MEMORY ORGANIZATION IN PIPELINED HIERARCHICAL SEARCH STRUCTURES FOR PACKET CLASSIFICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÇAĞLA IRMAK RUMELİLİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JUNE 2013

Approval of the thesis:

MEMORY ORGANIZATION IN PIPELINED HIERARCHICAL SEARCH STRUCTURES FOR PACKET CLASSIFICATION

submitted by **ÇAĞLA IRMAK RUMELİLİ** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Engineering** _____

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Supervisor, **Electrical and Electronics Eng. Dept., METU** _____

Assist. Prof. Dr. Oğuzhan Erdem
Co-supervisor, **Electrical and Electronics Eng. Dept., Trakya Univ.** _____

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Gözde B. Akar
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Ece Schmidt
Electrical and Electronics Engineering Dept., METU _____

Dr. Atilla Özgit
Computer Engineering Dept., METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ÇAĞLA IRMAK RUMELİLİ

Signature :

ABSTRACT

MEMORY ORGANIZATION IN PIPELINED HIERARCHICAL SEARCH STRUCTURES FOR PACKET CLASSIFICATION

Rumelili, Çağla Irmak
M.Sc., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Co-Supervisor : Assist. Prof. Dr. Oğuzhan Erdem
June 2013, 46 pages

Packet classification is a main requirement in routers to manage network security and traffic. In high speed networks packet classification in line rates has become a major challenge. Our design mainly benefits from a parallel pipelined architecture implemented on field programmable gate arrays (FPGA) to achieve high speed packet processing. The presented solution is based on Hierarchical Hybrid Search Structure (HHSS) [5]. Our work solves the deep pipeline problem of HHSS in a memory efficient way. This study has focused on changing the memory structure of HHSS to decrease its latency without increasing its memory requirement or decreasing its throughput. The use of memory blocks with variable word lengths on the trie structure has decreased the tree depth while preserving the throughput and memory storage requirement values. Our design uses a parallel pipelined architecture implemented on FPGA in order to achieve high speed packet processing. The proposed algorithm supports approximately 128 Gbps throughput and can handle 10K rules with only 28 KB memory requirement. Comparing with the state of art packet classification algorithms, our design offers a significant performance without long latency of packet processing.

Keywords: Packet Classification, FPGA, Pipeline, SRAM, Reconfigurable Architecture

ÖZ

PAKET SINIFLANDIRILMASI İÇİN BORUHATTINDA HİYERARŞİK ARAMA YAPILARINDA BELLEK ORGANİZASYONU

Rumelili, Çağla Irmak
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç Dr. Cüneyt Bazlamaçcı
Ortak Tez Yöneticisi : Yar. Doç. Dr. Oğuzhan Erdem
Haziran 2013, 46 sayfa

Paket sınıflandırılması ağ güvenliği ve ağ trafiğinin yönetiminde en temel gereksinimlerdendir. Yüksek hızlı ağlarda, ağdaki veri hızıyla baş edebilecek sürelerde paket sınıflandırılması üstesinden gelinmesi gereken önemli bir meseledir. Bu çalışma, yüksek hızlı paket sınıflandırması yapabilmek amacı ile temel olarak Alanda Programlanabilir Kapı Dizileri'nde (APKD) yer alan paralel ardışık dizin mimarisini kullanmaktadır. Önerilen çözümde Hierarchical Hybrid Search Structure (HHSS) [5] adlı çalışmada önerilen yöntem temel alınmıştır. Bu tezde HHSS tarafından sunulan çözümde yer alan derin boruhattı problemi bellek daha verimli bir biçimde kullanılarak çözülmüştür. Bunun için, bellek üzerinde değişken genişlikte bloklar kullanılarak gecikme süresi kısaltılmış; aynı zamanda bellek ihtiyacı ve işlem hacmi korunmuştur. Yüksek işlem hacmi için APKD üzerinde gerçeklenmiş bir boru hattı mimarisinden yararlanılmıştır. Bu tezde önerilen yöntem, minimum paket boyutu 40 byte olarak kabul edildiğinde, 128 Gbps işlem hacmine ulaşabilmekte ve 10K boyutunda bir kural kümesi için sadece 28 KB'lık bir depolama alanına ihtiyaç duymaktadır. Literatürde var olan önemli ve en üstün çalışmalarla karşılaştırıldığında, yüksek performansa erişilirken, bu tip paket işleme uygulamalarında gecikme süresi düşük seviyelerde tutulmaktadır.

Anahtar Kelimeler: Paket Sınıflandırması, APKD, Boruhattı, SRAM, Yeniden Programlanabilir Mimari

To My Family

ACKNOWLEDGMENTS

In the first place I would like to express my gratitude to my supervisor Assoc. Prof. Dr. Cüneyt Bazlamaçcı and my cosupervisor Assist. Prof. Dr. Oğuzhan Erdem, for their guidance and support in supervision of this thesis. Working with them has always been a pleasure.

I would like to express my special thanks to TÜBİTAK.

I would like to thank to my employer, ASELSAN.

I can not thank enough to my collegues for their support and feedbacks during my master study. Their presence made this study more enjoyable.

I would like to thank to Emin Yiğit Köksal for his encouragement and support. He has been an indispensable part of my life while I was working on this thesis.

Last but not least, I am greateful to my family for their continuous love, trust and for their sincere and endless support at every moment througout my life and providing me every opportunity they can, especially for my education. Their encouragement made me believe that I can achieve anything.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

ACL	Access Control List
APKD	Alanda Programlanabilir Kapı Dizileri
BRAM	Block Random Access Memory
BS	Bit String
BV	Bit Vector
BV-TCAM	Bit Vector - Ternary Content Addressable Memory
DA	Destination Address
DCFL	Distributed Crossproducing of Field Labels
DP	Destination Port
EGT	Extended Grid of Tries
EN	Empty Node
FPGA	Field Programmable Gate Array
FW	Firewall
Gbps	Gigabit per second
GoT	Grid of Tries
H-Trie	Hierarchical Trie
HD	High Definition
HHSS	Hierarchical Hybrid Search Structure
IP	Internet Protocol
IPC	Internet Protocol Chain
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISE	Integrated Simulation Environment
ISP	Internet Service Provider
K	Kilo
KB	Kilo Byte

PC	Path Compression
PRTCL	Protocol
PTR	Pointer
QoS	Quality of Service
RAM	Random Access Memory
RLE	Recursive Leaf Extraction
SA	Source Address
SP	Source Port
SV	Skip Value
SRAM	Static Random Access Memory
TCAM	Ternary Content Addressable Memory
TTϵ	Tree Trie Epsilon

CHAPTER 1

INTRODUCTION

1.1 Packet Classification

Recently, the Internet has been used for real time applications such as video conferencing, HD video streaming over web and other voice over Internet Protocol (IP) processes that require certain delay guarantees. This state of the Internet leads internet service providers (ISP) to offer better service for real time applications. Hence, quality of service (QoS) levels in terms of delay and jitter are defined for different types of packets. In order to achieve predefined timing limitations, routers have to manage resources. Packet scheduling and buffer management algorithms to decide which packet is served with priority, admission control algorithms to limit the load in a system to preserve enough resource for e services, and many other sophisticated functions and algorithms are proposed for appropriate resource allocation.

In order to maintain resource allocation mechanisms, routers need to distinguish packets from each other and then make an action to satisfy the needs of QoS such as dropping or forwarding a packet accordingly. In other words, the traffic has to be classified into flows using predefined filters and then the corresponding action is taken. This process is called packet classification or packet filtering interchangeably. The routers that have the ability to classify packets into predefined flows based on the needs of the ISPs are called flow aware routers [7]. The flows, more commonly known as rules or filters are identified by the context of packets. The IP header comprising source address (SA), destination address (DA), source port (SP), destination port (DP) and the protocol (PRTCL) fields of an incoming packet is compared with the rules in a rule set to identify its flow. In a simple rule, the addresses are represented as prefixes while port numbers are specified as ranges and protocol field is a fixed length number.

Packet classification problem in flow aware routers has been studied for a long time. However, it still attracts a great deal of attention from the researchers because of the fast growth of the Internet. The need for fast and efficient algorithms arises since the bandwidth of the network doubles in every nine months [21]. The growth of the network bandwidth requires to provide admission control, traffic management, packet scheduling and network security for complex flows. Furthermore, the line rate of the Internet is increasing due to the new advancements in optical technology. Therefore, packet classification algorithms must be accommodated by suitable hardware to achieve packet processing in

line rates. Due to all these reasons stared, packet classification is still a major problem that craves for novel solutions.

1.2 Performance Challenges

1.2.1 Speed

Speed or throughput is the major performance metric that is used to evaluate packet classification algorithms. Packet classification requires fast processing to support the speed of the Internet which currently reached beyond 40Gbps [12]. In other words, an incoming IP datagram has to be processed in 8ns (assuming minimum sized 40 bytes IP packets). In order to achieve this goal, the hardware has to be as simple as possible since the complexity results in extra processing time [4].

1.2.2 Memory

In addition to speed, memory requirement is another important metric for packet classification. The memory efficiency is defined as the storage requirement per rule [5]. The hardware resources are limited and should be used efficiently to serve larger rule sets. Additionally, larger memory needs extra power consumption and extra processing time. Therefore, memory efficient algorithms are required to achieve fast and power efficient results.

1.2.3 Scalability

The size of rule sets in a router is increasing due to the growth of internet. Therefore, the proposed algorithms has to be scalable with respect to the number of rules in filter (rule) sets. In addition to scalability in the number of rules, scalability in the size and the number of header fields of a rule are also important for classification engines. The latest revision of the internet protocol IPv6 with 128 bits of address fields and next generation OpenFlow switches with 12 header fields for each rule necessitate scalable solutions for efficient packet classification [17], [6].

1.2.4 Latency

Latency is another major metric of a packet classification algorithm. The total time of packet processing in a router is called latency. Throughput and latency are different metrics and an algorithm might have a high throughput while having a high latency value. For example, pipelined algorithms achieve good results for the throughput. The processing time of one pipeline stage determines the throughput value. On the other hand, latency is the time from the packet enters the pipeline until it leaves. In other words, latency is equal to the number of pipeline stages multiplied by the processing time of a single pipeline stage. Deep pipeline in decision based algorithms increases the latency which eventually has

a negative effect on system synchronization of real time applications. Therefore, low latency is as significant as high throughput for delay guaranteed services.

1.2.5 Updates

Last but not least, update performance is another metric to evaluate packet classification. The rule tables alter in time with rule insertions, deletions and modifications. The algorithm should support the updates in a timely manner in order to ensure reliable packet classification.

1.3 Motivation

Most of the recent studies on packet classification utilize SRAM based parallel architectures and benefit from prior algorithms in the literature. These architectures achieve high throughput by utilizing pipelining techniques. On the other hand, the memory efficiency of existing algorithms has to be improved. Decision tree based solutions that employ cutting algorithms and hierarchical tree structures with several optimizations increase memory efficiency. However, it is observed that cutting based algorithms (e.g. [11], [14]) suffer from rule overlaps resulting in high storage requirement for several rule sets. On the other hand, Hierarchical Hybrid Search Structure (HHSS) [5] as a representative of hierarchical tree structures offers a significant improvement on memory efficiency. However, it is observed that the numerous pipeline stages that cause high latency in this packet processing task is a major problem.

1.4 Contributions

This thesis focuses on designing a fast and efficient algorithm to classify packets on flow aware routers to increase the number of packets classified per unit time with minimum storage requirement and low latency value, which becomes a fundamental challenge with the rapid growth of the Internet. In order to achieve this goal, the Hierarchical Hybrid Search Structure is used. The latency problem of the HHSS algorithm is solved by using multiple but different sized BRAMs on the hardware structure. The algorithm that is proposed in this work achieved high throughput of approximately 128 Gbps and low storage requirement which is only 28 KB for 10K rule set while offering a drastically decreased latency value.

1.5 Thesis Organization

The remainder of the thesis is structured as follows. Chapter 2 presents the prior algorithms used to solve the packet classification problem. Chapter 3 explains the contribution of the present work to the existing packet classification methodologies and introduces our proposed algorithm. Chapter 4 presents the implementation details while the performance of our work is compared with other state

of the art algorithms in Chapter 5. Finally, Chapter 6 provides a summary of the work completed and presents some future directions.

CHAPTER 2

LITERATURE OVERVIEW

2.1 Prior Work

Over the past several years, numerous packet classification algorithms have been presented [9] [27]. Decision tree based and decomposition based algorithms have been considered as two major techniques for the packet classification.

2.1.1 Decomposition Based Algorithms

Decomposition based algorithms make search in each field independently and then combine the results afterwards. Bit Vector [16] and Distributed Crossproducing of Field Labels [26] algorithms are two of the main representatives of these algorithms.

2.1.1.1 Bit Vector

In Bit Vector Algorithm (BV) [16] each field of the rule set is searched independently in parallel. This algorithm benefits from the geometrical representation of the header fields and bit vectors are used to keep a record of matching regions. As it is seen in figure 2.1, the rules are first geometrically placed on a coordinate system in which each axis represents a rule field [16]. For each dimension, the axis is split into regions formed by the edges of the range of header fields. Each region is specified with a bit vector and the vector is completed by placing '1's for the match cases and '0's otherwise.

The search is done with range lookups and the matching vectors of each field is combined with AND operation. Since the vectors are sequenced according to priority, most significant '1' indicates the best match in the resulting vector. Assume that there are N rules and each rule has k dimensions. In the worst case N rules divide a dimension into $2N + 1$ sub-regions. $(2n + 1) * n$ memory is required for one dimension. This leads to $k * n * (2n + 1)$ storage requirement in total. Briefly, substantial storage is required in BV algorithm which is feasible in small rule sets, however, this method is not scalable to large number of tables.

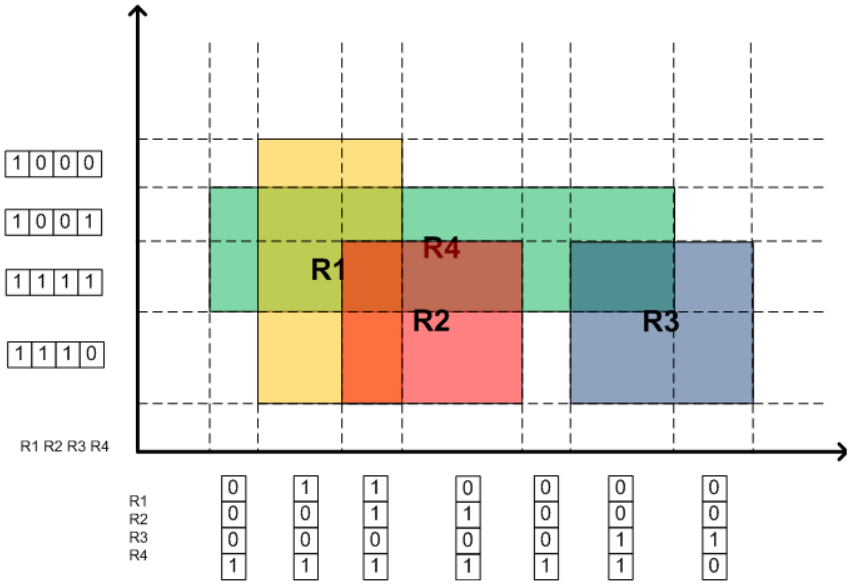


Figure 2.1: Bit Vector algorithm.

2.1.1.2 Distributed Crossproducing of Field Labels

Similar to BV algorithm, Distributed Crossproducing of Field Labels (DCFL) [26] also benefits from a parallel search structure on each field. The algorithm constructs multi-bit tries for IP addresses while Bloom Filters are used for other header fields. The key focus of DCFL algorithm is the structure of rule sets and the aggregation process.

The algorithm states that the number of unique prefixes for each header is small compared to the total number of prefixes for each header. Therefore, DCFL stores only unique prefixes to reduce the memory storage requirement. More importantly, DCFL combines the search results in a distributed fashion unlike BV algorithm where all search results are combined at once. For instance, after each independent field search is complete, the result of the first and second fields are combined while the similar process is done for the third and forth in parallel. The aggregation process is completed by integrating the combined results. The main motivation behind the distributed crossproducing is to reduce search time. The number of combined headers are smaller than the actual rule set size because the different rules might share the same header prefixes. For instance, SA and DA fields might be exactly the same for more than one rules. However, storing only one SA and DA pair is adequate in the aggregation process. In addition to the reduction of the number of combined and unique headers, the number of matching headers are remarkably small (less than five) for an incoming packet which reduces the complexity of the aggregation process.

2.1.2 Decision Based Algorithms

In decision based algorithm, the rules are constructed on a tree or a trie structure. The incoming packet traverses the tree structure until it finds the most prior match. Decision based algorithms offer memory efficient solutions compared to decomposition based algorithms due to its tree structure [27]. Also, the decision tree structure allows incremental updates. Moreover, since the tree traversal is suitable for pipelining, high throughput results are gained [13]. Hicuts [8], Hypercuts [22] and hierarchical tree structure algorithms have been the most sophisticated representatives of decision based solutions.

2.1.2.1 Hicuts

Hicuts [8] algorithm uses all the headers (SA, DA, SP, DP, PRTCL) to construct a tree. The rules are first placed on a hypercube that each header field represents one dimension. The decision tree is structured in a way that each branching leads to fewest number of rules that many rules with different specifications are gathered in child nodes by considering the distribution of rules on this hypercube. Each branching is called a cut since it cuts the main hypercube to smaller ones. The cutting operation is done in one dimension but more than one cuts are allowed in each stage. The number of cuts in each level is limited with the ”spfac” parameter. The algorithm that is proposed in Hicuts chooses the dimension to cut and the number of cuts to be made. The cutting operation continues until the number of rules in all of the nodes are smaller than the bucket size (”binth parameter”) that is the threshold for the number of rules in a leaf node.

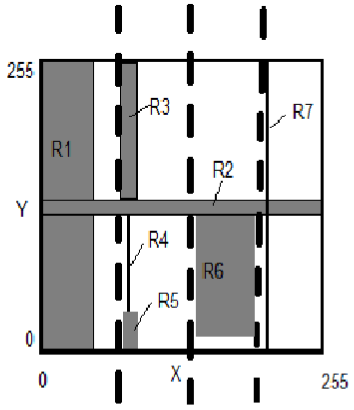
An example rule table that only contains two header fields, its geometric representation and tree structure is illustrated in figure 2.2 [8]. The bucket size is assumed as two for this example. The first branching is done by cutting the x dimension into four equal pieces. Then the y dimension is cut into the two equal pieces. Since every leaf node has less than two rules, the cutting operation is completed.

After the decision tree is structured, the search mechanism is easier. The incoming packet traverses the tree until it finds a leaf node, then starts linear searching. The linear search is ended when a suitable match is found.

2.1.2.2 Hypercuts

Hypercuts [22] algorithm is similar to the Hicuts algorithm since they both represent the rule set as a hypercube and define cutting algorithms to achieve smaller sub-spaces with fewer rules. The main difference between the Hicuts and the Hypercuts algorithms is the aspect of cutting on multiple dimensions in hypercuts. Figure 2.3 illustrates the difference between the two algorithms [22]. The aspect of cutting in multi dimensions results in compact tree structures that leads to lower latency values.

Rule	Xrange	Yrange
R1	0 - 31	0 - 255
R2	0 - 255	128 - 131
R3	64 - 71	128 - 255
R4	67 - 67	0 - 127
R5	64 -71	0 - 15
R6	128 - 191	4 - 131
R7	192 - 192	0 - 255



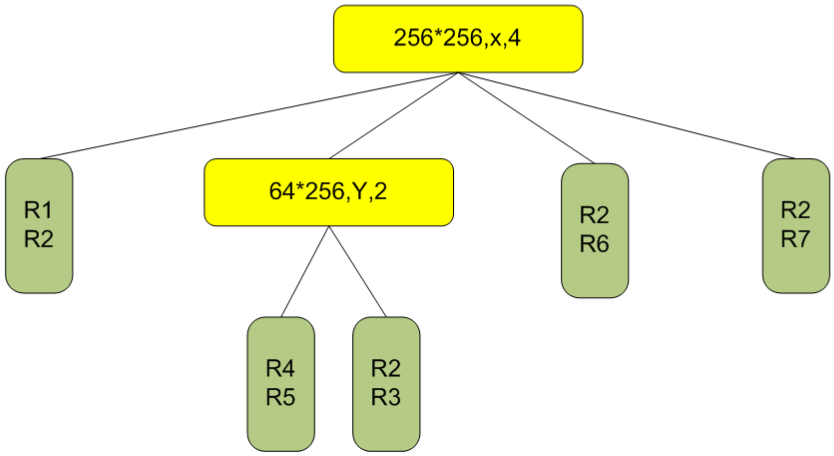


Figure 2.2: Hicuts.

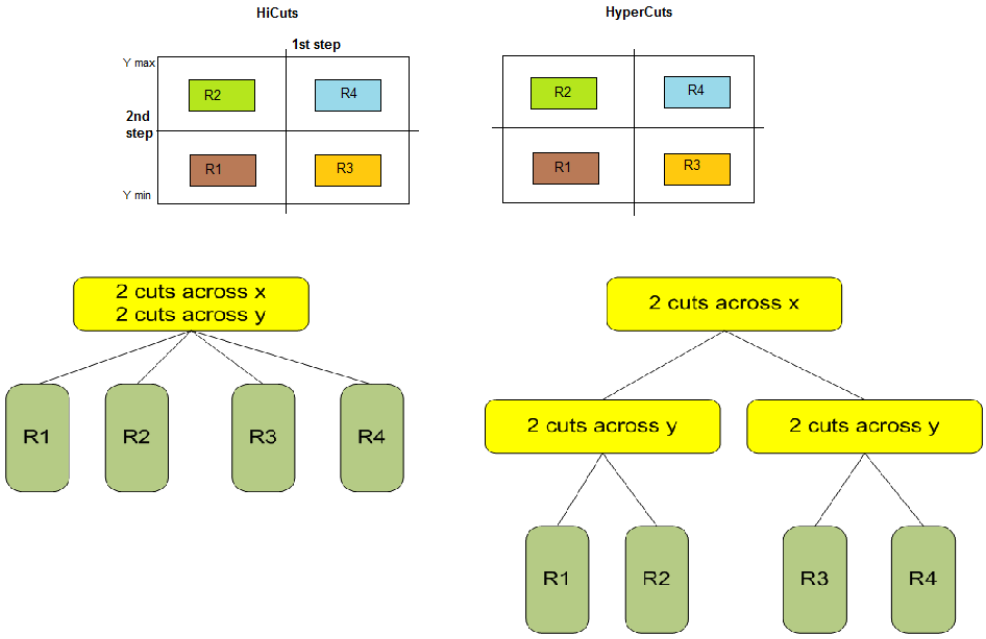


Figure 2.3: Hicuts vs. Hypercuts.

2.1.2.3 Hierarchical Structures

The hierarchical structures enable building a tree or trie by using rule fields hierarchically. For instance, the DA field of the rules are used to built the tree and then each DA node in the tree is connected to an SA tree where each SA prefix in the tree corresponds to a rule with the related DA. An incoming packet traverses the trees by comparing related bit(s) with the node until the best match is found. The major problem of these algorithms is to find the best match since there might be several match cases in different sub trees with different priorities. For example, an incoming packet with DA= 000, SA= 100 is a match for F1, F3, F4, F8, F9 and F10 filters in the example rule set in table 2.1. Therefore, the incoming packet has to traverse all the SA trees connected with 00* and 0* DA nodes. The backtracking on the tree structure both creates a complexity on hardware and causes non-deterministic delay.

Cecilia tries, most commonly known as Set Pruning Tree [28] offers an algorithm where several nodes are stored for one rule in order to eliminate backtracking. Figure 2.4 illustrates an example set pruning trie corresponding to the rule set in table 2.1 [27]. Although the backtracking problem is eliminated, memory storage requirement of the algorithm is exponentially increased. Due to rule duplication in several nodes, it is not scalable and substantial memory requirement prevents the use of Set Pruning Tree for large rule sets.

Another algorithm that solves the backtracking issue is The Grid of Tries [24] which proposes use of switch pointers. Switch pointers in a trie structure allows jumping to appropriate node without traversing the tree from the beginning as it is demonstrated in figure 2.4. As a result, both search time and

Table 2.1: Sample rule set [27].

Filter	DA	SA	DP	SP	PR
F1	0*	10*	*	80	TCP
F2	0*	01*	*	80	TCP
F3	0*	1*	17	17	UDP
F4	00*	1*	*	*	*
F5	00*	11*	*	*	TCP
F6	*	00*	*	*	*
F7	0*	10*	*	100	TCP
F8	0*	1*	17	44	UDP
F9	0*	10*	80	*	TCP

the memory requirement is reduced compared to Set Pruning Tree. Since the tree structure is properly utilized the algorithm is promising with its overall performance including throughput, memory, scalability and latency. However, the algorithm does not cover how to handle the rest of headers which are SP, DP and protocol. Also, the algorithm leads the incoming packet only to longest prefixes, which is not necessarily the best matching rule in all cases.

In addition to set pruning tree and Grid of Tries algorithms, the Extended Grid of Tries (EGT) [2] offers another solution for packet classification as an example of hierarchical tree structures. EGT is similar to GoT in the structure except its covering of all five headers and its searching for best matching rules. The switch pointers in GoT are replaced with jump pointers in EGT. The jump pointers directs an incoming packet to any possible match case. Figure 2.5 illustrates the tree structure of EGT corresponding to the rule set in table 2.1 [27]. EGT offers a complete solution compared to other hierarchical tree methods. However, backtracking problem on this type of tree still remains and causes high latency value due to numerous memory accesses.

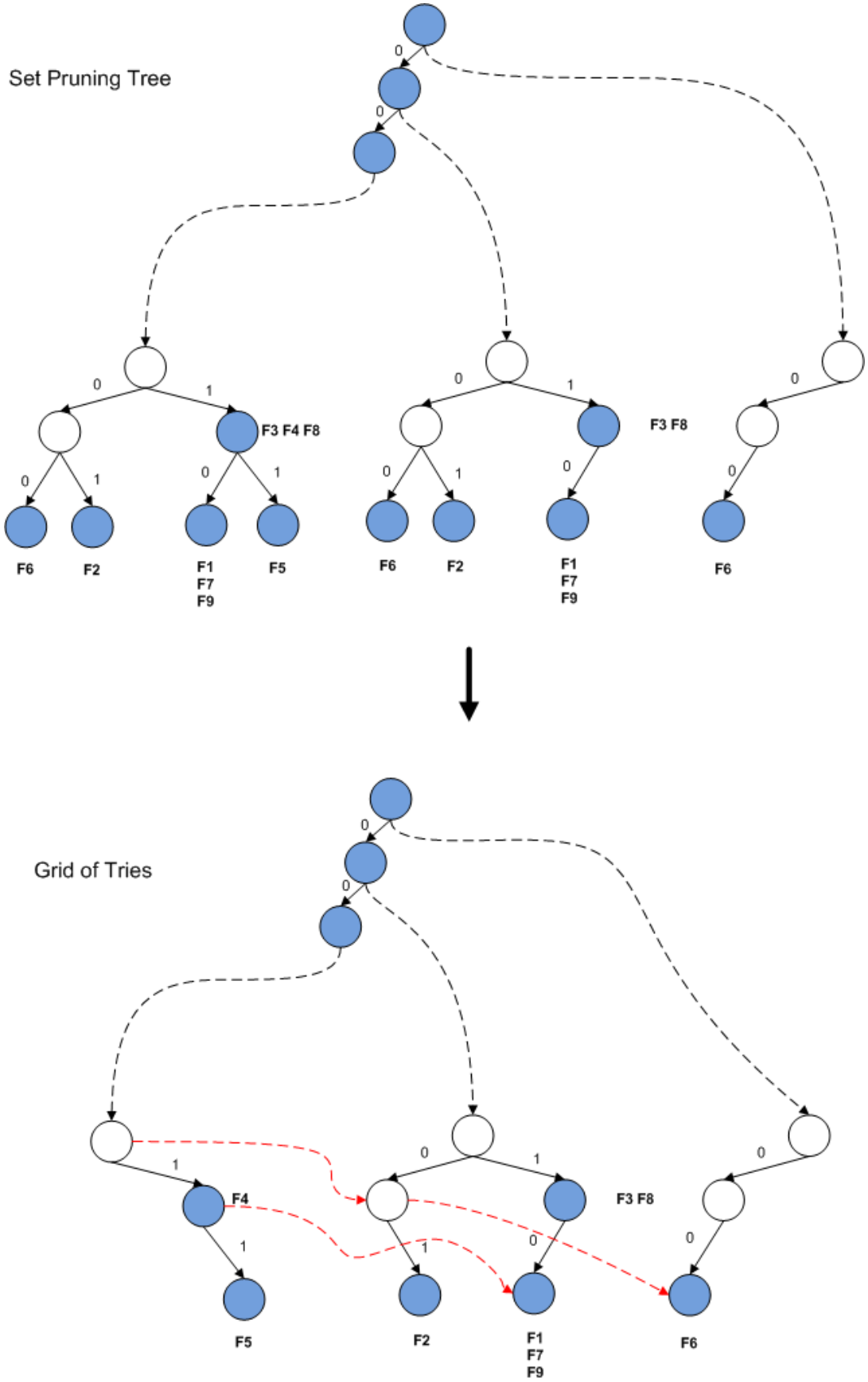


Figure 2.4: Set Pruning Tree and Grid of Tries.

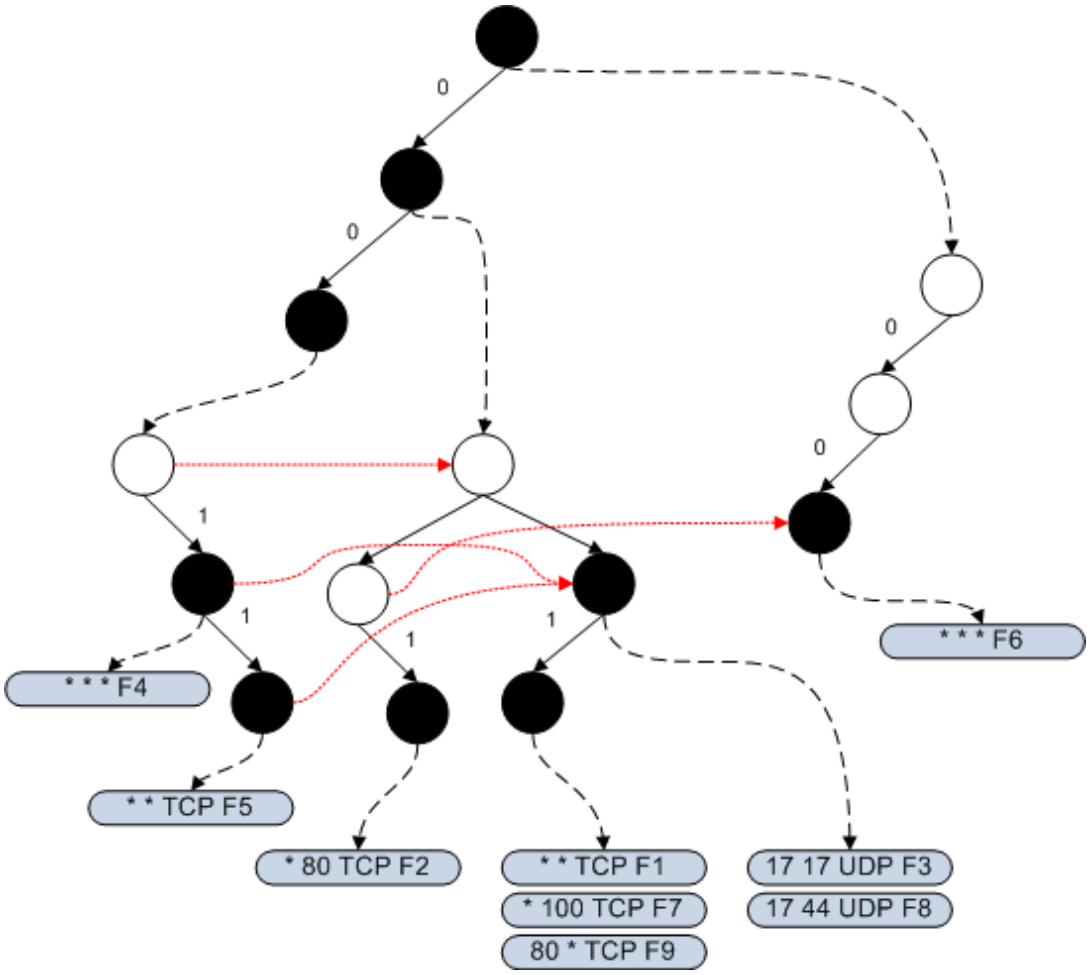


Figure 2.5: Extended Grid of Tries.

CHAPTER 3

MEMORY ORGANIZATION IN PIPELINED HIERARCHICAL SEARCH STRUCTURES FOR PACKET CLASSIFICATION

3.1 Hierarchical Hybrid Search Structure for High Performance Packet Classification

To achieve high throughput in packet classification, the hardware that supports the proposed algorithm should be simple and easy to implement. For instance, trie algorithms that simply determine the next node by making one bit inspection can easily be realized with a simple hardware and higher throughput can be achieved when compared to other designs. Although trie structures offer high throughput, they usually suffer from backtracking, which makes the search process difficult and complex as it was discussed in chapter 2.

The Hierarchical Trie (H-Trie) [5] is structured using SA and DA prefixes. First, all SA prefixes in the rule set are stored in a trie. Then, DA prefixes are connected to related SA nodes via DA tries. As a result, hierarchically connected SA and DA tries are built. When a packet comes, it traverses H-trie starting from SA trie. When a SA prefix is reached in the trie, the search continuous on DA trie. However, even if there is a match, the packet has to backtrack to SA trie and continue the search to find a prior match.

Table 3.1 shows a sample rule set for packet classification. In this table, the SA prefix of R4 and R5 (00*) is a descendant of SA prefix of R1, R2, R3, R8, R9 and R10 (0*). Moreover, these prefixes are both descendants of the SA prefix of R7 (*). When a packet whose SA prefix starting with ”00” comes, search has to traverse all three sub-branches. Even if a match condition occurs somewhere, the search operation should continue by backtracking since there might be a higher priority match.

Figure 3.1 illustrates the backtracking of an incoming packet on H-Trie and clearly indicates all the paths that it follows in a proper search [5]. In this example, the incoming packet has SA value of ”000” and DA value of ”110”. Since the SA of this packet matches all the nodes on the path, the corresponding DA nodes must be visited.

HHSS for high performance packet classification solves the backtracking issue on tree/trie structures [5]. This work emphasizes how backtracking negatively affects the search process specifically for the pipelined architectures. The pipeline has to be stalled while backtracking and this leads to unpre-

Table 3.1: Sample rule set [5].

Rule	SA	DA	SP	DP	Protocol	Priority	Action
R1	0*	10*	80	*	TCP	1	Act0
R2	0*	01*	17	17	UDP	2	Act1
R3	0*	1*	44	*	TCP	2	Act2
R4	00*	1*	17	44	UDP	3	Act3
R5	00*	11*	*	100	TCP	4	Act4
R6	10*	1*	*	*	*	5	Act5
R7	*	00*	*	*	TCP	5	Act6
R8	0*	10*	*	100	TCP	6	Act7
R9	0*	1*	*	*	TCP	7	Act8
R10	0*	10*	17	17	UDP	7	Act9
R11	111*	000*	80	*	TCP	8	Act10

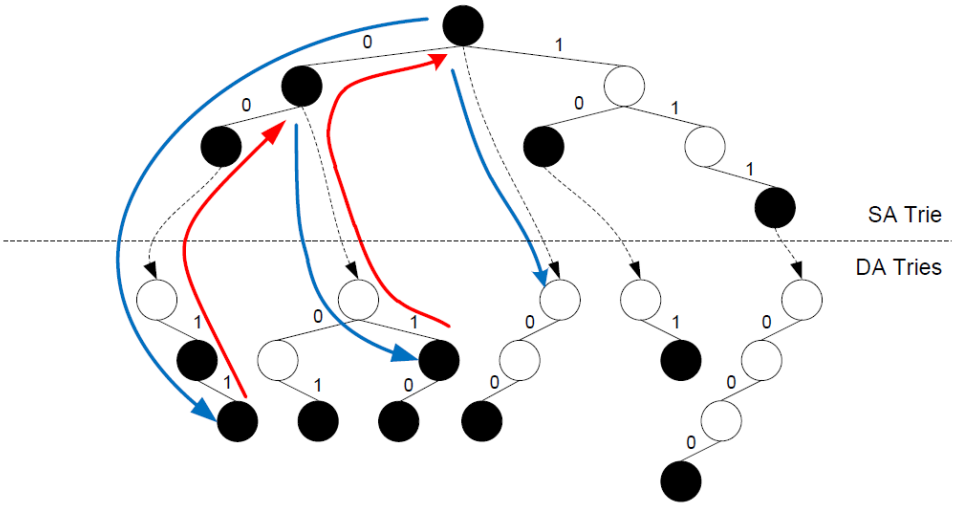


Figure 3.1: Backtracking in H-trie structure.

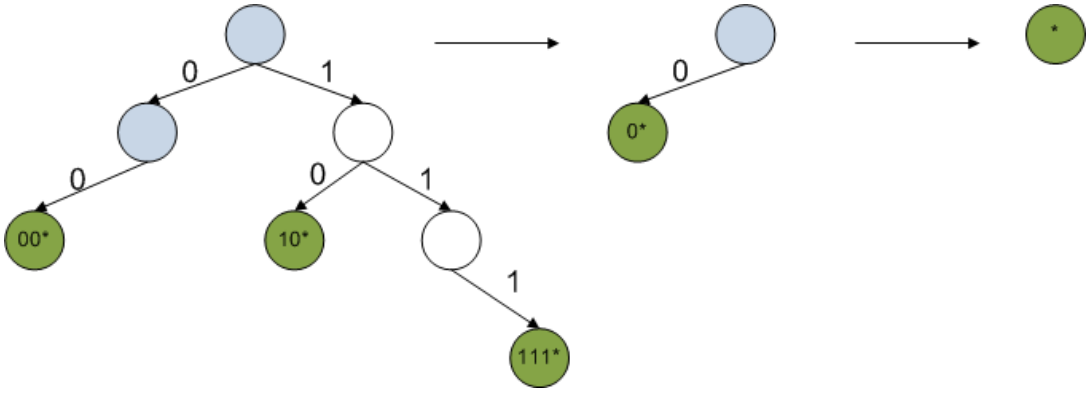


Figure 3.2: Clustering scheme.

dictable and possibly unacceptably long delay.

The essence of the idea in HHSS is to search each of these DA branches in parallel which will completely prevent the need for backtracking. In order to achieve this goal a clustering algorithm is proposed. This algorithm simply divides the SA tree in subtrees such that the nodes of each subtree are pair wise disjoint; in other words the nodes in the subtrees are not subsets of each other. All these sub-trees are searched in parallel. Since backtracking is eliminated, the resulting search structure can easily be implemented on pipelined hardware.

HHSS uses the recursive leaf extraction (RLE) algorithm in order to divide the SA tree into sub-trees each containing disjoint SA prefixes. In RLE, first all SA prefixes in the rule set are placed into a binary trie (see figure 3.2 [5]) . The leaves of the trie are moved to binary search tree. The empty leaves of the remaining trie are removed. These two operations are performed recursively until reaching the root node. (see Algorithm 1.)

Algorithm 1 Clustering Algorithm [5]

1: **procedure** CLUSTERING **Input:**Prefix set S. Number of clusters R.
2: **Output:**A partition of S into a collection of non-empty prefix subsets such that within each subset all the prefixes are pairwise disjoint.
3: $i = 0$
4: Construct a binary trie using prefix set S.
5: **while** $i \leq R - 1$ **do**
6: Move the leaves of the trie into S_i
7: Trim the leaf-removed trie.
8: $i = i + 1$
9: **end while**
10: Leaf-push the trie and move the leaf-pushed leaves into S_i
11: $\{S_i\}, 0 \leq i < R$
12: **end procedure**

Each binary search tree is called a cluster and every node in a cluster is connected to DA tries (see figure 3.3 [5]). Each incoming packet is fed into all clusters in parallel. Once there is a match in a DA

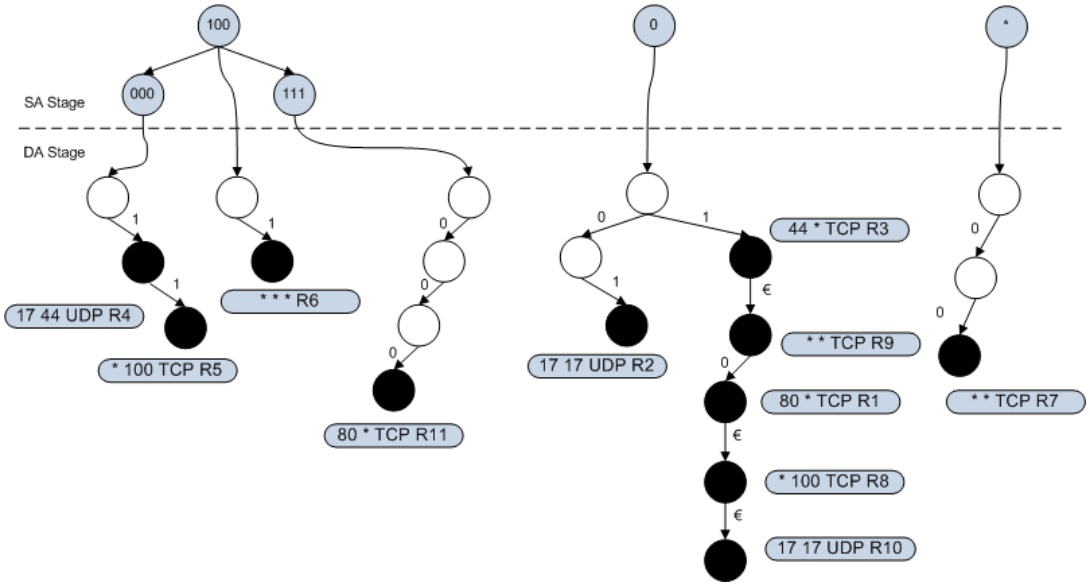


Figure 3.3: Hierarchical Hybrid Search Structure.

node (marked black in the figure 3.3) the remaining fields (SP, DP and protocol) in that node are also compared with the incoming packet. When more than one rule has the same DA and SA values, the DA node that holds these rules is called a super node. The variety in the size of supernodes leads to memory inefficiency in hardware implementations. To solve this problem, these supernodes are split into sequentially connected smaller and fixed size nodes in the proposed scheme [5]. These rules are linked listed in the memory structure and in every stage (every clock cycle) one rule is inspected. The transition between these nodes are called epsilon transitions and the whole structure is named TTϵ as an acronym of tree, trie, epsilon sequence. This algorithm offers 134 Gbps throughput while achieving 23byte/rule memory efficiency for a rule set of ACL 10K from [25].

3.2 Latency vs. Memory

Although TTϵ algorithm solves the backtracking problem in H-tries and also offers memory efficiency in high speed packet classification, the number of pipeline stages causes a remarkably long latency for packet forwarding. Due to the epsilon transitions used to represent supernodes, the number of pipeline stages increases up to 55. Moreover, the depth of the trie might increase more with the rule updates. The latency of processing a packet is critical since packet forwarding has strict limitations to meet QoS requirements. In addition to latency problem, the power management of the hardware is severely affected due to large number of pipeline stages.

In order to decrease the number of pipeline stages, HHSS [5] proposes a new memory structure to store more than one rule in one DA trie node. The number of rules in a supernode is called Ptrie. The tradeoff between number of pipeline stages and memory requirement is demonstrated in figures 3.4 and 3.5 [5]. The number of pipeline stages (and hence latency) decreases by increasing Ptrie value, which also increases the memory requirement.

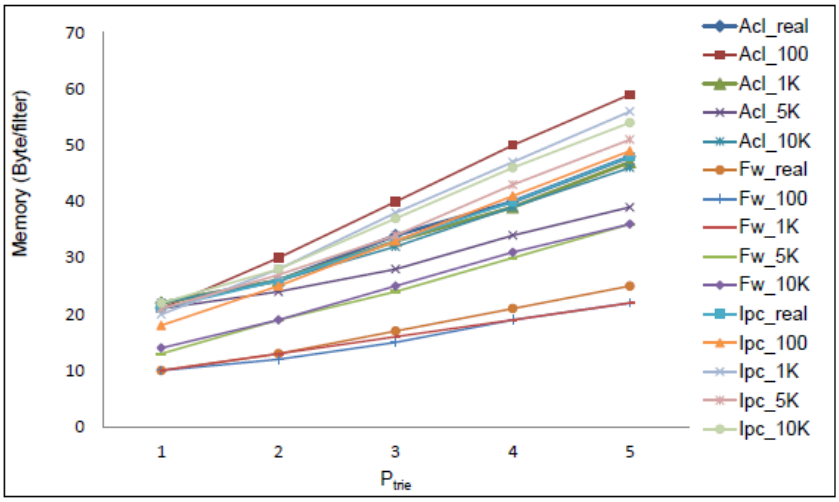


Figure 3.4: Memory requirement for various Ptrie values.

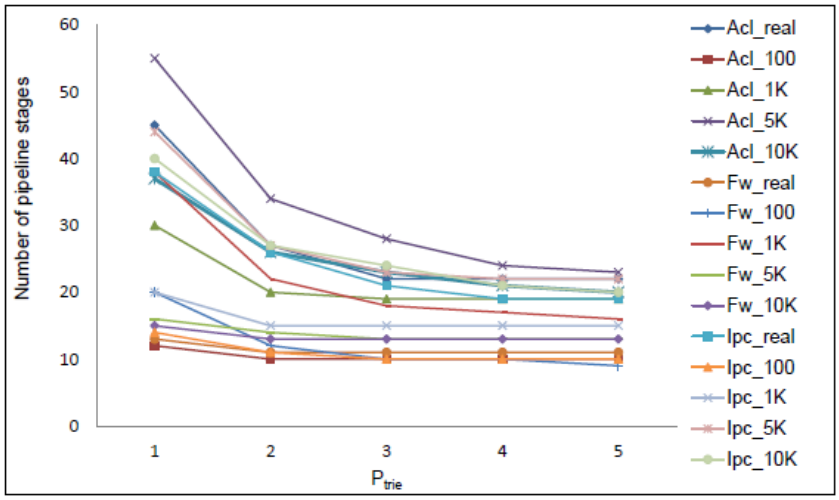


Figure 3.5: The number of pipeline stages for various Ptrie values.

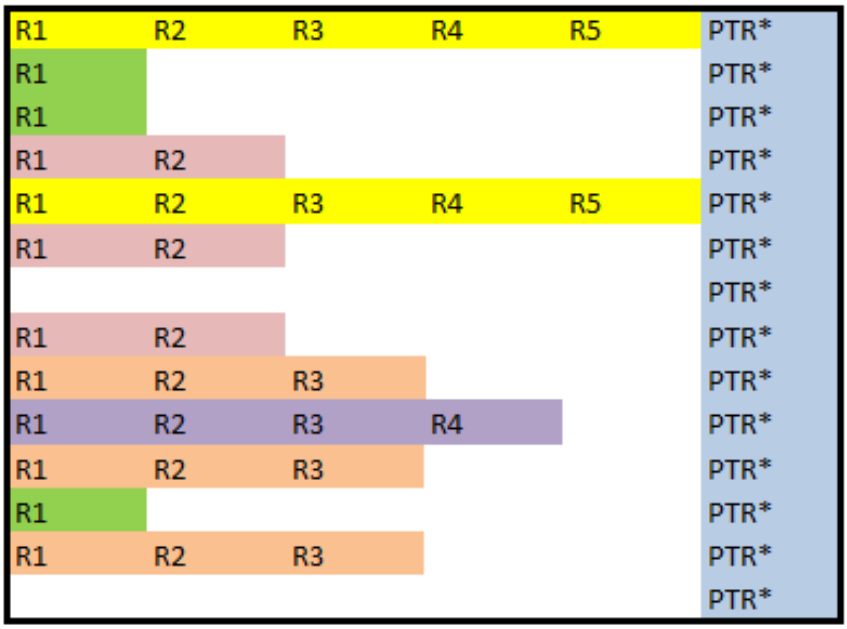


Figure 3.6: Memory structure for Ptrie=5.

The main reason behind the increase in memory requirement is the large gaps present in memory blocks. For example, a node that is actually capable of storing 5 rules (in Ptrie=5 case) carries only the pointer data for the next node. The simulations indicate that the memory requirement is doubled when Ptrie value is equal to 5. The motivation of this thesis is to organize the memory structure to achieve both a memory efficient and a low latency solution for packet classification while still providing high throughput.

3.3 Bin Packing Problem

When Ptrie value is set to 5, BRAM width is chosen as to store data for five rules, pointers and other headers such as path compression parameters. This amount of memory is more than sufficient to store nodes that contain less than five rules. The gaps, which occur when less than 5 rule nodes are stored in the block RAMs, make the memory usage inefficient as it is clearly seen in figure 3.6. Better memory management would increase the memory efficiency while achieving low latency values.

The problem of memory organization (fragmentation) might be considered as a bin packing problem.The solution of this problem has two aspects. Firstly, the smallest container (memory) for different sized nodes has to be chosen. Secondly, the packing process of the nodes has to be easy to implement. The approaches to fragment the memory blocks are proposed in the content of this thesis as follows.

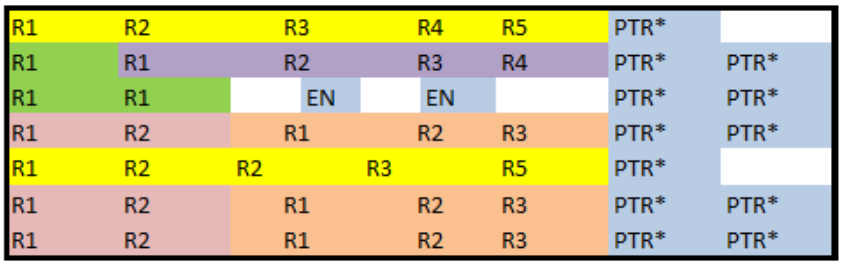


Figure 3.7: Memory structure for one large BRAM.

3.3.1 One Large Bram For Each Stage

This approach offers to use one large memory block in each stage of the pipeline. Trie nodes in the same level might be stored in the same row of the memory in order to minimize the gaps. The word size of the memory can be optimized to provide the best memory utilization. Although this bin packing problem is an NP-complete problem, one simple heuristic might be to choose the memory word size such that it can store five rules plus two pointer (PTR*) sized data.

When the rules in figure 3.6 are organized based on this scheme, total memory usage is drastically decreased. The empty node (EN) headers might be placed into smaller gaps in memory as it is shown in figure 3.7 .

Although this approach offers a better memory usage, storing multiple node data in the same entry requires additional addressing and indexing resources, which directly affects hardware complexity. This increased complexity is expected to have a negative influence on throughput.

3.3.2 Using Auxiliary Data Structure

The width of the BRAM can be determined based on the average number of node sizes. The rules that can not be located in a BRAM can be moved out to an auxiliary data structure such as TCAM, BV or some other implementation that maybe executed in parallel.

Figure 3.8 illustrates the shape of memory for a word length of 3 rules and pointers. This scheme achieves a better memory utilization compared to the five rule sized nodes of HHSS, however, still considerable amount of gaps (fragmentation) in the memory blocks will occur.

3.3.3 Multiple One Rule Sized Brams For Each Stage

Employing multiple BRAMs, each one being capable of storing a single rule per stage, is very similar to what is proposed in one large BRAM case above (see figure 3.9) The distinguishing part here is that the required data can directly be output by enabling the related memory blocks without the need for an additional process to obtain the appropriate part of the selected entry. In this scheme, there will be

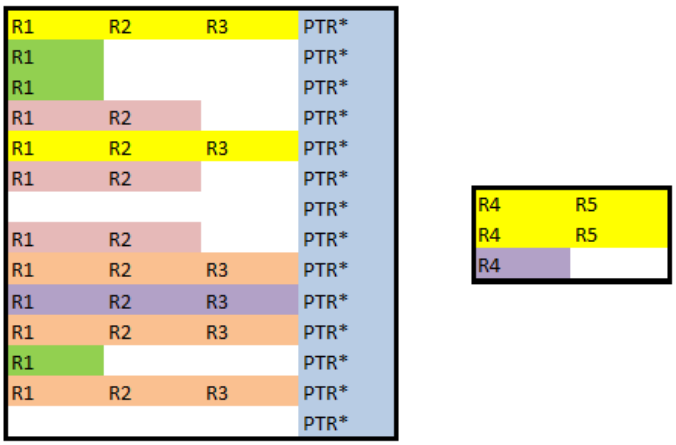


Figure 3.8: Memory structure with auxiliary data structure.

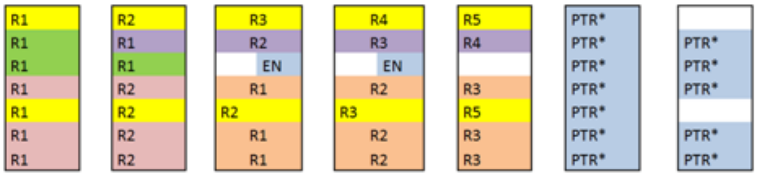


Figure 3.9: Memory structure for one rule sized BRAMs.

additional memory mapping and small gaps in the memory similar to one large BRAM approach. The other remark for this approach is the power budget. In this case, since unused memory blocks can be disabled, substantial power saving may be possible.

3.3.4 Multiple BRAMs with Variable Word Lengths For Each Stage

To use memory even more efficiently we propose to deploy multiple and variable sized memory blocks at each pipeline stage. In this approach individual memory blocks will be reserved to place empty nodes, one rule nodes, two rule nodes and so on as shown in figure 3.10.

In each stage, only a single memory block is enabled at a time unlike the multiple but one rule sized BRAMS approach. Also, the output of the memory block directly provides the required data. Therefore, memory mapping is much simpler when compared to others. Last but not the least, since there is no memory gap in this approach, the memory utilization is expected to be much higher than the previous cases.

In this thesis work, due to its simplicity and high memory efficiency, this approach is implemented and called optimized HHSS. Optimized HHSS is tested and compared to other state of art algorithms and

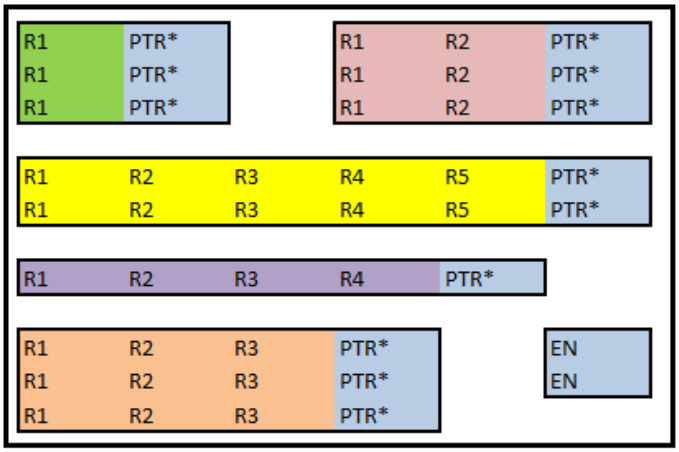


Figure 3.10: Memory structure for multiple BRAMs with variable word lengths.

described in detail in the following chapters.

CHAPTER 4

IMPLEMENTATION

This chapter presents the implementation details of the proposed approach in the previous chapter.

4.1 Sample Rule Sets

The performance of the packet classification algorithms depends on the filter set used. For example, the use of non overlapped SA filters solves the backtracking problem implicitly and the proper selection of DA filters drastically changes the tree depth. Therefore, using a common base for benchmarking is necessary for comparable results and for a proper evaluation.

In this work, Classbench [25] is used in order to evaluate the proposed algorithm. The real life rule sets such as Access control lists (ACL), Firewall (FW), IP Chain (IPC) are used to seed filter set generator and the rule sets that has the same characteristics of real filter sets are generated [1]. Each filter set type has its own characteristics, therefore it has different distribution on the proposed data structure used to store the rule sets. This behavior offers various analysis possibilities on the memory structure. Moreover, the size of the rule sets alters between hundred to ten thousand rules hence giving the opportunity to measure the scalability of the proposed algorithm.

4.2 Memory Requirement

To observe the trie node size distributions, we simulated the proposed algorithm in section 3.3.4 with the sample rule sets using a C++ platform. Our algorithm inputs the sample rule sets and constructs a trie structure using the SA field prefixes of the filters. Next, the trie is divided into clusters as it was described in section 3.1. Following the clustering process, the SA nodes are connected with DA tries. Finally, the algorithm outputs the number of nodes in each stage (both SA tree and DA trie) and the number of rules in DA trie nodes. These numbers are used in both memory calculations and hardware design.

The distribution of the node sizes for each rule set is listed in tables 4.1, 4.2 and 4.3 where the first columns show the type of nodes. These first results indicate that the empty nodes in the tree structure keeps a significant amount of space, for instance 84% of the total memory is used to store empty nodes

Table 4.1: Memory distribution without path compression for ACL rule sets.

	ACL 1k	**ACL 5k**	**ACL 10k**	**ACL 100**
empty node	8466	44000	182944	2072
1 Rule Node	352	1444	5378	75
2 Rule Node	52	303	870	7
3 Rule Node	31	128	293	0
4 Rule Node	19	57	129	1
5 Rule Node	10	37	67	1
6 Rule Node	9	32	29	0
7 Rule Node	5	21	15	0
8 Rule Node	4	16	13	0
9 Rule Node	2	16	8	0
10 Rule Node	3	13	8	0

Table 4.2: Memory distribution without path compression for FW rule sets.

	FW 1k	**FW 5k**	**FW 10k**	**FW 100**
empty node	3326	46567	152547	315
1 Rule Node	203	3563	9234	20
2 Rule Node	48	278	2	6
3 Rule Node	17	77	0	2
4 Rule Node	12	28	0	2
5 Rule Node	5	13	0	1
6 Rule Node	6	10	0	3
7 Rule Node	4	0	0	2
8 Rule Node	5	0	0	0
9 Rule Node	2	0	0	1
10 Rule Node	3	0	0	0

for ACL 10k rule set. In order to minimize the number of empty nodes, the path compression algorithm can be utilized [18].

The path compression algorithm allows to remove empty nodes if the node has only one child. With this method significant amount of the empty nodes are removed from the trie. Skip value (SV), the address value skipped till the next node, and the bit string (BS), the bit length of the skip value, are recorded in the nodes to adjust proper tracking. Although this algorithm brings additional overhead to trie nodes, the number of empty nodes in the resulting trie structure is substantially reduced. For this reason, the path compression algorithm is quite beneficial to use for sparse trie structures that suffer from large number of empty nodes.

In our next iteration, search tree and tries are re-constructed using the path compression algorithm and the results are recorded in tables 4.4, 4.5 and 4.6. When compared to the previous case, it is observed that the number of empty nodes are drastically decreased especially for the rule sets with large number of entries.

Table 4.3: Memory distribution without path compression for IPC rule sets.

	IPC 1k	**IPC 5k**	**IPC 10k**	**IPC 100**
empty node	15558	45148	104022	2036
1 Rule Node	777	2651	5856	88
2 Rule Node	49	363	711	3
3 Rule Node	8	126	212	0
4 Rule Node	7	57	87	0
5 Rule Node	1	26	34	1
6 Rule Node	1	21	22	0
7 Rule Node	0	11	14	0
8 Rule Node	0	5	6	0
9 Rule Node	0	2	5	0
10 Rule Node	0	3	4	0

Table 4.4: Memory distribution with path compression for ACL rule sets.

	ACL 1k	**ACL 5k**	**ACL 10k**	**ACL 100**
empty node	280	2062	6731	80
1 Rule Node	310	1444	5378	75
2 Rule Node	45	303	870	7
3 Rule Node	30	128	293	0
4 Rule Node	17	57	129	1
5 Rule Node	9	37	67	1
6 Rule Node	7	32	29	0
7 Rule Node	5	21	15	0
8 Rule Node	4	16	13	0
9 Rule Node	2	16	8	0
10 Rule Node	2	13	8	0

Table 4.5: Memory distribution with path compression for FW rule sets.

	FW 1k	**FW 5k**	**FW 10k**	**FW 100**
empty node	202	2851	6948	24
1 Rule Node	203	3563	9234	20
2 Rule Node	48	278	2	6
3 Rule Node	17	77	0	2
4 Rule Node	12	28	0	2
5 Rule Node	5	13	0	1
6 Rule Node	6	10	0	3
7 Rule Node	4	0	0	2
8 Rule Node	5	0	0	0
9 Rule Node	2	0	0	1
10 Rule Node	3	0	0	24

Table 4.6: Memory distribution with path compression for IPC rule sets.

	IPC 1k	**IPC 5k**	**IPC 10k**	**IPC 100**
empty node	615	2106	4672	79
1 Rule Node	777	2651	5856	88
2 Rule Node	49	363	711	3
3 Rule Node	8	126	212	0
4 Rule Node	7	57	87	0
5 Rule Node	1	26	34	1
6 Rule Node	1	21	22	0
7 Rule Node	0	11	14	0
8 Rule Node	0	5	6	0
9 Rule Node	0	2	5	0
10 Rule Node	0	3	4	0

Furthermore figure 4.1 illustrates the proportion of the rules stored in the trie with respect to the rule size of nodes. This graph gives the proportion of the rules covered in the trie (y-axis) versus the maximum number of rule size for the DA nodes (x-axis). For instance, when the maximum number of rule size is equal to 3, approximately 0.42 of the rules in FW 100 rule set is stored in the trie. The limit number of rules in a node has to be chosen to cover as many rules as possible without over usage of hardware resources.

As it is seen in the graph, one to five rule carrying nodes covers 90% of the rules in half of the rule sets. Moreover at least 50% of the filter set is covered by one to five rules for the other half. The proposed algorithm has the flexibility to use one to any sized nodes in DA trie, however, these results show that choosing 5 as a limit is meaningful since beyond this has little gain for the latency while consuming hardware resources unnecessarily and causing routing complexity.

On the other hand, these results also indicate that a method for handling the remaining nodes that are not covered by the one to five rule sized DA nodes is also needed. For the rule sets with the 90% coverage the remaining nodes might be stored in TCAM, while epsilon transition [5] method is applied for the remaining ones.

4.3 Memory Calculation

Based on the simulation results in Section 4.2, the limit node size is chosen as 5 as a result of the distribution of rules over the tree. This section describes how to calculate the required total memory to store a single filter set using the proposed algorithm.

Hierarchical tree has two main parts which are SA binary search tree and DA trie. The nodes in SA tree are called SA nodes. When a packet comes to a SA node it is compared with the SA value that is stored in the node. If the packet has a smaller SA value, the packet is directed to the node that is pointed by the left pointer while the right pointer is followed if the packet has a larger SA value. Lastly, if the packet has the same value with the SA node data then it jumps to the DA trie following the DA

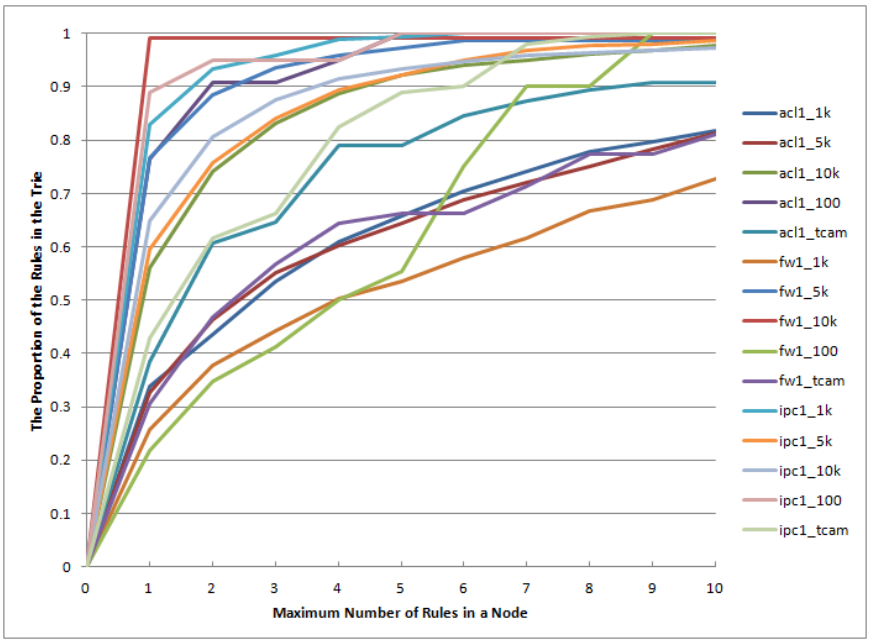


Figure 4.1: Rule set coverage due to node types.

pointer. Briefly, each SA node includes SA value (32 bit), DA pointer, right pointer and left pointer. Although SA value requires exactly 32 bits, the memory required to store address pointers are variable since the amount of memory that will be addressed changes. Because of this reason, the address bits are decided according to the largest memory required for a stage (level) of the tree.

In a DA trie, the nodes are called DA nodes and these nodes are allowed to store one to five rules. If a node does not store a rule then it is called empty node which comprises of only the left pointer, right pointer, bit string, skip value and the memory select bits. If a DA node carries the information part of a rule, then it additionally stores SP high (16 bits), SP low (16 bits), DP high (16 bits), DP low (16 bits) and Protocol (8 bits) values which results in a total of 72 bits per rule. Note that the priority bits and the action bits are omitted as [5], [12], [13] reports.

Similar to the SA tree, the left and right pointers are determined according to memory size. The bit string parameter requires 5 bits while 32 bits are reserved for the skip value for path compression (PC) process. The memory select bits are 3 bits since there are six different memory blocks. Each DA node includes these parameters and the memory calculation is done for different rule sets by considering all these values.

For the TCAM part, each rule is represented as SA, DA, SP high and low, DP high and low and the protocol bits. Hence a 17 byte memory consumption is required for each rule in the TCAM.

Table 4.7 illustrates the distribution of DA nodes on different trie levels for ACL 10K rule set. Before calculating the total memory, one has to decide the required pointer size. In this example, the largest block involves 4759 nodes. To address this memory, the pointer size has to be 13 bits as $\lfloor log_2(4759) \rfloor = 13$. Moreover, since the PC is done for all 32 bits of DA, bit string value is 5 since the skip value is represented using 32 bits. By adding 3 bits for memory selection to these values, the memory mapping

Table 4.7: Distribution of different types of DA nodes in each level of the trie.

	1st Level	**2nd level**	**3rd Level**	**4th Level**	**5th Level**	**6th Level**	**7th Level**	**8th Level**
Empty Nodes	4759	1402	445	101	20	3	1	0
1 Rule Nodes	24	3459	1294	449	123	23	5	1
2 Rule Nodes	1	535	226	84	18	6	0	0
3 Rule Nodes	0	178	82	21	9	2	0	1
4 Rule Nodes	0	58	48	19	4	0	0	0
5 Rule Nodes	0	36	16	9	5	1	0	0

Table 4.8: Memory calculation of DA Trie for ACL 10k rule set.

	Node Count	**Memory of a Node (bits)**	**Total Memory (bits)**
Empty Nodes	6731	66	444246
1 Rule Nodes	5378	138	742164
2 Rule Nodes	870	210	182700
3 Rule Nodes	293	282	82626
4 Rule Nodes	129	354	45666
5 Rule Nodes	67	426	28542
Total			1525944

requires 66 bits. Furthermore, 72 bits are required to store a rule and these values are multiplied by the rule size in a node.

Table 4.8 shows node count and the amount of memory required for each node type in the DA trie. As seen in table 4.8, 190743 bytes (1525944 bits) of memory is required to store the DA trie when ACL 10k rule set is used.

In addition to DA trie, the SA tree statistics are also required to calculate the total amount of memory needed. The pointer length has to be determined for this calculation. Since a binary search tree with depth n has at most 2^{n-1} nodes at level n, $n-1$ bits are adequate to address each memory level.

The SA node distribution on clusters are illustrated in table 4.9. For example, cluster 1 has 4276 nodes and the depth of the binary search tree is 13. Therefore, 12 bits are required to address the next node. The calculation is done similarly for the other clusters and noted in table 4.9. Finally, SA node sizes are determined and multiplied with the node count of each cluster. The amount of memory for the SA tree is calculated as 40689 bytes (325514 bits) in total.

In addition to DA trie and SA tree, the remaining nodes that has more than five rules in it (less than

Table 4.9: Memory calculation of SA Tree for ACL 10k rule set.

	CLUSTER1	**CLUSTER2**	**CLUSTER3**	**CLUSTER4**	**TOTAL**
SA Node Count	4276	381	126	1	
DA Pointer	13	13	13	13	
SA Pointer	12	8	6	1	
SA Value	32	32	32	32	
SA Node Size	71	63	59	47	
Memory (bits)	303596	24003	7434	47	325514

10% of total nodes) is stored in TCAM. 755 nodes are not stored in the tree/trie structure in ACL 10k rule set. The required memory is calculated by multiplying the number of remaining nodes with 17 bytes. As a result, 12835 bytes of memory is required for the rules stored in TCAM.

In total, 244267 bytes are required for storing the rule set ACL 10k. The memory requirement varies according to the number of entries in a rule set. In order to achieve a better comparison, memory requirement per rule is used as alternative metric. In this case, since ACL 10k has 9603 rules 25.44 bytes/ rule memory is the final result. For the other rule sets the same method is applied and the calculation results are as follows:

- ACL: 18.63 bytes/rule
- ACL 100: 21.83 bytes/rule
- ACL 1k: 18.39 bytes/rule
- ACL 5k: 19.87 bytes/rule
- FW: 16.90 bytes/rule
- FW 100: 15.92 bytes/rule
- FW 1k: 16.99 bytes/rule
- FW 5k: 23.47 bytes/rule
- FW 10k: 26.28 bytes/rule
- IPC: 18.05 bytes/rule
- IPC 100: 23.98 bytes/rule
- IPC 1k: 22.69 bytes/rule
- IPC 5k: 19.47 bytes/rule
- IPC 10k: 20.72 bytes/rule

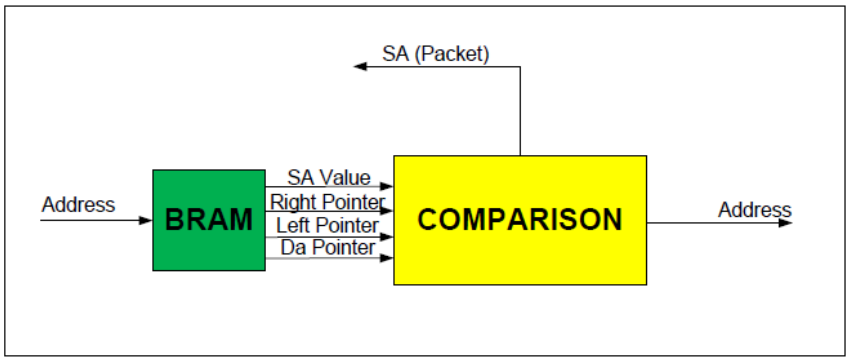


Figure 4.2: Hardware Structure of a SA node.

4.4 Hardware Simulations

The proposed memory scheme is implemented in hardware to find its throughput. The throughput is directly calculated from the clock period, that is the time passed in a single stage in processing a packet.

The proposed algorithm is simulated on Field Programmable Gate Arrays (FPGAs) since it is the most convenient platform to implement pipelined architectures due to its reconfigurable architecture and abundant parallelism. The computations conducted in C++ platform with the sample rule set gives us the mapping of the rules on the tree/trie structures. The FPGA simulations are completed using these results.

ACL 10k rule set is used in our simulations since it is the largest rule set and expected to give the worst case results. The hardware is implemented on Xilinx Virtex5 XC5VFX200T (-2 speed grade) using Xilinx ISE 13.4 development tool. This specific device is chosen to obtain comparable results with the ones reported in the literature [5], [11], [29].

The hardware has two main blocks which are SA blocks and DA blocks. In SA blocks there are two components as follows: a comparison module to decide the next memory address and a BRAM module to store nodes as illustrated in figure 4.2. In the comparison module, SA value output from the BRAM is compared with the incoming packet. If the SA of the packet has a larger value compared to SA value, the right pointer is output as the address. If the SA of the packet has a smaller value the left pointer is output. Finally if the SA value is equal to the SA of the packet the DA pointer is activated. The output of each stage provides an input for the next one and this sequence goes through the pipeline stages.

DA modules are more complicated compared to SA modules because, the following processes are executed in a DA stage (i) comparison of remaining fields (SP, DP, Protocol) with the rules, (ii) next memory address selection using path compression parameters. The direction of a traversal in a trie is decided according to DA bits of the incoming packet. If the most significant bit is equal to zero the left otherwise the right direction is chosen. In every DA module, DA is shifted left by one bit. Therefore, in every stage the 31^{st} bit of the DA is used to select the direction for further moves on the

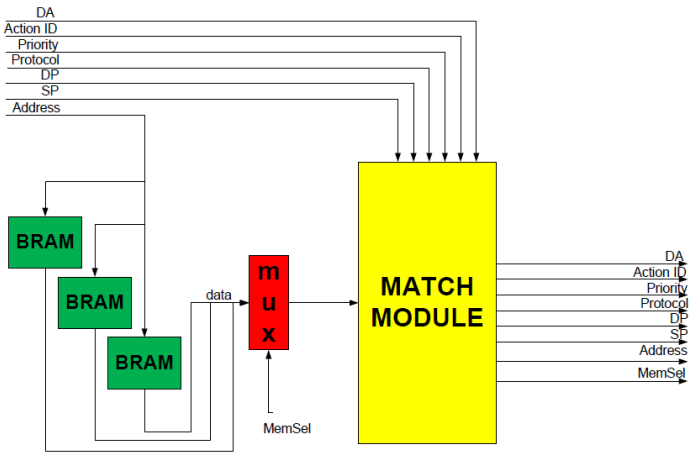


Figure 4.3: Hardware Structure of a DA node.

trie. However if path compression is applied, left shift is done according to the value of the bit string field. The multiplexer in the DA module determines how many shifts will be carried out and this is controlled by the bit string and skip value.

In addition to the next node decision task, comparison between the rules and the packet header is done also in the DA module. For each rule, the incoming packet is checked whether it has the SP and DP in range and the exact same protocol with the rule and match condition is output. The top level block diagram for a DA node is illustrated in figure 4.3.

Both SA and DA modules are designed as combinational logic. Only memory modules are enabled with a clock. The Post Placement and Routing Analysis made in Xilinx ISE gives the clock speed by adding the routing delay to the maximum combinational delay between two clock cycles. In other words, the resulting clock period is equal to the pipeline stage period that is required for the throughput calculation.

Since speed is one of the major design concerns ”Design Goals and Strategy” of Xilinx ISE is set to timing performance during the simulations. The experiments indicated that the multiplexer that is used for PC algorithm causes a large delay when the bit string is equal to 5. In this case, 32 different PC cases that skips one to 32 bits of DA occur. If bit string is reduced to 4, the path compression is limited to 16 cases which means that the combinational delay of the multiplexer will be less compared to 32 bit PC. In order to measure the effect of the PC limit to the clock speed, different levels of path compression is applied in hardware. Figure 4.4 summarizes the behavior of PC over the hardware. Considering this behavior, limited PC on the trie is required to achieve better throughput.

For this reason, path compression is limited to eight bits, where at most 8 nodes are skipped between two nodes, and the bit string value is reduced to 3 bits. The memory simulation for 8 bit PC is generated and the memory calculations for DA trie is done for 8 bit and 32 bit PC. As it is seen in table 4.10, the memory storage requirement of 8 and 32 bit PC are close each other. Since the 8 bit PC gives quite

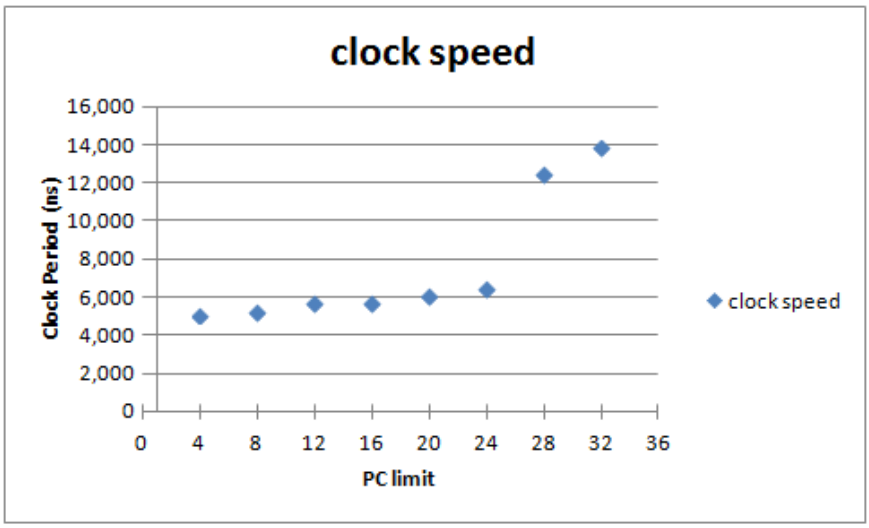


Figure 4.4: PC and clock speed trade off.

better results for the hardware simulations and comparable results with higher degree PC values, the PC value is limited to 8 bits.

After the PC value is set, the hardware simulations are repeated. Since the routing and placement of an FPGA is also a difficult problem, longer simulations gives better results. By changing the design goal and strategies, better timing results might be achieved. In order to get the best result, map placer level extra effort and place and route level extra effort are spent and the final result is found as 4,977ns. As it was stated previously, this hardware design neglects the priority bits and the encoder where the best match is chosen between all matching rules since the sample rule sets [1] has no information on the priority and the literature has also discarded this step. Therefore, all match signals are output in the top level in the hardware architecture. Additionally, in order to simulate the priority encoder, another simulation is done where match signals are connected via OR gates. In this simulation the clock speed is noted as 5.045 ns.

Table 4.10: DA trie memory requirements for PC - 8 and PC - 32.

rule set	**PC - 8 bits**	**PC - 32 bits**
ACL 1k	13.567	13.537
ACL 5k	13.942	12.651
ACL 10k	24.098	19.837
ACL 100	20.906	19.878
FW 1k	8.2421	8.8148
FW 5k	19.339	20.099
FW 10k	26.97	22.835
FW 100	7.3804	8.5761
IPC 1k	20.921	20.204
IPC 5k	17.43	17.261
IPC 10k	19.02	18.321
IPC 100	21.338	20.227

CHAPTER 5

PERFORMANCE EVALUATION

In this chapter, the final results are presented and compared with other classification algorithms with respect to memory requirement, throughput and latency.

5.1 Experimental Setup

The simulations are completed with several sample filter sets that has the same characteristics of real filter sets. ACL, FW, IPC are used as sample sets [1]. 100, 1k, 5k and 10k rule sets are generated for each filter type.

In addition to that, the maximum rule size in a DA node is limited to 5. Also, the PC is limited to 8 bits in both memory calculations and hardware simulations.

5.2 Memory

The memory requirement for each set is calculated and compared with the results of several other algorithms in table 5.1.

BV [16] algorithm as a representative of decomposition based packet classification algorithms requires a large amount of memory. As it was discussed in section 2.1.1.1, the storage requirement of the BV algorithm increases geometrically. Therefore, for the rule sets with larger number of entries, a significant amount of memory is required compared to the other algorithms.

Although the Hypercuts algorithm proposes a sophisticated solution to distinguish the rules from each other, it also suffers from high memory consumption specifically for FW rule sets because of the overlapping regions of the header fields. In order to explain this figure 5.1 [11] illustrates the geometrical representation of six different and highly overlapped rules and the distribution of them on Hypercuts decision tree. Since R1 and R2 are highly overlapping with other rules in the coordinate system, it is hard to distinguish them from the other rules. The consecutive cut operations to distinguish these rules causes deep decision trees. More importantly, R4 and R5 need to be stored in many nodes during these cut operations and therefore the redundant storage of the rules increases the memory consumption

Table 5.1: Memory efficiency (bytes per rule).

Rule Sets	**# of rules**	**BV [16]**	**Hyper Cuts [22]**	**Hybrid scheme [11]**	**EGT [2]**	**TTϵ [5]**	**TTϵ Ptrie=5 [5]**	**Opt. HHSS**
ACL real	752	71.80	32.58	N/A	25.41	22.08	48.76	17.403
ACL 100	98	47.35	27.78	24.44	27.69	21.42	59.95	22.86
ACL 1k	916	91.63	38.15	22.98	24.96	22.2	47.15	17.64
ACL 5k	4415	257.23	59.64	24.83	24.87	21.44	39.78	20.20
ACL 10k	9603	789.22	54.22	25.51	30.23	22.62	46.54	29.47
FW real	269	40.72	399.18	N/A	25.31	10.48	25.48	14.27
FW 100	92	27.46	113.37	56.63	23.42	10.37	22.74	13.28
FW 1k	791	67.08	6110.58	215.06	23.80	10.41	22.91	15.04
FW 5k	4653	691.69	16132.65	255.13	39.04	13.27	36.17	22.63
FW 10k	9311	1582.18	12554.18	248.54	49.45	14.51	36.93	30.44
IPC real	1550	61.57	128.52	N/A	26.63	21.64	48.03	20.59
IPC 100	99	69.16	24.57	23.65	31.60	18.34	49.30	25.15
IPC 1k	938	176.03	61.34	25.63	29.95	20.05	56.61	23.38
IPC 5k	44.60	358.61	406.80	49.46	27.62	21.56	51.53	19.42
IPC 10k	9037	788.69	2378.35	43.30	28.92	22.81	54.92	21.24

considerably.

The memory storage calculations done in section 4.2, indicates that the number of the entries in FW 10k, FW 100, FW real, ACL 1k and ACL 5k rule sets that share exactly the same SA and DA prefixes with at least 5 other rules are between 35% the 45% of the total rule set (see figure 4.1). The overlapping regions of such rules can hardly be distinguished by Hypercuts algorithm. As seen in table 5.1, these rule sets, especially the FW 10k, FW 100 and FW real ones require large amount of memory. However, in addition to SA and DA fields, SP, DP and PRTCL fields are used in Hypercuts to construct a tree. Therefore, in order to measure the total overlap degree, the effect of these fields should also be included.

The overlap degree of a rule (r) is defined as the ratio of number of overlapping rules with the rule ($No(r)$) to the total number of entries of a rule set ($|R|$). The total overlap degree of a rule set (R) is the mean of overlap degree of all rules [11].

$$Overlapdegree(r) = \frac{No(r)}{|R|} \quad (1)$$

$$Overlapdegree(R) = \frac{\Sigma_{r \in R} Overlapdegree(r)}{|R|} \quad (2)$$

The overlap degree of the six different rule sets employed are calculated according to equations (1) and (2) and are listed in table 5.2. The overlap degree of the FW rule sets are larger than ACL and IPC rule sets. Therefore the memory requirement of Hypercuts using FW rule sets are larger when compared

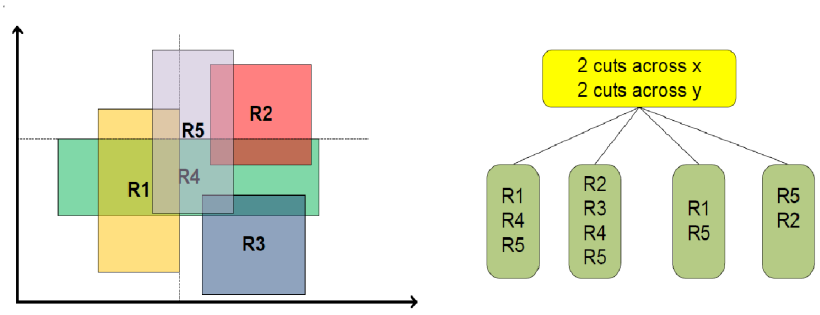


Figure 5.1: Example Hypercuts geometrical view and decision tree.

Table 5.2: The overlap degree of different rule sets [11].

rule set	**# of rules**	**Overlap Degree**
ACL	752	0.0101
IPC	1550	0.0161
FW	269	0.1327
ACL 10k	9603	0.0007
IPC 10k	9037	0.0051
FW 10k	9311	0.0964

to other rule sets.

A hybrid solution comprising decision and decomposition based algorithms has been proposed in literature to reduce the overlap degree in the Hypercuts. In this solution, the rules with the high overlap degree are selected and moved into a different set called the common set. The remaining rules are stored in Hypercuts decision tree. After the common rule set is separated the rules with the smaller overlap degree can easily be distinguished, hence rule duplication in the tree structure is reduced as shown in figure 5.2 [11]. The hybrid scheme uses BV algorithm to store the rules in common set. Since the BV algorithm is not sensitive to the overlapped regions and the number of entries in the common rule set are small enough, the memory consumption for the highly overlapped rules is reduced in the hybrid scheme.

Table 5.1 presents that the memory consumption of the hybrid scheme is significantly smaller compared to both BV and Hypercuts algorithms. However, the memory requirement is still remarkably large for FW rule sets, the reason still being the highly overlapped rule field structure of FW. When the number of rules in the common set is large, the memory requirement increases drastically due to $O(N^2)$ storage requirement of the BV algorithm.

In table 5.1, TTϵ [5] algorithm, EGT [2] and optimized HHSS has comparable results with respect to memory storage. Since all these algorithms use hierarchical tree structures, it is expected that they have similar storage requirements. Although memory efficiency is similar, both EGT and TTϵ suffer from long latency values. In EGT algorithm, the backtracking process during the tree traversal causes an

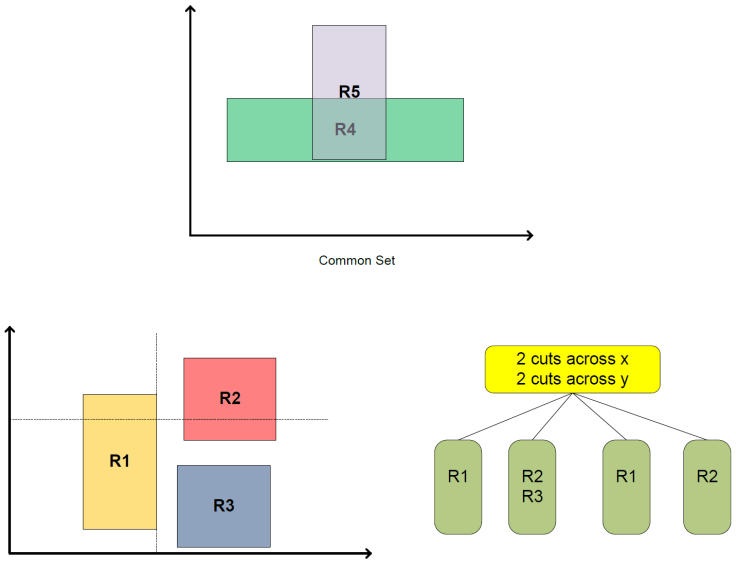


Figure 5.2: Decision tree without highly overlapped rules.

increased search time. In TTϵ algorithm, numerous epsilon transitions leads to deep pipeline problem. Although TTϵ with Ptrie-5 reduces the depth of the tree and offers shorter pipeline structure, then the memory requirement increases remarkably due to large gaps in the memory blocks. On the other hand, our proposed memory organization strategy reduces the number of pipeline stages without disrupting the memory storage requirement by using multiple sized memory blocks.

5.3 Throughput

Xilinx Virtex-5 XC5VFX200T (-2 speed grade) is used as the target device and the simulations are done using Xilinx ISE 13.4 platform. The proposed algorithm is mapped on the hardware for ACL 10k rule set which has the largest number of entries among the used sample rule sets. The clock speed of the algorithm is found to be 4,977 ns for the acl 10k rule set when PC limited eight bits. Using dual port BRAMs and by utilizing a pipelined structure, the hardware supports approximately 4 million packets per second which leads to 128 Gbps throughput for 40 byte (minimum size) packets.

5.4 Latency

The latency for our approach is calculated to be approximately 104 ns for ACL 10k rule set. The latency value of TTϵ is approximately 263 ns in the worst case because of deep pipelining problem of epsilon transition. Our proposed algorithm achieves a smaller latency value while offering similar results for the other performance metrics, which are already superior to other state of the art algorithms.

5.5 Scalability

The variation of the memory requirement for a rule for different sized rule sets generally used to indicate the scalability of a packet classification algorithm. As can be seen in table 5.1, the memory requirement per rule metric does not change drastically by the number of entries in the rule sets. In other words optimized HHSS may also be regarded as scalable to large rule sets.

5.6 Rule Update

The updates in the rule sets include addition of a new rule and modification or deletion of an existing rule. In our proposed memory strategy, the updates are handled similar to other linear pipeline classification algorithms [5], [11], [14]. The new memory content is calculated offline and write bubbles are used to delete or modify the content of the memory in the pipeline stages [3]. Each epsilon transition occupies a level in the tree structure. If a new rule is connected to the tree via an epsilon transition, the rules below this stage are required to be shifted in the pipeline. Hence, a new rule update with an epsilon transition may cause deep changes in the tree structure and a number of write bubbles are needed to maintain a rule insertion. Therefore, the new rules are buffered in TCAM and the FPGA is reconstructed when the buffer is full.

In our proposed approach on the other hand, the node size is limited by five and six different sized BRAMs are allocated to store zero to five rules. However, this number might be increased up to the point where there is no need for the epsilon transitions. Although the hardware resources are overly exploited with parallel comparison blocks in this architecture, the new rule updates are maintained in one clock cycle.

5.7 Performance Comparison

Optimized HHSS is compared with other state of art packet classification algorithms in the Table 5.3. The memory requirement per rule and throughput value are indicated in this table. Morevover, the memory efficiency versus throughput performances of these algorithms are illustrated in figure 5.3.

All throughput values of the algorithms that uses FPGA platform are scaled to Xilinx Virtex-5 devices in order to make a fair comparison. The original results that are stated in related papers are also presented in parentheses. Note that the TTϵ with 5 rule nodes and BV-TCAM [23] are not simulated on hardware. For the TTϵ with 5 rule nodes the throughput value is assumed that it is equal to TTϵ with 1 rule nodes (*). However, it is likely that the TTϵ with 5 rule nodes has a lower throughput value because of the additional hardware and routing complexity. For the BV-TCAM the expected throughput value given in [23] is recorded in table 5.3.

These results clearly indicate that our design offers improved throughput and memory results compared to other architectures except TTϵ. However, TTϵ suffers from long latency due to the number of pipeline stages and decreasing it by TTϵ-Ptrie=5 algorithm causes an extra use of memory resources.

Table 5.3: Performance comparisons.

Packet Classification Methods	**Platforms**	**# of rules**	**Total Memory used (bytes/rule)**	**Throughput (Gbps)**
B2PC[20]	ASIC	3300	163,63	13,60
Memory-based DCFL[10]	FPGA	128	1726,56	24,00(16)
2sBFCE[19]	FPGA	4000	44,50	2,06(1,88)
BV-TCAM[23]	FPGA	222	72,07	10,00(N/A)
Simplified HyperCuts[15]	FPGA	10000	28,60	10,84(5.12)
Optimized HyperCuts[14]	FPGA	9603	63,73	80,23
Hybrid Scheme[11]	FPGA	9603	25,51	80,00
TTϵ[5]	FPGA	9603	22,62	134,0
TTϵ-Ptrie=5[5]	FPGA	9603	46,54	134,0
Optimized HHSS	*FPGA*	*9603*	*29,47*	*128,00*

Optimized HHSS on the other hand, solves the latency problem of TTϵ using a comparable amount memory and without sacrificing from throughput performance.

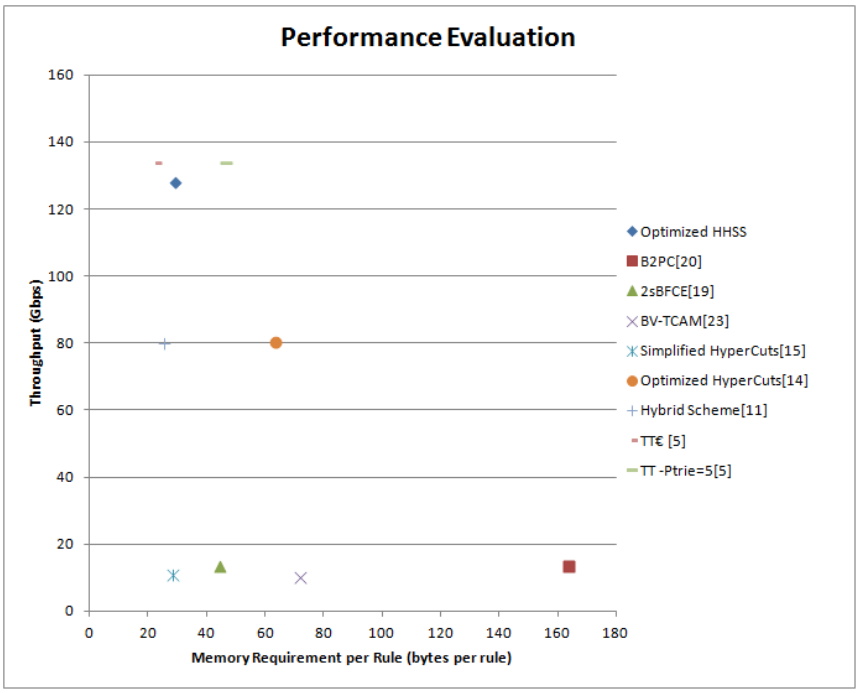


Figure 5.3: Overall performance comparisons.

CHAPTER 6

CONCLUSION

The packet classification engines in flow aware routers are required to satisfy the increasing needs of current internet structure due to increase in real time applications. The packet classification has to be as fast as line rates in order not to disturb the overall flow. In addition to fast processing, it has to minimize the memory storage due to limited hardware resources and continuous increase in the rule set sizes. The latency of a packet processing is also required to guarantee the delay limitations of the network.

This study has focused on changing the memory structure of HHSS [5] to decrease the latency value without disrupting the memory storage or throughput. The use of differently sized memory blocks on the trie structure has decreased the tree depth while preserving the throughput and memory storage requirement values. In other words, our work has solved the deep pipeline problem leading to smaller latency value.

Our design benefits the parallel pipelined architecture implemented on FPGA in order to achieve high speed packet processing. The proposed algorithm supports approximately 128 Gbps throughput and can handle 10K rules with only 28 KB memory requirement. Comparing with the state of art packet classification algorithms, our design offers a significant performance improvement without long latency of packet processing.

Despite these benefits, the algorithm has certain drawbacks. The incoming updates are buffered in an external hardware. In case the buffer is full, the tree structure has to be reconstructed and the FPGA has to be reconfigured.

The present work uses the hierarchical tree structure to solve the packet classification problem, however, the dynamic memory structure it suggests can also be used in other classification algorithms to reduce the memory consumption. Moreover, the look up and virtual router algorithms that has limited memory resources might take advantage of such multiple memory blocks.

The packet classification problem in routers is still a hot topic with the increasing and changing demands of the Internet. This research mainly has been focused on the packet classification for IPv4 packet structure on flow aware routers. The same algorithm is expected to provide acceptable speed, memory storage and latency values also for IPv6 as in the case of IPv4. Hence, a future study might include experimenting the behavior on sample rule sets for IPv6. Furthermore, next generation Open

Flow structures that require 12 tuple packet classification need even more memory efficient packet classification for storing all 12 headers. Decision tree based structure might be more advantageous, since aggregation of the independent searches will be quite complex when the number of headers is increased to 12. Our work as a fast and memory efficient solution for packet classification for 5 tuple might be restructured as a solution for 12 tuples, too.

REFERENCES

[1] W. U. Applied Research Lab. Packet classification filter sets. `http://www.arl.wustl.edu/~hs1/`, 2007.

[2] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: is there an alternative to cams? In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 53–63 vol.1, 2003.

[3] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 64–74 vol.1, 2003.

[4] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking packet forwarding hardware. In *Proc. Seventh ACM SIGCOMM HotNets Workshop*, 2008.

[5] O. Erdem, H. Le, and V. Prasanna. Hierarchical hybrid search structure for high performance packet classification. In *INFOCOM, 2012 Proceedings IEEE*, pages 1898–1906, 2012.

[6] O. Foundation. Openflow switch specification, version 1.0.0, 2009.

[7] P. Gupta. *Algorithms for routing lookups and packet classification*. PhD thesis, Stanford Univesity, 2000.

[8] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *Micro, IEEE*, 20(1):34–41, 2000.

[9] P. Gupta and N. McKeown. Algorithms for packet classification. *Netwrk. Mag. of Global Internetwkg.*, 15(2):24–32, Mar. 2001.

[10] G. Jedhe, A. Ramamoorthy, and K. Varghese. A scalable high throughput firewall in fpga. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 43–52, 2008.

[11] W. Jiang and V. Prasanna. Scalable packet classification: Cutting or merging? In *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, pages 1–6, 2009.

[12] W. Jiang and V. Prasanna. Scalable packet classification on fpga. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(9):1668–1680, 2012.

[13] W. Jiang, V. Prasanna, and N. Yamagaki. Decision forest: A scalable architecture for flexible flow matching on fpga. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 394–399, 2010.

[14] W. Jiang and V. K. Prasanna. Large-scale wire-speed packet classification on fpgas. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 219–228, New York, NY, USA, 2009. ACM.

[15] A. Kennedy, X. Wang, Z. Liu, and B. Liu. Low power architecture for high speed packet classification. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 131–140, New York, NY, USA, 2008. ACM.

[16] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *In ACM SIGCOMM*, pages 203–214, 1998.

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

[18] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

[19] A. Nikitakis and L. Papaefstathiou. A memory-efficient fpga-based classification engine. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 53–62, 2008.

[20] I. Papaefstathiou and V. Papaefstathiou. Memory-efficient 5d packet classification at 40 gbps. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1370–1378, 2007.

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

[22] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *In Proceedings of ACM SIGCOMM*, pages 213–224, 2003.

[23] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 238–245, New York, NY, USA, 2005. ACM.

[24] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, Oct. 1998.

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

[26] D. Taylor and J. Turner. Scalable packet classification using distributed crossproducing of field labels. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 269–280 vol. 1, 2005.

[27] D. E. Taylor. Survey and taxonomy of packet classification techniques. volume 37, pages 238–275, New York, NY, USA, Sept. 2005. ACM.

[28] P. F. Tsuchiya. A search algorithm for table entries with non-contiguous wildcarding. Unpublished Report.

[29] J. M. Wagner, W. Jiang, and V. K. Prasanna. A scalable pipeline architecture for line rate packet classification on fpgas. In *Proceedings of the 21st IASTED International Conference*, volume 668, page 284, 2009.