 rules. At this time the top level classifier has *subspaceSet* as its rules. However, if the size of *subspaceSet* is larger than N_0 we should continue to partition the top level classifier to generate another level of subspaces. In the end the sizes of top level classifier and the included sub-classifiers are all less than or equal to N_0 .

Function **Partition**(*someSpace*) is responsible for selecting the *best* dimension to cut along and returning the two children subspaces. In **Partition**() the space is cut along the dimension *j* by the function **Cut**(*someSpace*, *j*) and two subspaces *leftChild* and

```

HiPartition() {
    level = 1;
    wholeSpace.ruleSet =  $\mathcal{R}$ ;
    wholeSpace.numRules =  $N$ ;
    while (wholeSpace.numRules >  $N_0$ ) {
        subspaceSet = {wholeSpace};
        i = 0;
        while (i < Sizeof(subspaceSet)) {
            if (subspaceSet[i].numRules >  $N_0$ ) then {
                [leftChild, rightChild] = Partition(subspaceSet[i]);
                Delete(subspaceSet[i]);
                AddtoSet(leftChild, rightChild);
            } else { i = i + 1; }
        }
        wholeSpace.ruleSet = subspaceSet;
        wholeSpace.numRules = Sizeof(subspaceSet);
        level = level + 1;
    }
}

Partition(someSpace) {
    bestDim = 1;
    bestMetric = 0;
    for j = 1 to d {
        [leftChild, rightChild] = Cut(someSpace, j);
        metric[j] = GetMetric(someSpace, j);
        if (metric[j] > bestMetric) then {
            bestMetric = metric[j];
            bestDim = j;
            bestLeftChild = leftChild;
            bestRightChild = rightChild;
        };
    }
    return [bestLeftChild, bestRightChild];
}

```

Figure 4.11: Hierarchical partition algorithm to partition the space and ruleset into multiple subspaces so that the size of the ruleset of each subspace is less than N_0 .

rightChild are returned. All the dimensions are tried to be cut on and the best dimension is selected by calling the function **GetMetric()** and comparing their qualities. In **GetMetric**(*someSpace*, *j*) we get the quality of the cut according to the metrics described in the previous section. And **Partition()** only returns the two subspaces in the best dimension.

When the space and ruleset is partitioned into hierarchical architecture, in each classifier of each level we build the searching tries for all the dimensions and store the bit vectors in appropriate nodes based on the BV algorithm. The details of the BV algorithm can be found in [1].

4.2.2 Search Algorithm

After the data structure is built, the query for an incoming packet goes through the hierarchical subspaces and leads to the best matching real rules. In each sub-classifier the BV algorithm is used for query. The searching algorithm is described in Figure 4.12.

In this algorithm, $p^{(1)}, p^{(2)}, \dots, p^{(d)}$ is the incoming packet with d fields in the header, $p^{(1)}, p^{(2)}, \dots, p^{(d)}$. And *level* is the number of hierarchical levels after the partition. We start from the top level classifier to search for the matching subspace. In the beginning we can view the whole space as a matching rule and assign it to *bestRule*. Then we get the j th dimension trie rooted on *bestRule* by function **GetTrie**(*bestRule*, j). Function **Truncate**(*bestRule*.Length(j), $p^{(j)}$) cuts off some starting bits of $p^{(j)}$ according to the prefix length of subspace *bestRule* in dimension j . The associated ruleset with current subspace *bestRule* is *ruleSet*. The searches in d tries leads to d bit vectors and we find the common bit by ADD operations. Since we don't use the positions as the priorities of rules, we need the function **Cost**(*someRule*) to get the cost of the rule *someRule* and find the smallest

```

Search( $p(F^{(j)}, F^{(j)}, \dots, F^{(j)})$ ) {
  bestRule = wholeSpace;
  for  $l = level$  to 1 {
    for  $j = 1$  to  $d$  {
       $trie^{(j)} = \text{GetTrie}(\textit{bestRule}, j)$ ;
       $ruleSet = \textit{bestRule}.ruleSet$ ;
       $\text{Truncate}(\textit{bestRule}.Length(j), F^{(j)})$ ;
    }
    for  $j = 1$  to  $d$  {
       $N^{(j)} = \text{LongestPrefixMatchingNode}(trie^j, F^{(j)})$ ;
    }
     $bitVector = 11\dots 1$ ;
    for  $j = 1$  to  $d$  {
       $bitVector = bitVector \cap N^{(j)}.bitVector$ ;
    }
     $bestRule = \text{NULL}$ ;
    for  $i = 0$  to  $\text{Sizeof}(ruleSet) - 1$  {
      if ( $bitVector[i] == 1$ ) & ( $\text{Cost}(ruleSet.R_i) < \text{Cost}(\textit{bestRule})$ ) then {
         $bestRule = ruleSet.R_i$ ;
      }
    }
  }
}
return bestRule;
}

```

Figure 4.12: The algorithm of searching best matching rule in HBV scheme.

cost matching rule. When we find the best matching rule in some level, we assign it to *bestRule*, which will be used to get the more tries in the next level. When we finish the searches in all levels the *bestRule* is just the best matching *real* rule.

4.2.3 Pre-group

We have mentioned that big or long rules can cause rules' splitting and duplications in multiple subspaces. In some observations of real classifiers [5] it is very common that wildcards appear in some fields of some rules, which can cause duplications of those rules. For example, in Figure 4.13 the rules have the format of $(a, *)$ or $(*, a)$, a wildcard in one of the dimensions. The partition makes the rules duplicated in subspaces. For instance, R_1 spans over the subspace AR_0 and AR_1 so that R_1 is included in those two sub-classifiers. Similarly R_2 spans over the subspace AR_0 and AR_2 so that R_2 is also included in two sub-classifiers. In the specific partition shown in Figure 4.13 each rule has two copies in two subspaces.

The reason for duplication lies in the fact that if the cut operation is carried on dimension j , the rules whose j th field is wildcard will be copied to both the child subspaces. When a large percent of the rules have wildcards and there are thousands of subspaces, the rules with wildcards are possible to be duplicated thousands of times, which wastes storage space. To avoid this situation we pre-group the classifier and divide it into several sub-classifiers before the hierarchical partitioning. In the example of Figure 4.13, we group the original classifier into two subsets, one has no wildcard in dimension 1, and the other has no wildcard in dimension 2, which is shown in Figure 4.14. Different from the subspaces and the sub-classifiers from the cut operations, these sub-classifiers' correspondent spaces

Figure 4.13: The wildcards cause serious duplications of rules in the subspaces. Each rule appears twice in the subspaces.

are still the whole space. Thus we call them pre-grouped sub-classifiers. Then we view each pre-grouped sub-classifier as a single problem and use the HBV scheme to solve it. At last the best matching rule is obtained by comparing the results from each pre-grouped sub-classifier.

Formally the classifier is pre-grouped into several sub-classifiers, $\mathfrak{R} = \{SR_1, SR_2, \dots, SR_{\hat{d}}\}$, where SR_i is the sub-classifier rooted on the whole space. There are a lot of ways to do the division to fulfill the goal of reducing the rule duplications. A simple method is to pick the rules from $\mathfrak{R}$ whose first field is not wildcard and put them into SR_1 , then pick the rules whose second field is not wildcard from $R - SR_1$, and so on. In this case $\hat{d} = d$. We show this pre-group algorithm in Figure 4.15.

The pre-group can reduce the rules duplications significantly, which will be shown in the section of experiment. However, the complexity of query time increases because we need to do multiple HBV searches and compare the results to find the best matching rule.

Figure 4.14: Group the ruleset into two subsets according to the wildcards.

```

PreGroup( $\mathcal{R}$ ) {
   $restR = \mathcal{R}$ ;
  for  $j = 1$  to  $d$  {
     $SR_j = \text{PickRules}(restR, j)$ ;
     $restR = restR - SR_j$ ;
  }
}

PickRules( $restR, j$ ) {
   $pickSet = \{\}$ ;
  for  $i = 1$  to  $\text{Sizeof}(restR)$  {
    if ( $restR_i^{(j)} \neq *$ )
       $pickSet = pickSet + restR_i$ ;
  }
  return  $pickSet$ ;
}

```

Figure 4.15: The pre-group algorithm

Forthurmore, these multiple HBV searches can be performed in parallel to speed up the query.

4.3 Evaluation

4.3.1 Analysis of HBV

4.3.1.1 Storage

Figure 4.16: Hierarchical classifiers.

In this section we describe the sizes of hierarchical classifiers and analyze the required storage for the bit vectors. We suppose there are N rules in the classifier $\mathfrak{R}$ and the hierarchical classifiers are shown in Figure 4.16. m_l denotes the number of classifiers in level l . In level 1 there are m_1 sub-classifiers and N distinct rules. In level 2 there are m_2 classifiers and m_1 rules. In the top level, there is only one classifier and m_{L-1} rules, where L is the

number of hierarchical levels. In each level we number all the classifiers and use the level number l and the sequence number i to refer to the i th classifier of level l : $C_{l,i}$. For classifier $C_{l,i}$ we use $N_{l,i}$ to denote the number of ‘rules’ (real rules or auxiliary-rules) in it, and $n_{l,i}^{(j)}$ to denote the number of distinct prefixes in dimensions j . Suppose we use N_0 -bit long bit vectors in all the classifiers. Then the storage for the bit vectors in classifier $C_{l,i}$ are ²

$$S_{l,i} = \sum_{j=1}^d n_{l,i}^{(j)} \times N_0 \leq dN_{l,i}N_0$$

bits. And the storage for the bit vectors in all the level l classifiers are

$$S_l = \sum_{i=1}^{m_l} S_{l,i} \leq \sum_{i=1}^{m_l} dN_{l,i}N_0$$

bits. Then the total storage for all the bit vectors are

$$S = \sum_{l=1}^L S_l \leq \sum_{l=1}^L \sum_{i=1}^{m_l} dN_{l,i}N_0$$

bits.

In level l we define a *duplication ratio*,

$$\alpha_l = \frac{N_{l,1} + N_{l,2} + \dots + N_{l,m_l}}{m_{l-1}}$$

to illustrate in average how many copies a rule is duplicated. The sum of $N_{l,i}$ is the total number of rules in all the classifiers in level l , and m_{l-1} is the number of classifiers in the lower level and it is actually also the number of *distinct* rules in this level. So $\alpha_l m_{l-1} =$

²Here we only consider the storage for bit vectors and ignore the memory for searching tries.

$\sum_{i=1}^{m_l} N_{l,i}$. For level 1 the number of distinct rules is N . Thus $m_0 = N$. note that in level L there is only one classifier and m_{L-1} rules from lower level are all included in this classifier, so that there is no duplication in level L and $\alpha_L = 1$.

We can describe the storage of bit vectors in each level by the number of sub-classifiers and the duplication ratio:

$$S_1 \leq d\alpha_1 m_0 N_0$$

$$S_2 \leq d\alpha_2 m_1 N_0$$

$$S_3 \leq d\alpha_3 m_2 N_0$$

and generally

$$S_l \leq d\alpha_l m_{l-1} N_0$$

the total memory are

$$S = \sum_{l=1}^L S_l \leq dN_0 \sum_{l=1}^L \alpha_l m_{l-1} = dN\lambda$$

bits, where

$$\lambda = \sum_{l=1}^L \frac{\alpha_l m_{l-1}}{N} N_0$$

. Here λ can be considered as the linear parameter for the storage before the elements d and N . If λ is small the storage $O(dN\lambda)$ is much better than that of BV scheme, $O(dN^2)$. We will measure the values of m_l , α_l and λ in the experiments in the next section.

4.3.1.2 Access Time

When looking up inside a sub-classifier with the BV algorithm, the number of memory accesses for the bit vectors is $\frac{dN_0}{w}$, where w is the width of the data bus. If the 1-dimensional searches are parallel with each other, the query time to find the matching ‘rule’ is $O(t_1 + \frac{N_0}{w})$, where t_1 is the time to locate the proper nodes in the tries. For a hierarchical structure with L levels the total number of memory accesses is $O(L(t_1 + \frac{N_0}{w}))$. While L is always small in our experiments, we believe the performance of HBV on query time is much better than the BV scheme and as good as the ABV scheme.

4.3.2 Experiments

In the lack of real classifiers we synthesize our 2-dimensional classifier databases, based on some real routing tables[15] and some observations in [5], to test our new classification scheme.

In the case of 2-dimensional classifiers, the two fields to be examined are destination and source addresses in the packet header. Since the classifier in a router only filters the traffic through this router, it is reasonable that the database of rules is based on the prefixes in the routing table. So our synthetic classifiers are generated by randomly picking the prefixes in the publicly available routing table [15]. And moreover in [5] the authors investigated some real firewall databases and found some characteristics:

- i) Most prefixes have either a length of 0 or 32. There are some prefixes with lengths of 21, 23, 24 and 30.
- ii) No prefix contains more than 4 matching sub-prefixes for each dimension.

- iii) The destination and source prefix fields in roughly half the rules were wildcarded, and roughly half the rules have ≥ 1024 in the port number fields. Thus the amount of overlap within each dimension was large.
- iv) No packet matches more than 4 rules.

In order to generate one rule, the basic way is to pick a pair of prefixes randomly from the routing table as the destination prefix and the source prefix. By repeating N times of that picking we can get a classifier of N rules. We should note that after we generate a rule we need to check if it has appeared before and make sure it is a new unique rule.

However, with this basic method it is almost impossible to get a rule with a wildcard. While in above observations iii), about half of the rules have wildcards in real classifiers. In order to generate more realistic database, a predefined value called *wildcard ratio* is used to control the percentage of the wildcard. Before we start to pick the prefix, we first generate a random number and compare it with the wildcard ratio. If it is larger than the ratio, we go on to pick the random prefix; and if it is less than the ratio we set a wildcard in this field.

4.3.2.1 Different Wildcards Percentages

We synthesize the 2-dimensional classifier of size from 1,000 to 20,000. For each size we also generate classifiers with different percentages of wildcards, from 0% to 50%. In the HBV scheme, we partition the classifiers into multiple levels of sub-classifiers and for each sub-classifier we calculate the memory size of bit vectors. Considering all the sub-classifiers, we can get the total storage requirement for the bit vectors. The query speed depends on the number of levels of the partition. Assuming the hierarchical size, or vector

length, is 64, we compare the query time of HBV with ABV, which is much better than the BV scheme.

Without Pre-group First we use the set of classifiers without the pre-group of ruleset, with 64-bit long bit vectors and the cutting metric 1 for partitioning in HBV. The storage performance as a function of the size of classifiers is shown in Figure 4.17. We can see that apparently the required memory space of BV is from ten to two hundred times of that of HBV. And the presence of wildcards has significant influence on the storage performance of HBV. Because some rules with wildcard are duplicated into multiple sub-classifiers. The required memory increases while the percentage of wildcards is getting larger. For example when the size of classifier is 20k, the memory of HBV with 50% wildcards is almost 100 times of the memory of the classifier with 0% wildcards.

Figure 4.17: Storage of BV and HBV scheme with different percentages of wildcards, without the pre-group and the vector length is 64-bit. The HBV scheme outperforms BV scheme by a factor of one or two magnitudes.

We also can use the parameter λ , which is introduced in Section 4.3.1.1, to illustrate the storage performance. Figure 4.18 shows the λ s for the classifiers with different percentages of wildcards and 64-bit long bit vectors. When the classifier has no wildcards λ is around 64. So the storage can be approximately considered as linear to the number of dimension d and the number of rules N . However, when there is a larger percentage of wildcards, λ increases almost linearly with the size of classifiers.

Figure 4.18: λ s for different classifiers without the pre-group. The length of bit vectors is 64.

In terms of the memory access time, HBV is slightly better than ABV. For the same set of classifiers above, assuming the word-length is 64-bit, the query time performances of the ABV and HBV schemes are shown in Table 4.3. From this table the lookup behavior of the HBV scheme is as good as ABV for the small classifiers, and when the classifiers are large HBV is better than ABV. Note here we test all the possible input packet headers to find the worst case for the ABV scheme. While for the HBV scheme the query time depends on the number of levels of the space partition. For our tests in Table 4.3 the number of levels are

not greater than 4. Thus the total memory access time for two dimensions is not more than

8. The detailed results about levels of partition are shown in Appendix A.

Size	ABV0	HBV0	ABV10	HBV10	ABV20	HBV20	ABV30	HBV30	ABV40	HBV40	ABV50	HBV50
1K	4	4	6	4	6	4	8	4	8	6	8	6
2K	4	4	8	6	10	6	10	6	8	6	10	6
5K	6	6	12	6	12	6	10	6	12	6	12	8
10K	8	6	14	6	14	8	14	8	14	8	14	8
20K	8	6	16	8	16	8	16	8	16	8	14	8

Table 4.3: Worst case access times of ABV and HBV schemes. ABV0 and HBV0 denote the ABV and HBV algorithms with 0% wildcards, respectively. Here the bit vector length in HBV is 64 and the aggregation size in ABV is also 64. And the memory word is 64-bit wide. For the classifiers of 1K, 2K and 5K, ABV of 2 levels is used; and for classifiers of 10K and 20K ABV of three levels is used.

With Pre-group The pre-group can reduce the storage when there is a large percentage of wildcards in the classifier. For the same set of classifiers used in Figure 4.17, the storage performances of HBV with the pre-group are shown in Figure 4.19. The different percentages of wildcards have no significant influence on the required storage space when the pre-group is used. The six lines almost merge into a single curve in the figure. With the pre-group the storage is as if there is no wildcards in the classifier and the λ s for the classifiers are all around 64. Thus the storage complexity for HBV with the pre-group is almost $O(dN)$.

However, the number of memory access time increases when the pre-group is adopted. For our synthetic 2-dimensional classifiers there are always two sub-classifiers after the pre-grouping and the total query time is the sum of query time for each sub-classifier. Table 4.4 shows the result query time performance. Because there are only 2 or 3 levels

Figure 4.19: Storage of BV and HBV scheme with different percentages of wildcards. The pre-group is used and vector length is 64-bit.

of partition in each of pre-grouped sub-classifiers, the memory access time is not more than 12. Although the access time is larger than the cases without the pre-group, it is still competitive to the ABV scheme. In our tests only for the small classifiers (such as 1000 rules), ABV outperforms our pre-grouped HBV scheme. For larger classifiers our scheme is better than the ABV scheme.

size	ABV0	HBV0	ABV10	HBV10	ABV20	HBV20	ABV30	HBV30	ABV40	HBV40	ABV50	HBV50
1K	4	4	6	8	6	8	8	8	8	8	8	8
2K	4	4	8	8	10	8	10	8	8	8	10	8
5K	6	6	12	10	12	10	10	10	12	10	12	10
10K	8	6	14	10	14	10	14	10	14	10	14	10
20K	8	6	16	10	16	10	16	12	16	12	14	12

Table 4.4: Worst case access times of ABV and HBV schemes with the pre-group. ABV0 and HBV0 denote the ABV and HBV algorithms with 0% wildcards, respectively. Here the bit vector length in HBV is 64 and the aggregation size in ABV is also 64. And the memory word is 64-bit wide.

4.3.2.2 Different Partition Metrics

Figure 4.20: Storage of BV and HBV scheme with different cutting metrics. The length of bit vectors is 64 and there are 30% wildcards. The pre-group is not used.

Since the optimal partition of the space is almost impossible to obtain, we tried three cut operation metrics mentioned in Section 4.2.1.2. When there are 30% wildcards, Figure 4.20 shows the result of required memory space of BV and HBV scheme with different cutting metrics. We can see when there are wildcards the cutting metrics 3 performs worse than the other two. And the metric 1 and 2 have almost the same performances in the terms of storage. While when there are no wildcards, the three metrics have no significant difference, which is shown in Figure 4.21. For all the classifiers in this thesis we use cutting metric 1, if it is not stated specifically.

Figure 4.21: Storage of BV and HBV scheme with different cutting metrics. The length of bit vector is 64 and there are 0% wildcards.

4.3.2.3 Different Vector Lengths

Until now our experiments only evaluate the performance of HBV using the 64-bit wide vectors. However, the memory data bus width can be larger and wider vectors can be used. For example, in the BV scheme the width of bus up to 1000 bits is used to reduce the times of memory access. We test the influence of the vector length on our HBV scheme's storage performance, which is shown in Figure 4.22 and Figure 4.23. In Figure 4.22 there are no wildcards in the classifiers and in Figure 4.23 there are 30% of wildcards. It is interesting that when there are no wildcards, shorter vector length leads to smaller storage for HBV. It is reasonable because in each sub-classifier when the number of rules is only a part of maximum value (vector length), longer the bit vector is, more bits in the vectors are wasted, which results in the larger storage. When there are 30% wildcards the required storage spaces have no very significant difference, because longer vectors lead to less rule duplications during the space partition, which somehow neutralizes the negative influence

of the waste of bits in vectors.

Figure 4.22: Storage of BV and HBV scheme with different length vectors and 0% wildcards.

As for the query time, longer bit vectors can reduce the number of levels of partition, which is directly related to the memory access times. The detailed query times for 0% and 30% wildcards with different vector lengths are shown in Table 4.5 and Table 4.6. We can see when there is 0% wildcards the length of 64-bit vectors already makes the number of memory access times very small (≤ 6). Longer bit vectors can not improve the lookup performance that much. For the 20K-rule classifier only the vector length larger than 20K bits can reduce the memory access times down to 2.

Figure 4.23: Storage of BV and HBV scheme with different length vectors and 30% wildcards.

	HBV-16	HBV-32	HBV-64	HBV-128	HBV-256
1K	6	6	4	4	4
2K	8	6	4	4	4
5K	8	6	6	4	4
10K	8	6	6	6	4
20K	10	8	6	6	4

Table 4.5: Query time of HBV for 0% wildcards with different vector lengths. No pre-group is used.

	HBV-16	HBV-32	HBV-64	HBV-128	HBV-256
1K	8	6	4	4	4
2K	10	8	6	4	4
5K	12	8	6	6	4
10K	12	10	8	6	6
20K	14	10	8	6	6

Table 4.6: Query time of HBV for 30% wildcards with different vector lengths. No pre-group is used.

Chapter 5

Conclusions

By using the bit vectors in each dimension, the fundamental BV scheme [1] splits a multi-field packet classification problem into several simple 1-dimensional sub-problems. Together with the parallelism, this scheme is suitable for both hardware and software implementations. However, it doesn't scale to large classifiers. Both storage space and query time deteriorate with the increasing of the size of classifier.

Based on the BV, the ABV scheme [5] outperforms the BV scheme by using the aggregation of bit vectors and rule rearrangement. In the tests in [5] ABV is at least an order of magnitude faster than the BV scheme. It is also simple to be implemented, both in hardware and software: only an aggregation bit vector is added in each node in the tries of the BV scheme. Although the ABV scheme has faster lookup performance, the storage required is even slightly worse than the BV scheme, because of the addition of aggregation bit vectors.

This thesis introduces the concept of hierarchical search and space partition, and develops the Hierarchical Bit Vector (HBV) scheme based on the BV scheme. The HBV scheme

is more scalable in terms of both speed and storage. As for the storage, the HBV scheme requires only one or two orders of magnitude smaller memory than the BV scheme for the bit vectors. As for the classification speed, in our tests it is much faster than the BV scheme and as fast as the ABV scheme for the small classifiers. When the classifiers are large, HBV scheme begins to be faster than ABV scheme. The parallelism and pipeline can still be used in HBV to further accelerate the classification speed.

We synthesized our 2-dimensional classifiers to evaluate the new scheme. According to some observations on the real databases, we built the classifiers including different percentages of wildcards, which appears to cause worse behavior of HBV. For instance, for a 20K-rule classifier and 0% wildcards, if the vector length and memory data width is 64-bit, we can implement the data structure with 200K bytes for storing the bit vectors and 8 memory accesses for query. While if there are 50% wildcards, the storage goes up 100 times to 20M bytes and the time for memory access is still 8. For the same size classifier the BV scheme needs 100M bytes for storage and 625 memory accesses for query, and the ABV scheme needs more than 100M bytes and 18 memory accesses. In order to further reduce the storage for the intense wildcards, we propose the concept of the pre-group, which can make the required storage much smaller, as if there are no wildcards. However, the cost is slightly more memory accesses, slowing down the classification speed. Another factor influencing the performance of HBV is the length of bit vectors. Longer vectors can result in larger storage space and less memory accesses for the classifier without the presence of wildcards. And thus we can tune the vector length to balance between the storage and the query time.

Based on our experiments for large classifiers with heavy wildcards, the HBV scheme

with the pre-group and mid-size vector length can achieve fast classification speed with relatively small memory. Considering the parallelism and pipeline and wider memory bus, we believe our scheme can be fast enough for the line-speed classification.

Although we only evaluate our scheme on 2-dimensional classifiers, it can be extended to multi-field cases. The more possible cost may be more pre-grouped sub-classifiers before building the hierarchical structures. More analysis and tests of the performance of the HBV scheme on multi-field classifiers are needed in future work.

Bibliography

- [1] T.V. Lakshman, D. Stiliadis. “High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching.” Proceedings of ACM Sigcomm, 1998.
- [2] C. Su. “High-speed Packet Classification Using Segment Tree.” GLOBECOM’00. IEEE, Vol. 1, 2000 Page(s):582-586.
- [3] A. Feldman, S. Muthukrishnan. “Tradeoff for Packet Classification.” Proceedings of Infocom, vol.3, 2000.
- [4] M.M. Buddhikot, S. Suri, M. Waldvogel. “Space Decomposition Techniques for Fast Layer 4 Switching.” Proceedings of Conference of Protocols for High Speed Networks, 1999.
- [5] F. Baboescu, G. Varghese. “Scalable Packet Classification.” Sigcomm, 2001.
- [6] P. Gupta, N. McKeown. “Algorithms for Packet Classification.” IEEE Network, March/April 2001.
- [7] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel. “Fast and Scalable Layer Four Switching.” Proceedings of ACM SIGCOMM’98.

- [8] V. Srinivasan, S. Suri, G. Varghese. "Packet Classification Using Tuple Space Search." Proceedings of ACM SIGCOMM'99.
- [9] P. Gupta, N. McKeown. "Packet Classification on Multiple Fields." Proceedings of Sigcomm, Comp. Commun. Rev., vol. 29, no. 4, Sept. 1999, pp. 147-60.
- [10] P. Gupta, N. McKeown. "Classification Using Hierarchical Intelligent Cuttings." Proceedings of Hot Interconnects VII, Aug. 1999.
- [11] Z. Yu, J. Wu, K. Xu, M. Xu. "A Fast IP Classification Algorithm Applying to Multiple Fields." Communications, 2001. ICC 2001. IEEE International Conference on, Volume:7, Page(s):2058-2062, 2001.
- [12] T. Y. C. Woo. "A Modular Approach to Packet Classification: Algorithms and Results." INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, Vol. 3, 2000 Page(s):1213-1222.
- [13] J. Van Lunteren, A.P.J. Engbersen, "Multi-field packet classification using ternary CAM." Electronics Letters , Volume: 38 Issue: 1 , 3 Jan. 2002 Page(s): 21 -23
- [14] J. Van Lunteren, A.P.J. Engbersen, "Prefix-based parallel packet classification." IBM research report RZ 3210(#93256), 2000 (<http://domino.watson.ibm.com/library/cyberdig.nsf/Home>)
- [15] <http://www.merit.edu/ipma>
- [16] D. Shah, P. Gupta. "Fast Updating Algorithms for TCAMs." IEEE Micro, Vol. 21, Issue 1, Jan.-Feb. 2001.

- [17] H.M. Ji, R. Srinivasan. "Fast IP routing lookup with configurable processor and compressed routing table." Global Telecommunications Conference, 2001. GLOBE-COM'01, IEEE, Volume: 4, 2001.

Appendix A

Measurements of Partitions of Synthetic Classifiers

- Different Percentages of Wildcards

We generated a cluster of classifiers with different percentages of wildcards and fixed bit vector length, 64. The partitions of the spaces and classifiers are shown in the following tables about m_l and α_l .

	Level 0	Level 1	Level 2	Level 3
1K	1,000	25	1	
2K	2,000	46	1	
5K	5,000	123	4	1
10K	10,000	234	5	1
20K	20,000	467	11	1

Table A.1: m_l of partitioned sub-classifiers. There is no pre-group and no wildcards. The bit vector length is 64.

	Level 1	Level 2	Level 3
1K	1	1	
2K	1	1	
5K	1	1	1
10K	1	1	1
20K	1	1	1

Table A.2: α_l of partitioned sub-classifiers. There is no pre-group and no wildcards. The bit vector length is 64.

	Level 0	Level 1	Level 2	Level 3	Level 4
1K	1,000	38	1		
2K	2,000	75	2	1	
5K	5,000	288	7	1	
10K	10,000	853	30	1	
20K	20,000	2,679	94	3	1

Table A.3: m_l of partitioned sub-classifiers. There is no pre-group and 10% wildcards. The bit vector length is 64.

	Level 1	Level 2	Level 3	Level 4
1K	1.5650	1		
2K	1.8605	1	1	
5K	2.7680	1	1	
10K	4.1369	1	1	
20K	6.3766	1	1	1

Table A.4: α_l of partitioned sub-classifiers. There is no pre-group and 10% wildcards. The bit vector length is 64.

	Level 0	Level 1	Level 2	Level 3	Level 4
1K	1,000	48	1		
2K	2,000	130	5	1	
5K	5,000	620	17	1	
10K	10,000	2,025	70	2	1
20K	20,000	7,338	245	7	1

Table A.5: m_l of partitioned sub-classifiers. There is no pre-group and 20% wildcards. The bit vector length is 64.

	Level 1	Level 2	Level 3	Level 4
1K	2.2480	1		
2K	3.2560	1	1	
5K	5.9878	1	1	
10K	9.8598	1	1	1
20K	17.8725	1	1	1

Table A.6: α_l of partitioned sub-classifiers. There is no pre-group and 20% wildcards. The bit vector length is 64.

	Level 0	Level 1	Level 2	Level 3	Level 4
1K	1,000	61	1		
2K	2,000	187	6	1	
5K	5,000	1,015	31	1	
10K	10,000	3,921	135	5	1
20K	20,000	13,203	400	10	1

Table A.7: m_l of partitioned sub-classifiers. There is no pre-group and 30% wildcards. The bit vector length is 64.

	Level 1	Level 2	Level 3	Level 4
1K	2.8180	1		
2K	4.7315	1	1	
5K	9.8866	1	1	
10K	19.1588	1	1	1
20K	32.6375	1	1	1

Table A.8: α_l of partitioned sub-classifiers. There is no pre-group and 30% wildcards. The bit vector length is 64.

	Level 0	Level 1	Level 2	Level 3	Level 4
1K	1,000	84	3	1	
2K	2,000	289	8	1	
5K	5,000	1,578	45	1	
10K	10,000	5,793	201	6	1
20K	20,000	19,750	578	15	1

Table A.9: m_l of partitioned sub-classifiers. There is no pre-group and 40% wildcards. The bit vector length is 64.

	Level 1	Level 2	Level 3	Level 4
1K	4.0670	1	1	
2K	7.2315	1	1	
5K	15.3742	1	1	
10K	28.3093	1	1	1
20K	49.3953	1	1	1

Table A.10: α_l of partitioned sub-classifiers. There is no pre-group and 40% wildcards. The bit vector length is 64.

	Level 0	Level 1	Level 2	Level 3	Level 4
1K	1,000	96	4	1	
2K	2,000	348	9	1	
5K	5,000	2,085	65	2	1
10K	10,000	7,869	258	7	1
20K	20,000	27,612	771	25	1

Table A.11: m_l of partitioned sub-classifiers. There is no pre-group and 50% wildcards. The bit vector length is 64.

	Level 1	Level 2	Level 3	Level 4
1K	4.7670	1	1	
2K	8.8615	1	1	
5K	20.3660	1	1	1
10K	38.5974	1	1	1
20K	69.1298	1	1	1

Table A.12: m_l of partitioned sub-classifiers. There is no pre-group and 50% wildcards. The bit vector length is 64.

- **Different Vector Length**

With a fixed percentage of wildcards, 30%, we also generated another cluster of classifiers with different bit vector lengths, from 16 to 256 bits, to test their influences on the performance. The following tables show the detailed partitions of the spaces and classifiers. Since the case with 64-bit long bit vectors and 30% wildcards is already shown above, it is omitted here.

	Level 0	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7
1K	1,000	622	74	8	1			
2K	2,000	2,856	317	37	5	1		
5K	5,000	15,083	1,615	200	24	3	1	
10K	10,000	54,849	6,418	753	86	9	1	
20K	20,000	20,1373	23,458	2,421	273	31	4	1

Table A.13: m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 16.

	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7
1K	7.4390	1	1	1			
2K	16.4915	1	1	1	1		
5K	36.3254	1	1	1	1	1	
10K	67.1720	1	1	1	1	1	
20K	124.0581	1	1	1	1	1	1

Table A.14: α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 16.

	Level 0	Level 1	Level 2	Level 3	Level 4	Level 5
1K	1,000	190	12	1		
2K	2,000	785	57	5	1	
5K	5,000	4,192	245	12	1	
10K	10,000	15,186	817	48	3	1
20K	20,000	51,527	2,893	158	9	1

Table A.15: m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 32.

	Level 1	Level 2	Level 3	Level 4	Level 5
1K	4.5390	1	1		
2K	8.8900	1	1	1	
5K	19.4338	1	1	1	
10K	36.1843	1	1	1	1
20K	63.0041	1	1	1	1

Table A.16: α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 32.

	Level 0	Level 1	Level 2	Level 3
1K	1,000	32	1	
2K	2,000	63	1	
5K	5,000	302	6	1
10K	10,000	1,112	15	1
20K	20,000	4,120	62	1

Table A.17: m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 128.

	Level 1	Level 2	Level 3
1K	2.2960	1	
2K	2.9695	1	
5K	5.7208	1	1
10K	10.2133	1	1
20K	18.8192	1	1

Table A.18: α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 128.

	Level 0	Level 1	Level 2	Level 3
1K	1,000	11	1	
2K	2,000	32	1	
5K	5,000	89	1	
10K	10,000	302	2	1
20K	20,000	1068	7	1

Table A.19: m_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 256.

	Level 1	Level 2	Level 3
1K	1.8330	1	
2K	2.3500	1	
5K	3.3604	1	
10K	5.7265	1	1
20K	9.8027	1	1

Table A.20: α_l of partitioned sub-classifiers. There is no pre-group and 30 % wildcards. The bit vector length is 256.