SCALABLE PACKET PROCESSING FOR HIGH-SPEED NETWORKS

By

YAN SUN

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2011

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of YAN SUN find it satisfactory and recommend that it be accepted.

Min Sik Kim, Ph.D., Chair

Carl H. Hauser, Ph.D.

Jose G. Delgado – Frias, Ph.D.

ACKNOWLEDGEMENTS

Firstly, I would like to acknowledge my thesis advisor Professor Min Sik Kim, who provided me with the opportunity and resources for accomplishing my graduate work. It is his hearty encouragement and inspiring instructions that have given me the most confidence in my scientific research and have guided me progressing along a forward path.

I acknowledge all the Network Research Lab group members I have encountered for their individual helps to my graduate work. I thank my thesis committee members, Professor Carl H. Hauser and Professor Jose G. Delgado-Frias, for their instructional questions and advices during my thesis work.

Finally, I am indebted to my parents, and my wife Wenya Lang, for their hearty support and encouragement.

SCALABLE PACKET PROCESSING FOR HIGH-SPEED NETWORKS

Abstract

by Yan Sun, Ph.D
Washington State University
December 2011

Chair: Min Sik Kim

Rapid expansion of the Internet has led to the exponential growth of requirement for high-speed packet processing, including both header processing and payload processing. In network applications such as Routers, Firewalls and Intrusion Detection Systems (IDS), there are three important issues needed to be designed efficiently to support today's high speed network: IP route lookup, packet classification and deep packet inspection. And these three procedures are usually implemented as different components, which make packet processing too slow to meet high-speed network's requirements. So how to design efficient packet processing components is very important. Special hardware, such as TCAMs, are widely used networking applications to achieve high throughput, but their disadvantages such as high cost and high power consumption usually limit their application. We focus on these issues and design efficient schemes to solve problems in today's and tomorrow's networks. Our approaches are based on the observation that all of these components contain redundancy and we can reduce the hardware consumption and increase the throughput if we can remove the redundancy efficiently. Furthermore, we remove the redundancy between different components to further improve the performance and we propose a new integrated architecture which integrates these three components efficiently.

In order to design an efficient combined security gateway system, we first focus on how to design efficient three individual components, including routing table lookup, packet classification

and deep packet inspection, and then propose an efficient combination approach to further reduce hardware consumption and increase overall throughput. Our approaches are mainly based on Ternary Content Addressable Memories (TCAMs) and the system can be easily implemented in a single FPGA. We focus on how to reduce redundancy in both individual components and integrated architecture to not only reduce the hardware consumption but also increase the overall throughput. The simulation results on both proposed individual components and integrated architecture show that we can achieve expected goals.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER one

INTRODUCTION

Rapid expansion of the Internet has led to the exponential growth of requirement for high-speed packet processing, including both header processing and payload processing. In network applications such as Routers, Firewalls and Intrusion Detection Systems (IDS), there are three important issues needed to be designed efficiently to support today's high speed network: IP route lookup [1,2], packet classification [3,4] and deep packet inspection [5]. And these three procedures are usually implemented as different components, which make packet processing too slow to meet high-speed network's requirements. So how to design efficient packet processing components is very important. Special hardware, such as TCAMs, are widely used networking applications to achieve high throughput, but their disadvantages such as high cost and high power consumption usually limit their application. We focus on these issues and design efficient schemes to solve problems in today's and tomorrow's networks. Our approaches are based on the observation that all of these components contain redundancy and we can reduce the hardware consumption and increase the throughput if we can remove the redundancy efficiently. Furthermore, we can remove the redundancy between different components to further improve the performance and we propose a new integrated architecture which combines these three components efficiently.

Current routers receive packets and check them, then forward them to their corresponding next destination on the Internet, so they typically focus on how to perform IP route lookup and the destination IP address in the packets need to be checked. The firewalls usually perform as traffic filters, and they usually check 5-tuple in the packet headers to decide whether to block or accept a packet. The Intrusion Detection Systems (IDS) usually focus on checking the payload of packets.

These three kinds of devices usually work independently, and routers can not support network security requirements in this situation. However, there is a trend that one such device integrates more than one function. For example, in the latest gateway system, it usually contains firewall functions besides routing function. Traditionally, the firewall only performs as a traffic filter, but one of the more recent innovations in firewall is the application of deep packet inspection, which is usually performed in Intrusion Detection System (IDS) before. Unified Threat Management (UTM) is a good example of an important tread in networks: combining network functions with comprehensive security functions. UTM integrates many security products in a single application to provide better security checking services. So our goal is to design a security gateway, which can support all these three important tasks in a single device. Unfortunately, current research only focuses on a single topic, such as how to design an efficient IP route lookup, how to design efficient packet classification or how to design efficient deep packet inspection. So we will focus on the combined security gateway which can provide greater functionality than separate individual devices because combined device can not only increase overall performance but also provide better understanding of the inspected packets to achieve better security goal. There are two challenges of improving the efficiency of the security gateway, first, how to design efficient three individual components, second, how to combine them effectively. So we focus on individual components first and then focus on combination scheme. Our main contribution is to propose a new efficient integrated packet processing architecture instead of simply combining independent devices together.

In the IP route lookup design, we propose two approaches to design efficient routing table lookups. In the first approach, our goal is to reduce the usage to TCAMs in IP route lookup without modifying the TCAM circuit itself. We achieve it by adding a small extra logic circuit, which occupies negligible area and consumes significantly less power compared with TCAMs. We identify two kinds of redundancy in the TCAM-based IP route lookup. Then we build a hybrid

scheme that eliminates such redundancy and take advantages of both Binary and Ternary CAMs. In the second approach, we present a novel approach to IP route lookup using TCAMs to save memory usage, increase update speed and improve the throughput of longest prefix matching process.

In the packet classification design, we propose an efficient TCAM-based packet classification algorithm by building minimum Range Tree, which consists of two parts: overlapping removal in a rule set using a tree representation of rules, and the postprocess optimization. The proposed algorithm first removes redundant rules and combines overlaying rules to build an equivalent, smaller rule set for a given packet classifier, and then removes priority encoder and memory access process based on the properties of non-overlapping rule set. Based on the non-overlapping rule set, we propose one-directional range extension algorithm and bidirectional range extension algorithm to further reduce the TCAM consumption, and we also propose a fast TCAM entry update approach based on the non-overlapping rule set.

In the deep packet inspection design, we first propose an efficient hierarchical NFA-based pattern matching approach which serves to exclude most packets from full regular expression matching leaving only a small percentage to be fully checked in the regular expression matching process. This hierarchical pattern matching improves throughput for the average case. Then we propose a hybrid algorithm that combines both NFA and DFA. It divides a complex regular expression and configures them into multiple cores to take advantage of available parallelism provided by multi-core processors. We also propose a DFA-based regular expression matching on compressed traffic at last.

In the combination design, we combine these three components together to further improve the overall performance and further reduce the hardware consumption. We share some hardware resources between routing lookup and packet classification to not only reduce the hardware consumption but also reduce the packet processing latency.

There is a trend to integrate multiple devices into a single device to reduce cost, and in this thesis, we show how to design efficient network components and how to integrate them into an efficient system. This approach can also be used to integrate other network components to further improve the overall performance.

CHAPTER two

IP ROUTE LOOKUP

2.1 Introduction

Rapid expansion of the Internet has led to the exponential growth of routing tables in routers, and the longest prefix matching in IP route lookup is often the bottleneck in today's routers with such large routing tables. Thus, designing an efficient scheme to perform longest prefix matching operations is a critical problem in high-speed routers. A routing table in a router stores variable-length IP address prefixes and corresponding outgoing ports. Table 2.1 shows a simple routing table with four IP address prefixes.

Table 2.1: A Simple Routing Table

Destination IP Address	Mask	Port
152.168.22.0	/24	5
152.168.30.0	/24	1
132.165.0.0	/16	8
122.128.0.0	/18	4

When a packet arrives, a router searches for the longest prefix match for the destination IP address of the packet, and then retrieve the corresponding outgoing port. Since this procedure has been the bottleneck of routers' performance, many approaches have been proposed by researchers, both software-based and hardware-based ones.

Ternary Content Addressable Memories (TCAMs) have been widely adopted by routers to improve the speed of the longest prefix matching. They allow the "don't care" state to be stored in each memory cell as well as binary states 0 and 1. A memory cell in a "don't care" state matches

both 0 and 1 in the corresponding input bit. A TCAM-based routing table is extremely fast because it allows the input key to compare with all the prefixes stored in the TCAM simultaneously and retrieve the result in a single clock cycle. The architecture of a TCAM used in the longest prefix matching is shown in Fig. 2.1.

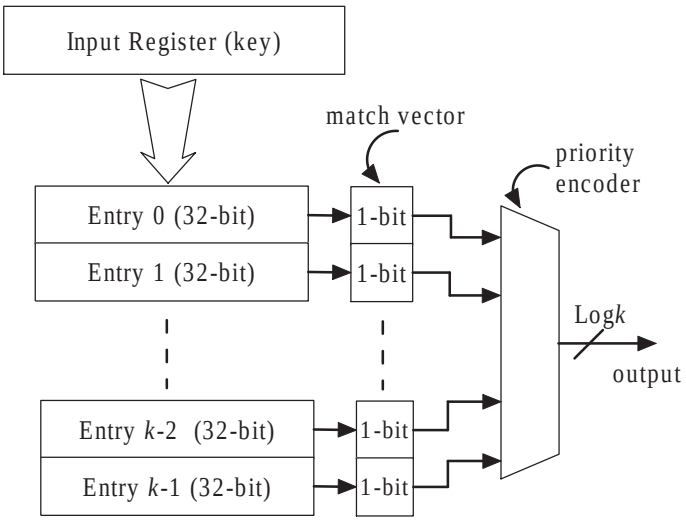


Figure 2.1: TCAMs used in the Routing Tables

A key (destination IP address) is stored in the input register and each prefix is stored in a single entry. The key compares with all the prefixes in parallel and the results are stored in the match vector, where 1's represent the corresponding entries match the key, and the priority encoder chooses the longest prefix match. At last, the output signal is used to find the corresponding outgoing port. While the TCAM-based search is very fast, TCAMs have two major disadvantages: high cost and high power consumption. In fact, both of them result from the circuit complexity of each TCAM cell. The high power consumption also affects the total cost and performance of routers, not only because it increases the power supply and cooling costs but also it reduces the port density since more space is needed between ports for cooling purpose. Therefore, how to use TCAMs efficiently becomes a critical issue, and many methods have been proposed to reduce

TCAM requirements for a given set of prefixes. Compared with TCAMs, Binary CAMs (BCAMs) require fewer transistors and less power because no mask is needed and the comparison circuit is simpler. However, they can store only 0 and 1; they don't have the "don't care" state.

We propose two approaches to design efficient routing table lookups in this chapter. In the first approach, our goal is to reduce the usage to TCAMs in IP route lookup without modifying the TCAM circuit itself. We achieve it by adding a small extra logic circuit, which occupies negligible area and consumes significantly less power compared with TCAMs. We identify two kinds of redundancy in the TCAM-based IP route lookup. Then we build a hybrid scheme that eliminates such redundancy by dividing all the input keys into seven groups, each of which is handled separately, taking advantages of both Binary and Ternary CAMs. We evaluate our scheme by measuring savings in terms of the number of transistors when it is applied to real-world routing tables. In the second approach, we present a novel approach to IP route lookup using TCAMs to save memory usage, increase update speed and improve the throughput of longest prefix matching process. Most of researches focus on how to reduce the usage of TCAM entries and power consumption in TCAM-based route lookup. Our goal is to find out other problems and propose an efficient algorithm to solve them. We achieve it by dividing prefixes into two groups and treat them separately. We discussed three important issues still have not been well studied in TCAM-based IP route lookup including large RAM usage and long memory accesses, slow update process of routing table and unscalable problem. Based on the observation that all of them result from the sorted storage in TCAM entries, we store IP prefixes can be stored out of order in the first group and the remaining ones in the second group, and deal with them separately.

The remainder of this chapter is organized as follows. Section 2.2 surveys related work on longest prefix matching algorithms. Section 2.3 details the design of the proposed hybrid scheme for longest prefix matching in IP route lookup. Section 2.4 details the design of the proposed

scalable scheme for TCAM-based prefix matching in IP route lookup.

2.2 Related Work

There are three major categories of approaches are used for IP route lookup, including hash-based approaches, trie-based approaches and TCAM-based approaches. We will discuss them separately.

Hash-based solutions Many hash-based routing lookup table approaches have been proposed in [6–9]. These approaches hash prefixes and store them in a hash table for looking up. They are memory efficient but the hash-based approaches do not deal with variable lengths well, so they need to process prefixes with different lengths separately or expend short prefixes to long prefixes to reduce the number of unique prefix lengths. Some approaches use boom filters to filter some lengths of prefixes [10–13]. These approaches are efficient in large lookup table applications, however, multiple sets of bloom filters are needed to process different lengths of prefixes and the false positive in bloom filters increase the number of memory accesses.

Trie-based solutions Many trie-based longest prefix matching techniques have been proposed [14–20]. These techniques use tree-like architectures to store prefixes and corresponding output port information. They are easy to be implemented in general purpose processors or network processors and efficient in reducing memory consumption. However, these approaches usually need multiple memory accesses for a single input packet, they usually cannot meet the requirement of today's high-speed forwarding.

TCAM-based solutions Among hardware-based longest prefix matching techniques, the CAM-based ones dominate the high-speed router market, especially of multi-gigabits per second and

faster routers. A disadvantage is that the TCAM chips are expensive and power-hungry. Therefore, techniques have been proposed to use TCAM entries more efficiently or to reduce the required amount of TCAMs [21–26]. These approaches usually need to pre-process prefixes to compress them and then configure into TCAMs. However, these approaches should undergo a very complex update process, because they have to pre-process the set of prefixes again. Other approaches focus on the TCAM cell itself and try to reduce the number of transistors and power consumed by a single TCAM cell [27–29]. They usually need to modify the circuit within a TCAM cell or within a TCAM entry.

2.3 Hybrid CAM-Based Approach

Ternary Content Addressable Memories (CAMs) are widely used by high-speed routers to find matching routes in a routing table, because they enable the longest prefix matching operation to complete in a single clock cycle. However, they are costly and their power consumption is very high. In this section, we identify two kinds of redundancy in the usage of TCAMs in IP route lookup, and then propose a hybrid scheme which combines Binary CAMs and Ternary CAMs to reduce the total area and power consumption, exploiting the uneven distribution of IP prefix lengths in real-world IP routing tables. We also introduce shared memory blocks for further simplification of the lookup circuit. The simulation results show that our approach can save more than 50% of transistors in CAMs, compared with the traditional way in storing a set of real-world routing tables, and that it reduces the critical path in IP route lookup significantly.

2.3.1 Hybrid Approach Using BCAMs and TCAMs

(1). Redundancy in Using of TCAMs for IP Route Lookup

Table 2.2: Four Real-World Routing Tables

Name of Routing Tables	Date of Collection	Number of Prefixes
Oix-1997	11/08/1997	19,017 $(10^{4.28})$
Oix-2001	08/01/2001	53,842 $(10^{4.73})$
Oix-2005	08/01/2005	107,679 $(10^{5.03})$
Oix-2009	08/19/2009	10,471,325 $(10^{7.02})$

Because IP route lookup is longest prefix matching, each 32-bit entry in a TCAM consists of the prefix part and the following "don't care" bits. This implies two kinds of redundancy as follows:

- Each bit in the prefix part has two possible values only: 0 and 1; there is no "don't care" bit in the prefix. Therefore, a Binary CAM should suffice to match the prefix.
- No comparison is needed for the "don't care" suffix, since they always match the input.

Based on these observations, we propose a new approach to mitigate both kinds of waste.

(2). Real-world IPv4 Prefix Distribution

In this section, we analyze the distribution of real-world prefixes. We collect the routing tables from the Route Views project [30]. In order to ensure that the characteristics of the distributions are not specific to some particular routers or time intervals, we inspect many routing tables and finally select four typical sets of routing table information from year 1997 to year 2009, where each set consists of all the data in a single typical day. The information regarding the selected routing tables are summarized in Table 2.2. The numbers of prefixes of the selected routing tables are plotted in Fig. 2.2. (Note the logarithmic scale on the vertical axis).

We can see that the number of prefixes is growing exponentially, and is even growing faster in recent four years. This indicates that the longest prefix matching will remain to be the bottleneck in high-speed routers.

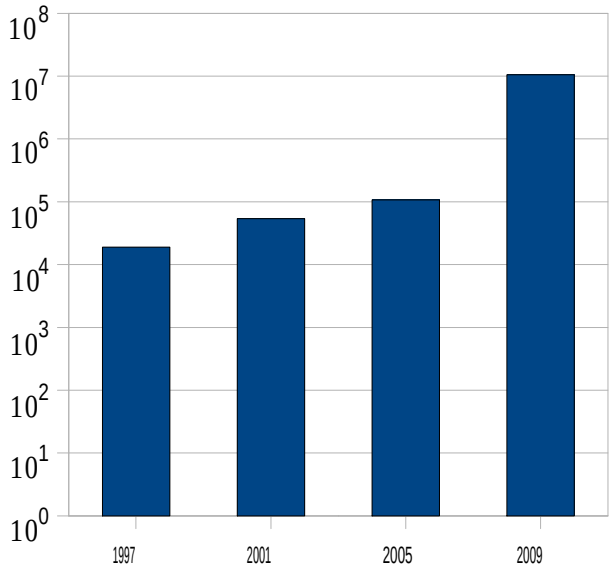


Figure 2.2: The Number of Prefixes in a Real Typical Network

The distributions of prefixes of these four sets of routing tables are shown in Fig. 2.3. The horizontal axis is the length of prefixes and the vertical axis is the number of prefixes. We can see that the distributions in the four different years are similar, and the trend is that the percentages of prefixes shorter than 16-bit and longer than 24-bit are decreasing. It is this trend that provides room to design a more efficient algorithm. We can see that in recent routing tables, the 24-bit Class C Prefixes dominate the number of prefixes (about 50% alone), and over 90% of the prefixes are between 18-bit and 24-bit long. Based on these observations, we propose a scheme to utilize CAMs more efficiently.

(3). Proposed Scheme

Based on the observations of TCAM redundancy and real-world IPv4 prefix distribution discussed above, we propose our approach to reducing the usage of TCAMs in IP route lookup. Given IP address prefixes with the maximum length of 32 bits, we divide them into seven cate-

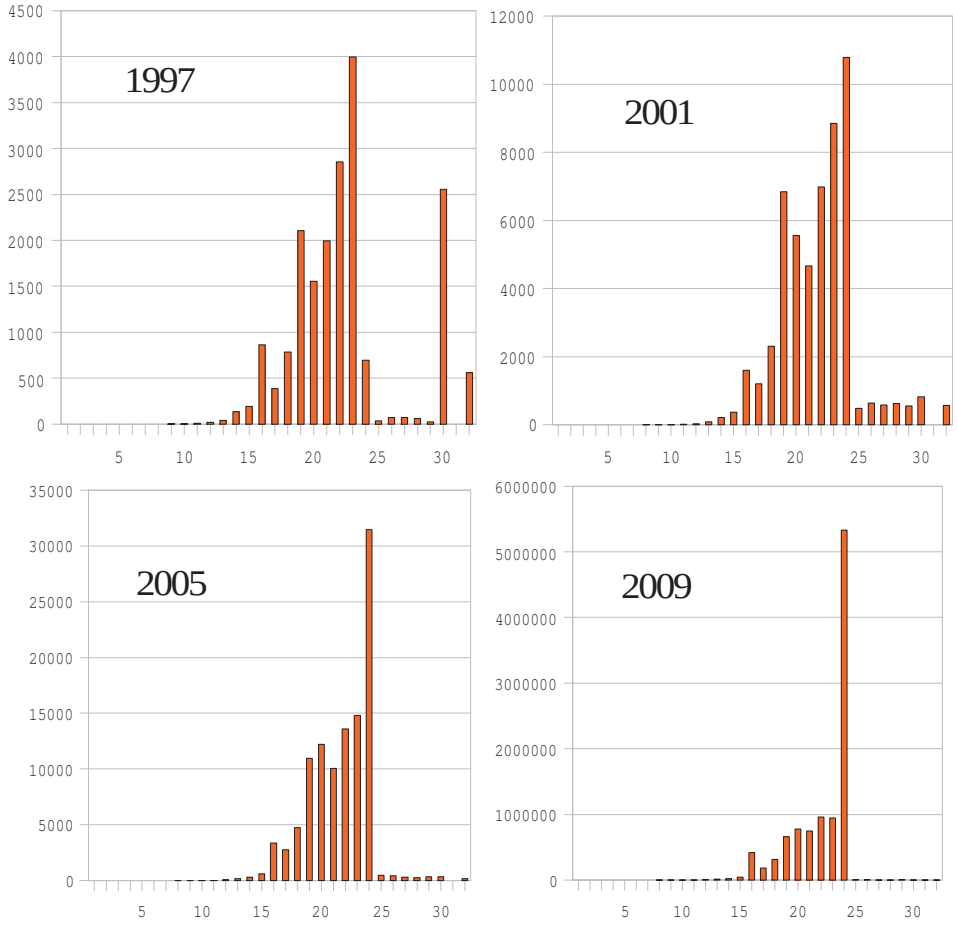


Figure 2.3: The Distribution of Prefix Lengths in the Past Twelve Years

gories, P_1, P_2, P_3, P_4, P_5, P_6, and P_7, according to the prefix lengths. All 8-bit long prefixes are added to P_1, 9 to 15-bit long prefixes to P_2, 16-bit to P_3, 17 to 23-bit to P_4, 24-bit to P_5, 25 to 31-bit to P_6, and 32-bit to P_7. For the categories P_1, P_3, P_5, and P_7, the length of prefixes is unique, and thus Binary CAMs, one for each category, should suffice to perform prefix matching. On the other hand, for the categories P_2, P_4, and P_6, we need to split each prefix into the fixed-length prefix part and the remainder part, the former containing binary values which can be stored in a Binary CAM, and the latter containing "don't care" bits which should be stored in a Ternary CAM. For example,

since P_2 has prefixes of length between 9 and 15, the first 9 bits are stored in a Binary CAM while the remaining bits are stored in a Ternary CAM. To combine the results from two CAMs, we need to perform an AND operation between the two to see whether an entry matches the destination IP address of a given packet. For further improvement, we pipeline our scheme as shown in Fig. 3.30.

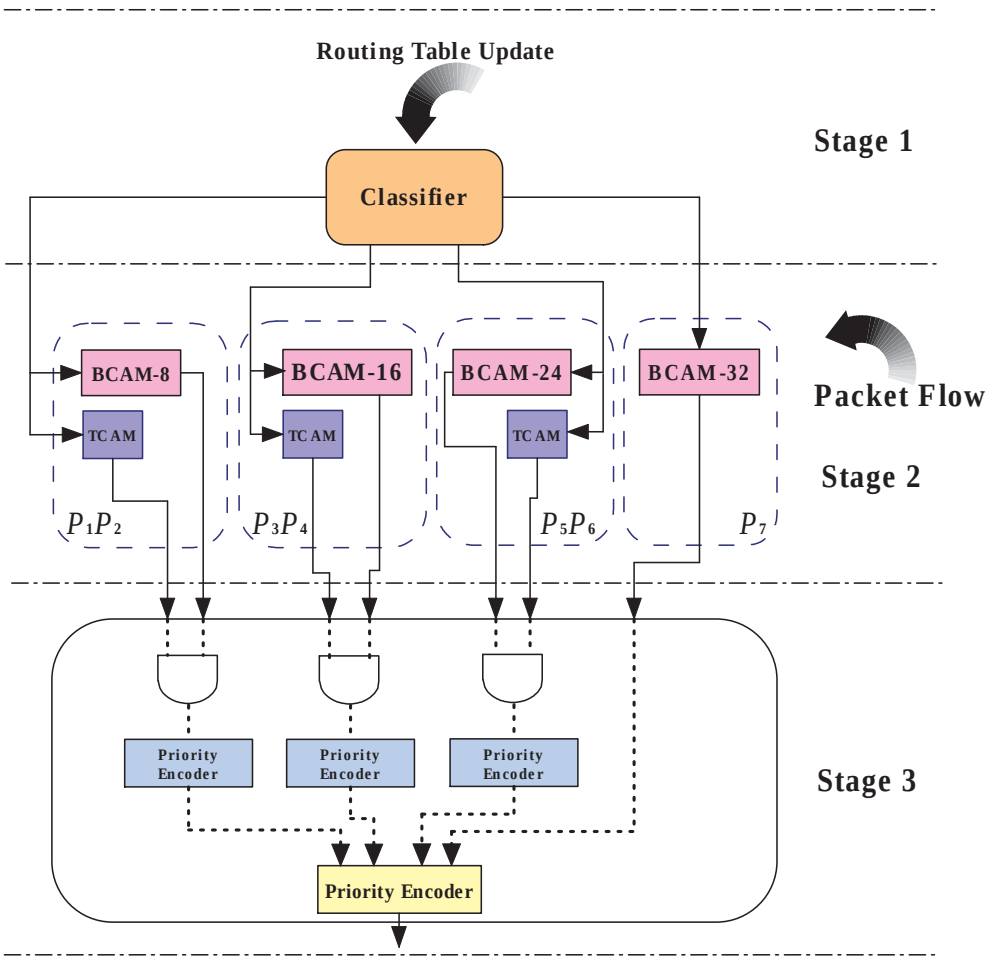


Figure 2.4: The Architecture of the Hybrid Approach

In the first stage, the routing table update prefixes are classified into seven categories. The second stage consists of BCAMs and TCAMs, and both updating the routing table and handling incoming packets are performed in this stage. All the TCAM blocks are 7 bits wide and the

four BCAM blocks are 8 bits, 16 bits, 24 bits, and 32 bits wide, respectively. Because a 32-bit destination IP address is divided into two parts if the length of prefix is not a multiple of 8, we need to combine the results of these two parts together using AND logic in the third stage to get the matching result. As shown in Fig. 3.30, the seven categories are divided into four groups, G_0, G_1, G_2, and G_3, containing P_1 and P_2, P_3 and P_4, P_5 and P_6, and P_7, respectively. Based on the longest prefix matching policy, each group chooses the matching bit with the highest priority, G_3 being the highest and G_0 the lowest. The three Priority Encoders used by the three groups can operate in parallel to reduce the latency of stage 3.

For each routing lookup entry, we need at most a 7-bit-wide TCAM. When the length of the prefixes is a multiple of 8, no TCAM is needed and only BCAM entries are used. When the length of entries is shorter or the TCAM is replaced by a BCAM, the memory access latency is reduced, especially for write operations, which result in higher clock speed.

One of major disadvantages of TCAM-based approaches is that all the prefixes stored in TCAM must in the order of increasing prefix length because the longest prefix policy is implemented using a priority encoder. In our approach, sorting overhead is significantly lower because four smaller sorting operations are performed in parallel.

In Fig. 3.30, different groups use different TCAM blocks. For further improvement, we can use a single TCAM block with dynamic boundaries between different groups.

(4). Shared CAM

The distribution of prefix lengths varies. Spatially, core routers have more short prefixes than edge routers. Temporally, the percentages of entries in the four groups change over time as shown in Fig. 2.3. To make our approach more flexible and efficient in utilizing CAM resources, we introduce the shared CAM mechanism, where four groups share the same memory. In this mechanism, we let entries from P_3, P_4, and P_7 share a BCAM block, and entries from P_1, P_2, P_5,

and P_6 share another BCAM block. For the first share group, each BCAM entry has 16 bits, and each element in P_3 and P_4 only uses one 16-bit entry while each element in P_7 uses two adjacent 16-bit entries. They are depicted in Fig. 2.5. A 32-bit IP address is stored in a pair of 16-bit BCAM entries, starting from the pair with the highest address. A 16-bit prefix is stored in a single 16-bit BCAM entry, starting from the entry with the lowest address.

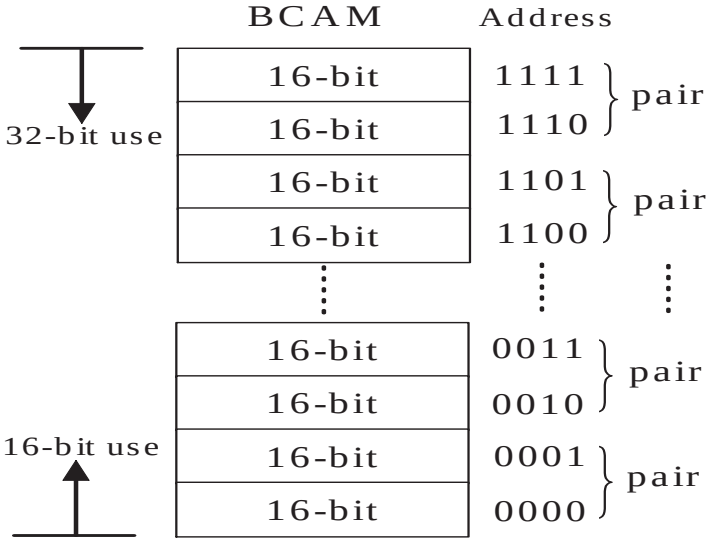


Figure 2.5: BCAM Shared by 32-bit and 16-bit Prefixes

Similarly, the 24-bit binary prefixes and 8-bit binary prefixes share the same resources. We use a 8-bit-wide BCAM block to locate both prefixes as shown in Fig. 2.6. A 24-bit prefix is be stored in a triple of three BCAM entries, starting from the triple with the highest address. A 8-bit prefix is stored in a single 8-bit BCAM entry, starting from the entry with the lowest address.

The two 8-bit entries in a single *pair* have different search operations, which are shown in Fig. 2.7. The value of "Sel" in Fig. 2.7 is based on whether this block is used by a 32-bit prefix.

Similarly, the three 8-bit entries in a $triple$ have different search operations, which are shown in Fig. 2.8. The value of "Sel" in Fig. 2.8 is based on whether this block is used by a 24-bit

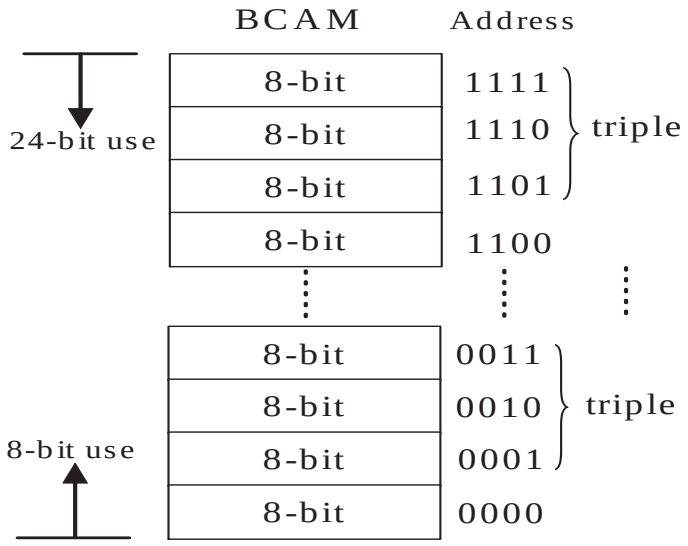


Figure 2.6: BCAM Shared by 24-bit and 8-bit Prefixes

prefix.

2.3.2 Evaluation

(1). Transistors Savings

In a Binary CAM entry cell, the content can be made up of binary bits, each of which has either 0 or 1. In a Ternary CAM cell, however, a third "don't care" state can be used as a bit value. An entry of a Ternary CAM stores content as a (value, mask) pair, where value and mask are W-bit numbers, requiring W storage cells for the value and additional W storage cells for the mask. Moreover, the matching circuitry is more complicated than that of a Binary CAM.

A typical TCAM cell requires six transistors as a SRAM cell. The same number of transistors are required to store the mask bit, and four transistors for the match logic. Thus, each TCAM cell requires 16 transistors, which is about 2.7 times of a typical SRAM cell. However, different techniques used by CAM manufactures result in different numbers [31]. For the evaluation of

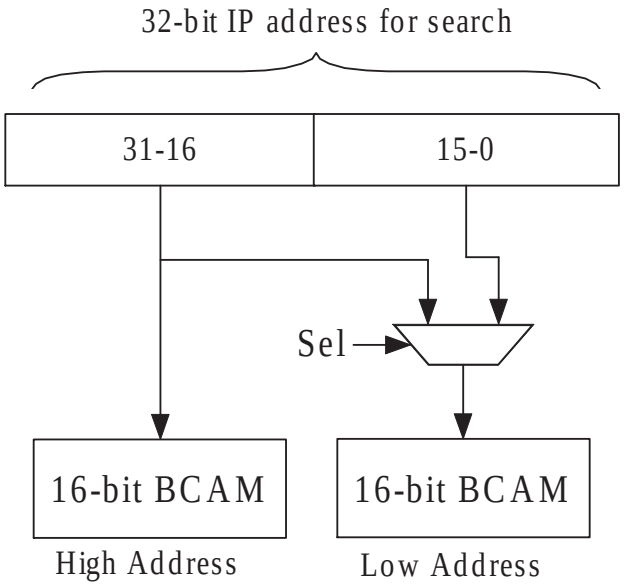


Figure 2.7: IP address selected by the shared BCAM for P_3, P_4 and P_7 in the Search Operation

our scheme in this section, we assume that the number of transistors and power consumption of a TCAM are two times as large as those of a BCAM cell on average.

Table 2.3 shows the transistors saved under different lengths of prefixes, compared with ordinary TCAM-based routing tables. On average, it can save about 60.7% of transistors.

(2). Simulation with Real-World IPv4 Prefix Distribution

We use the data collected in year 2009 shown in Fig. 2.3 to evaluate our approach with

Table 2.3: Transistors Saved

Prefix	BCAM length (bits)	TCAM length (bits)	Transistor Saved (%)
8-bit	8	0	87.5
9-bit to 15-bit	8	7	62.5
16-bit	16	0	75.0
17-bit to 23-bit	16	7	50.0
24-bit	24	0	62.5
25-bit to 31-bit	24	7	37.5
32-bit	32	0	50.0

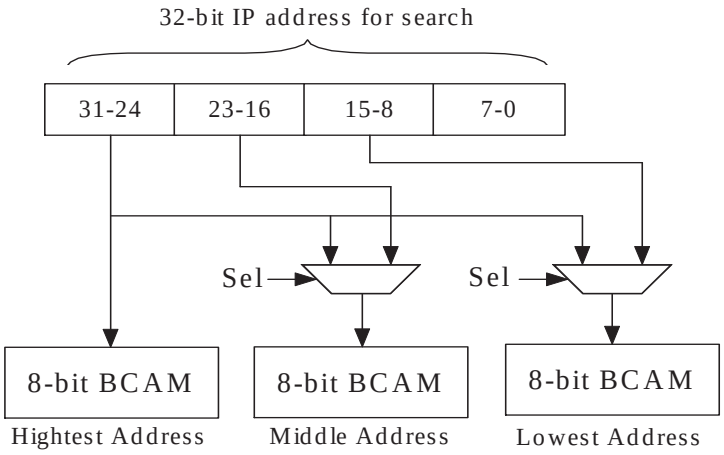


Figure 2.8: IP address selected by the shared BCAM for P_1, P_2, P_5 and P_6 in the Search Operation

Table 2.4: Transistors saved by our approach

Length of Prefix(bits)	Number of Prefix	Percentage(%)	Transistors Saved (%)
8	768	0.2368	87.5
9 to 15	93423	0.8922	62.5
16	419800	4.0090	75.0
17 to 23	4595478	43.8863	50.0
24	5326651	50.8689	62.5
25 to 31	24795	0.2368	37.5
32	258	0.0025	50.0

real-world IP address prefixes. We see that the 24-bit Class C Prefixes dominate the number of prefixes (about 50% alone), and over 90% of the prefixes are between 18-bit and 24-bit. Under this condition, the traditional way requires 10,471,325 32-bit TCAM entries. The BCAM and TCAM usage by our approach is shown in Table 2.4.

We calculate the total transistors saved by our approach using following equation:

$$\text{TotalTransistorsSaved} = \sum_{i=1}^{7} \text{TransistorsSaved}(i) \times \text{Percentage}(i) \tag{2.1}$$

where i represents each prefix group shown in Table 2.4.

According to data shown in Table 2.4 and the equation above, we can save 57.6% transistors by reducing and replacing the TCAMs in IP route lookup, and the additional logic introduced by our approach is much smaller than typical logic in TCAMs. Thus we conclude that our approach can reduce both the area and power consumption significantly.

Our approach can be used together with existing algorithms that solve other problems of CAM-based IP route lookup, such as dealing with more routing table entries than TCAM entries. For example, our approach can be combined with software-based approaches to reduce the number of required CAM entries, because our approach is designed to be a drop-in replacement of TCAM-based longest prefix matching.

Furthermore, our approach enables different groups of prefixes to update operate in parallel, which leads to further increase of the throughput of packet processing. Using the same data collected in year 2009, our approach needs about 51% of total clock cycles used by the traditional way to update these prefixes.

Finally, using narrower TCAMs can help increase the clock speed in ASIC or FPGA. In our algorithm, the critical path is the delay of the 32-bit-wide BCAM, which is about half of the delay of a 32-bit-wide TCAM in the SMIC 0.13 μm CMOS technology; we get the similar results using FPGAs. Therefore, the critical path of the ACL subsystem can be shorter compared with the traditional approach.

2.3.3 Summary

In this section, we present a hybrid approach to IP route lookup using Binary CAMs to save memory usage and improve the throughput of longest prefix matching process. We treat prefixes with different lengths separately in parallel, and use different types of CAMs to take advantage of their characteristics. The simulation results show that our approach saves 57.6% of transistors, reducing the area and power consumption significantly. Our approach can double the throughput of the longest prefix matching under the same clock speed. Furthermore, the clock speed can also be increased because the paths traversing through CAMs are shorter. Because our approach provides the same interface as that of the traditional TCAM-based approach, it can be used with other preprocessing-based approaches together to increase performance further. Our approach can be easily extended to be used in IPV6 applications based on the properties of routing tables in IPV6.

2.4 Scalable TCAM-Based Route Lookup

Ternary Content Addressable Memories (TCAMs) are widely used by high-speed routers to find matching routes in a routing table. TCAMs are very fast because all the TCAM entries work in parallel and can perform the longest prefix matching operation in a single clock cycle. However, they are costly and power hungry because of the complexity of hardware circuit. Some approaches have been proposed to solve these problems, but some issues still have not been well studied. First, large priority encoders used after TCAMs consume a lot hardware resources and increase the overall latency. Second, the memory accesses after TCAMs match operations often take down the high speed of TCAMs. Third, the prefixes must be sorted according to lengths in

decreasing order, which makes the update process of routing table very slow. Last, even though the TCAMs are pretty fast, they can only perform one prefix match at one time, which makes them unscalable. In this section, we first discuss these problems and propose an efficient algorithm to solve them and the simulation results show that our approach can reduce the usage of TCAM entries and increase the throughput significantly. Furthermore, the update process speed also has been increased significantly compared with the traditional way of sorting a set of real-world routing tables.

While the TCAM-based search is very fast, TCAMs usually have some major disadvantages such as well studied problems: high cost and high power consumption. In fact, both of them result from the circuit complexity of each TCAM cell. A typical TCAM cell requires two SRAM cells to store both value bit and mask bit, and four transistors for the match logic. A typical SRAM cell requires six transistors, which means each TCAM cell requires 16 transistors, which is about 2.7 times of a typical SRAM cell [32]. The high power consumption also affects the total cost and performance of routers, not only because it increases the power supply and cooling costs but also it reduces the port density since more space is needed between ports for cooling purpose. For the overall power consumption, a single TCAM chip usually consumes more than 20W of power and we should use multiple TCAM chips for a large routing table, however, a single line card only allows no more than a few hundred watts [11], so there are very limited power budget left for other components when using TCAM chips. And some solutions have been proposed. Such as divide the TCAM into several small blocks and only a subset of them will be triggered each time. However, these approaches are still not efficient enough to meet fast growing routing lookup tables especially when moving to IPv6, which consumes four times as long as a prefix in IPv4.

Therefore, how to use TCAMs efficiently becomes a critical issue. Furthermore, there are some other important drawbacks have not been well studied in TCAM-based approaches. For

instance, in IP route lookup applications, large priority encoder and the memory accesses to fetch the corresponding output port usually take down the high speed of TCAMs and consume a lot of hardware resources; and the update of a TCAM entry usually takes much longer than a parallel search process in TCAMs, combined with the requirement that all the prefixes stored in TCAMs must be sorted according to lengths in decreasing order, which makes the update of routing table very slow, especially in today's increasing route table update frequency. Furthermore, even though the TCAMs are pretty fast, they can only perform one match at one time, which makes them unscalable in today's high throughput requirement. last but not least, a large priority encoder is needed to select the matched TCAM entry with the highest priority, and such a large priority not only consumes much hardware area but also increase the latency of the whole process.

In this section, we first discuss these problems which have not attracted enough attention and propose an efficient algorithm to solve these problems. We divide all the prefixes into two groups, and there is no overlap in the first group, which provides more room to solve these problems. The simulation results show that our approach can reduce the usage of TCAM entries and increase the throughput significantly, and the power consumption can also be reduced with the same throughput. The update process speed also has been increased significantly compared with the traditional way of sorting a set of real-world routing tables.

2.4.1 Problems in TCAM-Based Route Lookup

We generalize problems in TCAM-based Rout Lookup and some of them have not been attract enough attention from research community. So we analyze four types of drawback of TCAMs especially in IP route lookup as follows and the last three types are not well studied and our approach focuses on how to solve these problems.

(1). Large Priority Encoder

In TCAM-based longest prefix matching applications, large priority encoders must be used together with TCAMs, the large priority encoders consume a lot hardware resources and increase the overall latency. The logic of an 8 to 3 priority encoder is shown below:

- $Y_6 = I_7$
- $Y_5 = \overline{I_7} \cdot I_6$
- $Y_4 = \overline{I_7} \cdot \overline{I_6} \cdot I_5$
- $Y_3 = \overline{I_7} \cdot \overline{I_6} \cdot \overline{I_5} \cdot I_4$
- $Y_2 = \overline{I_7} \cdot \overline{I_6} \cdot \overline{I_5} \cdot \overline{I_4} \cdot I_3$
- $Y_1 = \overline{I_7} \cdot \overline{I_6} \cdot \overline{I_5} \cdot \overline{I_4} \cdot \overline{I_3} \cdot I_2$
- $Y_0 = \overline{I_7} \cdot \overline{I_6} \cdot \overline{I_5} \cdot \overline{I_4} \cdot \overline{I_3} \cdot \overline{I_2} \cdot I_1$
- $O_2 = Y_6|Y_5|Y_4|Y_3$
- $O_1 = Y_6|Y_5|Y_2|Y_1$
- $O_0 = Y_6|Y_4|Y_2|Y_0$

And "I_i" is input, "Y_i" is temporary value, "$\overline{I_i}$" is negation logic, "|" is OR logic, "$\cdot$" is AND logic and "O_i" is output. We can see that the priority encoder becomes complex when the input is very large. In [33], the 32 bits priority encoder consumes 1106 transistors and a maximum power consumption of 13.8 mW, and the latency is about 1.5 ns using 0.15 μm TSMC CMOS technology. For thousands of TCAM entries, the latency of the priority encoder will be longer than the TCAM matching process, which makes the priority encoder the bottleneck of the routing lookup.

(2). Slow Memory Access

After finding a match in the TCAM entries, we need to access SRAM or DRAM to retrieve the corresponding output port. The rapid growth of global routing tables has increased the usage of large RAM, which results in both slow memory access and high cost, and the memory accesses often take down the high speed of TCAMs. Then how to solve this problem is important.

(3). Slow Update

Due to the IP route lookup is the longest prefix matching process, the prefixes must be sorted in prefix length decreasing order, so updating a TCAM entry usually affects other TCAM entries, which makes the update of routing table very slow. With the increasing update frequency of IP route lookup table, the overall performance declines dramatically.

(4). Only Process One Match at a Time

Even though the TCAMs are pretty fast, they can only process one match at a time, which makes TCAM-based approaches unscalable. The naive approach to increase throughput is to use multiple sets of TCAMs to work in parallel and each set of TCAMs contains the whole IP route table, but this approach consumes too much TCAMs. So how to increase TCAM through using reasonable TCAM entries is very important.

2.4.2 Proposed Algorithm to Reduce Drawbacks of TCAM-Based IP Route Lookup

(1).Separate IP Prefixes into Two Groups

Because the IP route lookup is the longest prefix matching process, the IP prefixes must be sorted in prefix length decreasing order. This will result in many problems such as slow update and it makes the route lookup inflexible. Our approach is based on the following two observations: first, changing the order of most of the IP prefixes does not change the rule set's semantics because they don't overlap with others or still have higher priority in new positions, second, we can design

much more efficient algorithm if we can store the IP prefixes out of order in the TCAM because there is no strict requirement any more. Based on these observations, we separate the prefixes into two groups, the first group contains IP prefixes have at least one of the following properties:

(i). It does not overlap with other IP prefixes;

(ii). It overlaps with some other IP prefixes, and it has higher priority than all other overlapped prefixes;

(iii). It overlaps with some other IP prefixes, and it has the same decision as the IP prefix with the highest priority in these IP prefixes

Then the remaining IP prefixes belong to the second group. This strategy ensure the maximum number of IP prefixes can be stored out of order in the first group because any more IP prefix addition to the first group will lead to overlap between IP prefixes in the first group. In the first group, there is no overlap between any two IP prefixes, so when a destination IP address comes in, there is at most one IP prefix matches. And we store the second group in a small set of TCAMs, when there is no match found in the first group, the output of the TCAMs is selected. The main architecture is shown in Figure 2.9.

In Figure 2.9, the upper TCAM block stores the first group of IP prefixes out of order and the lower TCAM block stores the second group of IP prefixes in order. Because there is no overlap between any two IP prefixes in the first group, there is at most one match in the upper TCAM block then only an ordinary encoder is used by the upper TCAM block, this does not only reduce the circuit complexity of a single bit in encoder but also reduce the circuit complexity and timing delay of long-length encoder, that because ordinary encoders do not need to consider priority compared with priority encoder. Actually, we can use a single TCAM instead of two blocks and just separate them in high level.

How to separate the IP prefixes into two groups as discussed above is an important issue.

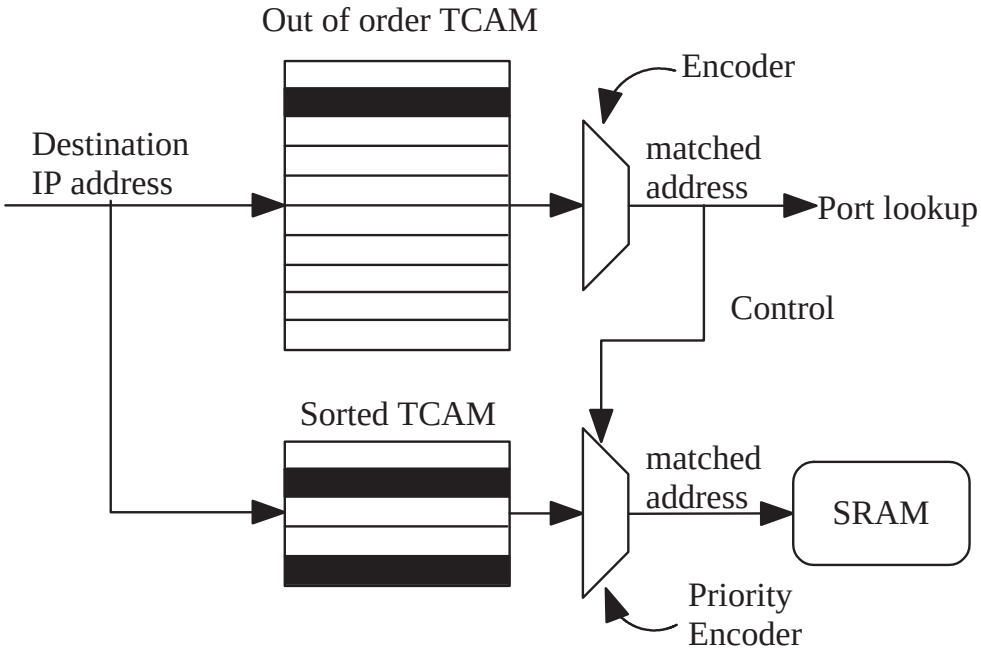


Figure 2.9: The proposed architecture using TCAMs in the Routing Tables

First, we should find out the relations between two prefix to see whether they overlap with each other. Actually, each prefix represents a range, and there are three possible relationship between two ranges: they do not have overlapping part, they partially overlap with each other and one range is a part of another. For two prefixes, which are two special ranges, we have the following theorem:

Theorem 1. *For any two different prefixes, there are only two relationship between them: there is no overlap between them or one prefix is a subset of another.*

Proof. For any two different prefixes P_1 and P_2, we assume their represented ranges partial overlap with each other. So there must be a common value V in both ranges, and the value V matches both prefixes, then we can say P_1 and P_2 are two different prefixes of value V, so one prefix must be the prefix of another one and one prefix must be a subset of another, that means one represented range contains another one, not partial overlap, which is a contrary.

Therefore, for any two different prefixes, there are only two relationship between them: there is no overlap between them or one prefix is a subset of another. □

So based on Theorem 1, when separating the prefixes into two groups, we only need to check whether an IP prefix is a prefix of any other IP prefixes, and the algorithm is shown in Algorithm 1 and we assume the input IP prefixes have been sorted in prefix length decreasing order.

Algorithm 1 Separate IP Prefixes into Two Groups

```
Input : A Set of IP Prefixes P: p_1, p_2, ..., p_n
Output: Two Sets of IP Prefixes Q: q_1, q_2, ..., q_m and R: r_1, r_2, ..., r_t
for i = n to 2 do
    j = i − 1
    while j > 0 do
        if P_i is a prefix of P_j then
            add P_i to R
            BREAK
        end if
        j − −
    end while
    if j == 0 then
        add P_i to Q
    end if
end for
add P_1 to Q
```

And the update process is also simple, which contains inserting a prefix and removing a prefix shown in Algorithm 2 and Algorithm 3 respectively.

We can see that these algorithms are straightforward and easy to be implemented. However, the complexity of algorithm is $O(n^2)$, so it is not suitable for large IP lookup tables. Actually, we can build a trie architecture to separate a set of IP prefixes into two groups, and all the IP prefixes in leaf nodes belong to the first group and the remaining IP prefixes belong to the second group. The

Algorithm 2 Insert a new IP Prefix into Existing Two Groups

Input : The first group Q: $q_1, q_2, \ldots, q_m$, the second group R: $r_1, r_2, \ldots, r_t$ and inserting prefix p_{n+1}
Output: Updated the first group Q' and the second group R'
$i = 1$
while $i < m + 1$ **do**
  if p_{n+1} is a prefix of q_i **then**
    insert p_{n+1} into R according to prefix length decreasing order
    BREAK
  else
    if q_i is a prefix of p_{n+1} **then**
      insert q_i into R according to prefix length decreasing order
      replace q_j with p_{n+1}
      BREAK
    end if
  end if
  $i++$
end while
if $i == m + 1$ **then**
  insert p_{n+1} into Q
end if

Algorithm 3 Remove a new IP Prefix into Existing Two Groups

Input : The first group Q: $q_1, q_2, \ldots, q_m$, the second group R: $r_1, r_2, \ldots, r_t$ and removing prefix p_{n+1}
Output: Updated the first group Q' and the second group R'
$i = t$
while $i > 0$ **do**
  if p_{n+1} == r_i **then**
    remove r_i from R
    BREAK
  else
    if r_i is a prefix of p_{n+1} **then**
      $j = i$
    end if
  end if
  $i - -$
end while
if $i == 0$ **then**
  for i = 1 to m **do**
    if p_{n+1} == q_i **then**
      replace q_i with r_j
      remove r_j from R
      BREAK
    end if
  end for
end if

algorithm is shown in Algorithm 4 and the update process is shown in Algorithm 5 and Algorithm 6.

Algorithm 4 Separate IP Prefixes into Two Groups Using a Trie Structure

Input : A Set of IP Prefixes P: $p_1, p_2, \ldots, p_n$
Output: Two Sets of IP Prefixes Q: $q_1, q_2, \ldots, q_m$ and R: $r_1, r_2, \ldots, r_t$
for i = 1 to n **do**
  insert P_i to the trie T and record P_i in the corresponding node
end for
traverse the trie to insert all prefixes in non-leaf nodes into R
insert all the remaining prefixes into Q

Algorithm 5 Insert a Prefix Based on the Trie Structure

Input : The first group Q: $q_1, q_2, \ldots, q_m$, the second group R: $r_1, r_2, \ldots, r_t$, a corresponding trie T and update prefix p_{n+1} with property
Output: Updated trie T', the first group Q' and the second group R'
insert P_{n+1} to the trie T and record P_i in the corresponding node
if the inserted node becomes a non-leaf node **then**
  insert p_{n+1} into R according to prefix length decreasing order
else
  if the inserted node becomes a leaf node and another leaf node representing q_i becomes a non-leaf node **then**
    replace q_i with p_{n+1} and insert q_i into R according to prefix length decreasing order
  end if
else
  insert p_{n+1} into Q
end if

We get two groups of IP prefixes after the separation process, the first group of IP prefixes can be stored out of order, and do not need priority encoder. The second group of IP prefixes need to be stored in order in the TCAM entries, and each IP prefix in the second group must be an IP prefix of at lease one prefix in the first group, and the priority encoder is needed in the second group. It is obvious the first group has higher priority compared with the second group, so the second group needs to be checked only if there is no match in the first group. Based on these

Algorithm 6 Remove a Prefix Based on the Trie Structure

Input : The first group Q: $q_1, q_2, \ldots, q_m$, the second group R: $r_1, r_2, \ldots, r_t$, a corresponding trie T and update prefix p_{n+1} with property
Output: Updated trie T', the first group Q' and the second group R'
remove P_{n+1} from the trie T
if the removed node is a non-leaf node **then**
  remove p_{n+1} from R
else
  if the removed node is a leaf node and a non-leaf node representing r_j becomes a leaf node **then**
    replace p_{n+1} in Q with r_j and remove r_j from R
  end if
else
  remove p_{n+1} from Q
end if

properties, we can further optimize TCAM-based IP route lookup as discussed in the following subsections.

(2). Reduce Memory Access

In order to reduce the memory access latency, we can use the address of the TCAMs to represent the output port number when the IP prefixes can be stored out of order. In fact, the number of output ports is very limited compared with the number of IP prefixes need to be stored, but we need a large RAM to store these output ports because all the IP prefixes need to be stored in order, and every output port number needs to be stored in RAM for many times in different addresses, which is a big waste. Fortunately, we do not need to consider the order of IP prefixes in the first group, then we can eliminate the usage of RAM here. We store prefixes with the same output port together and distinguish different sets of prefixes from each other with a small number of range comparators. We use the comparison-enabled CAM (CCAM) which stores ranges in CAM like entries and compares ranges in parallel like TCAMs. And the matched address of CCAM represents the corresponding output port. The architecture is shown in Figure 2.10. Here

we assume there are 6 output ports, and we store the distinguish lines of TCAM address in the CCAM entries. Because the design of CCAM is easy and only a few CCAM entries are needed, we will not discuss the detail architecture of CCAM.

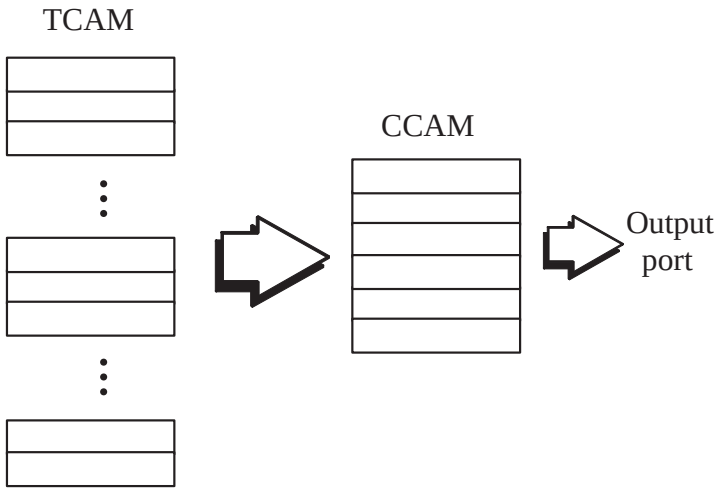


Figure 2.10: The proposed prefix storage using TCAMs without memory access in the Routing Tables

We use a small set of comparators instead of a large RAM, this not only eliminates memory access in the first group but also reduces hardware resource consumption.

(3). Process Route Lookup in Parallel

In order to enable the TCAMs to process prefix matching in parallel, the naive approach is to use multiple sets of TCAMs and each set of TCAMs contains a whole set of prefixes, but many more TCAM entries are required. We separate the first part into multiple blocks, and we separate it into four blocks in this section for example, and each block deals with different kinds of prefixes, so we separate the prefixes by the 23th bit and the 24th bit in the IP prefixes because of the following two reasons:

- The length of any IP prefix is between 8-bit and 32-bit, so we should choose common bits

from the highest 8-bit of IP prefixes.

- The IP prefixes have well balanced distribution on these two bits based on our simulation, that because higher bits in IP prefixes usually used to indicate different classes or groups.

So the first block stores IP prefixes with such two bits “00”, the second block stores IP prefixes with such two bits “01”, the third block stores IP prefixes with such two bits “10”, and the last block stores IP prefixes with such two bits “11”. When a packet comes in, these two-bit of the destination IP will be checked at first and forwarded to corresponding TCAM block, if a match can be found, the output port can be found by the matching address of the TCAM entry, otherwise, the destination IP address will be forwarded to the second group of TCAMs and the memory access is needed. Each TCAM block and the second part have their own waiting queue to store destination IP addresses waiting for check. The parallel structure is shown in Figure 2.11. The distributor distributes the incoming destination IP address into different TCAM blocks, and it also controls the input of sorted TCAM blocks based on the feedback from the four out of order TCAM blocks and the feedback signals are omitted in Figure 2.11.

In order to utilize the second part efficiently, we use two sets of the second IP prefix group to not only increase the overall throughput but also provide fast update in the second group. Because the IP prefixes in the second group are much fewer than in the first group, so we only need to add a small number of TCAM entries. As we discussed, it is pretty fast to update IP prefixes in the first group because of out of order storage, but the update process in the second group is slow even though it has much less IP prefixes because all of them should be stored in order. With two sets of the second IP prefix groups, one works as usually and another performs prefix update process when update is needed, then they switch their tasks until both of them have the latest IP prefixes.

The parallelism approach can not only increase the throughput of the route lookup, but also

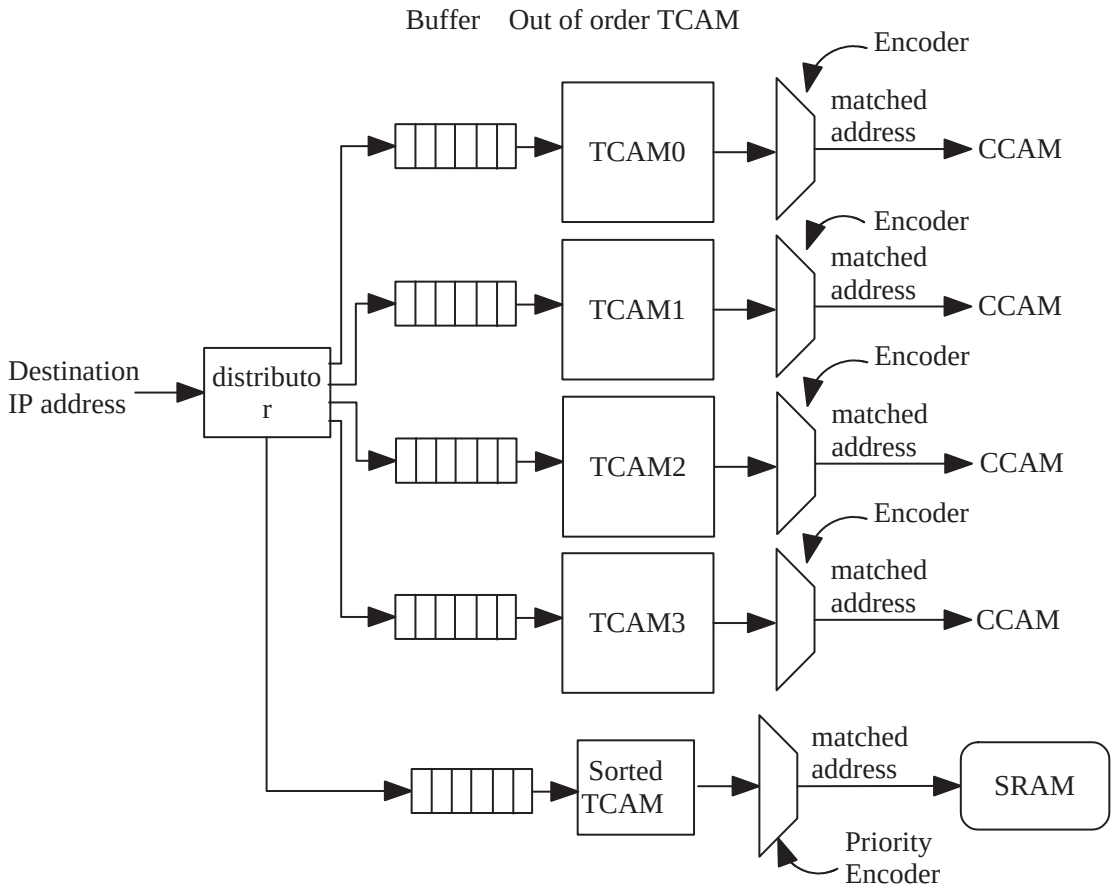


Figure 2.11: The proposed parallel processing architecture using TCAMs in the Routing Tables

reduce the power consumption because only a subset of TCAM entries are active when processing a lookup. And our approach can be easily pipelined to increase the throughput.

2.4.3 Evaluation

We first analyze the properties and trends of IP prefixes, and then simulate the benefits of three main ideas: separate prefixes, memory access removal and parallel processing units.

(1). Real-world IPv4 Prefixes

In this section, we analyze the properties of real-world prefixes. We collect all the routing

Table 2.5: Four Real-World Routing Tables

Name of Routing Tables	Date of Collection	Number of Prefixes
Oix-1998	09/18/1998	20,417 $(10^{4.31})$
Oix-2002	09/18/2002	61,659 $(10^{4.79})$
Oix-2006	09/18/2006	154,882 $(10^{5.19})$
Oix-2010	09/18/2010	16,218,100 $(10^{7.21})$

tables from the Route Views project [30]. In order to ensure that the characteristics of the distributions are not specific to some particular routers or time intervals, we inspect many routing tables and finally select four typical sets of routing table information from year 1998 to year 2010, where each set consists of all the data in a single typical day. The numbers of prefixes of the selected routing tables are shown in Table 2.5. and the comparison between them are plotted in Figure 2.12. (Note the logarithmic scale on the vertical axis).

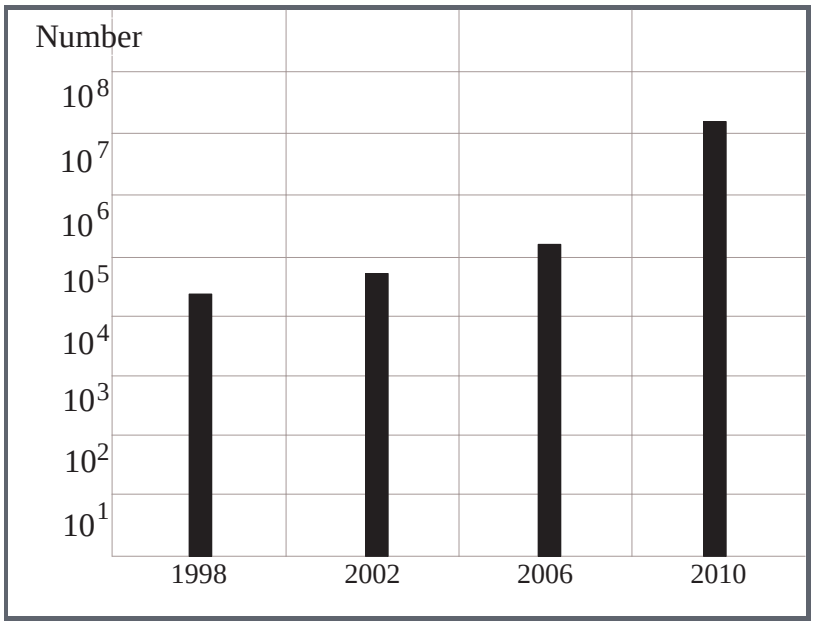


Figure 2.12: The Number of Prefixes in a Real Typical Network

We can see that the number of prefixes is growing exponentially, and is even growing faster

Table 2.6: Prefixes need to be stored sorted in different years

Routing Tables	Collection Date	Sorted Prefixes	Total Prefixes	Percentage (%)
Oix-2003	08/16/2003	8,157	81,654	9.99
Oix-2004	08/16/2004	7,871	94,891	8.29
Oix-2005	08/16/2005	7,077	105,947	6.68
Oix-2006	08/16/2006	8,255	126,180	6.54
Oix-2007	08/16/2007	10,229	149,700	6.83

Table 2.7: Prefixes need to be stored sorted in different time period in the same day

Routing Tables	Collection Time	Sorted Prefixes	Total Prefixes	Percentage (%)
Oix-2007-0	0am/08/16/2007	10,229	149,700	6.83
Oix-2007-2	2am/08/16/2007	10,198	149,703	6.83
Oix-2007-4	4am/08/16/2007	10,237	149,677	6.84
Oix-2007-6	6am/08/16/2007	10,258	149,862	6.84
Oix-2007-8	8am/08/16/2007	10,242	149,812	6.84

in recent four years. This indicates that the longest prefix matching will remain to be the bottleneck in high-speed routers.

(2). Benefits of separating prefixes

We simulate the percentages of prefixes in different groups, the first group eliminates the usage of priority encoder and the update speed increase.

(3). Overlap between IP Prefixes

In order to find out how many percentages of prefixes are needed to be stored in the sorted TCAM block, we evaluate five sets of prefixes in different years from 2003 to 2007 and five sets of prefixes in different time period in the same day, and the results are shown in Table 2.6 and Table 2.7.

We can see that there is usually less than 10 percent of the total prefixes need to be stored in order. That also means that the RAM usage can be reduced more than 90 percent, and smaller RAM also means faster memory access.

Table 2.8: Prefixes update in year 2002

Continuous time periods	Prefixes removed	Prefixes added	Update times per second
8am-12pm	976	1,048	0.28
10am-2pm	1,029	991	0.28
12pm-4pm	956	998	0.27
2pm-6pm	1,006	973	0.27
4pm-8pm	1,044	1,093	0.30
6pm-10pm	960	1,053	0.28

One of major disadvantages of TCAM-based approaches is that all the IP prefixes stored in TCAM must in the order of increasing prefix length because the longest prefix policy is implemented using a priority encoder. This simulation result means that most of the IP prefixes can be stored in the first group out of order in our approach, so this shows that our approach is reasonable.

(4). Eliminating use of priority encoder in the first group

With the increasing number of IP prefixes, the priority encoder consumes considerable hardware resource and the latency increases. Assume the number of IP prefixes in the first groups is n, which is also the bits of input of the priority encoder, the transistor consumption is O(n), and the latency is $O(\log n)$. Replaced with simple encoder, the hardware resource and latency can be reduced.

(5). Update Speed Improvement

In order to evaluate prefix update frequency, we collet data from the same router in different years: 2002, 2006 and 2010. Each data set consists prefixes in two hours, and we extract prefix update in two such continuous time periods.

The Table 2.8, Table 2.9 and Table 2.10 present the update frequency in different years.

The update frequencies in different years are shown in Figure 2.13.

We can see that the update frequency is steady in a short period, but increasing dramatically in recent years.

Table 2.9: Prefixes update in year 2006

Continuous time periods	Prefixes removed	Prefixes added	Update times per second
8am-12pm	2,017	1,970	0.55
10am-2pm	1,930	2,078	0.56
12pm-4pm	2,063	1,994	0.56
2pm-6pm	2,063	2,065	0.57
4pm-8pm	1,942	2,073	0.56
6pm-10pm	2,067	1,918	0.55

Table 2.10: Prefixes update in year 2010

Continuous time periods	Prefixes removed	Prefixes added	Update times per second
8am-12pm	10,101	10,306	2.83
10am-2pm	11,087	10,225	2.96
12pm-4pm	10,341	10,930	2.95
2pm-6pm	10,099	10,144	2.81
4pm-8pm	10,426	10,342	2.88
6pm-10pm	10,312	10,340	2.87

In this part, we use the IP prefixes selected from the last IP prefix group in 2010 in Figure 2.2 for our simulation. In average, for a group of 100,000 IP prefixes, each IP prefix update needs 23,482 TCAM entry writing processes in original approach, and in our approach it only needs 0.9 TCAM entry writing process in the first group and 2,196 TCAM entry writing processes in the second group. That's because when we need to update an IP prefix, at most one TCAM entry need to be written in the first group, and it may also affect TCAM entries in the second group. Actually, we can leave some gaps in the TCAMs to reduce the number of TCAM entry writing processes in the sorted TCAMs, but the update speed improvement ratio will not be changed by our approach. With two sets of the second IP prefix group, one works as usually and another performs prefix update process when update is needed, then they switch their tasks until both of them have the latest IP prefixes. By this way, packets go through the first IP prefix group will not be affected and packets go through the second IP prefix group are routed according to

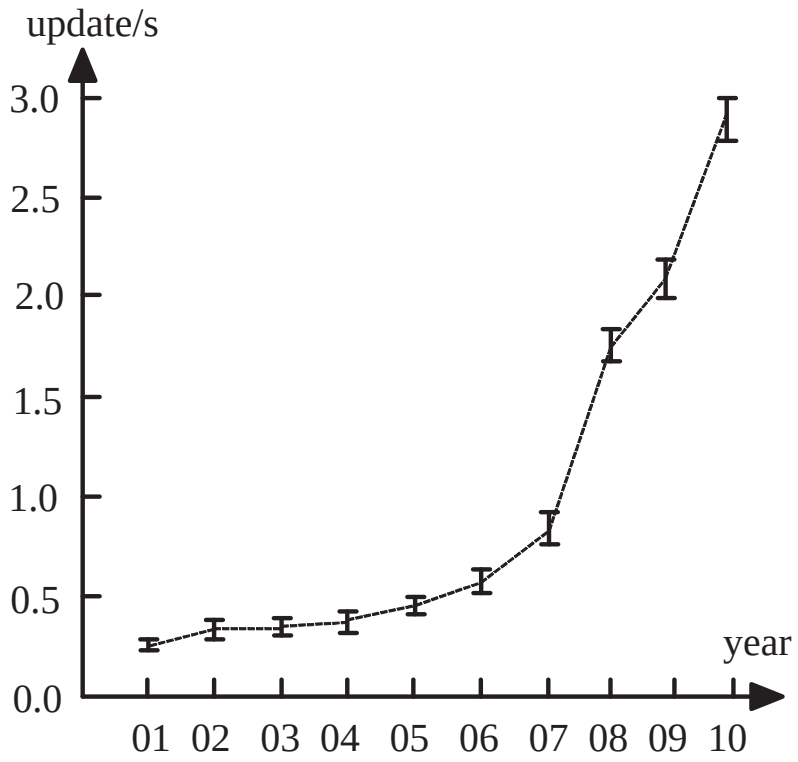


Figure 2.13: The differences of update frequencies in the past ten years

the previous route lookup table instead of being congested or dropped. That means with less than 10 percent more TCAM entries, we not only increase the overall throughput but also the update process dramatically.

(6). Benefits of memory access removal

We simulate the SRAM reduction, encoder removal and latency reduction on NetFPGA [34]. We use two bytes to store one output port information and the SRAM we used runs at 300 MHz with a Quad Data Rate (QDR II) SRAM interface [35]. We use the same sets of IP prefixes in Table 2.6 and the memory consumption reductions are shown in Table 2.11.

There are two data paths in our proposed architecture, including the first group of IP prefixes followed by OR logics and the second group of IP prefixes followed by priority encoder and

Table 2.11: SRAM consumptions reduction in different sets of IP prefixes

Routing Tables	Original Consumption (KB)	Our Approach (KB)	Reduction (%)
Oix-2003	163.308	16.314	90.01
Oix-2004	189.782	15.752	91.71
Oix-2005	211.894	14.154	93.32
Oix-2006	252.36	16.510	93.46
Oix-2007	299.4	20.458	93.17

Table 2.12: Hardware resource consumption of Priority Encoder(PE) and the latency of an IP prefix matching process based on different number of prefixes

Number of prefixes	PE reduction (%)	Latency reduction (%)
100	92.65	75.84
200	91.78	73.16
1000	90.84	71.94
2000	90.23	71.22

memory access. And our simulation shows that the second date path is the critical path and the hardware resource consumption of priority encoders and the latency of an IP prefix matching process based on different number of prefixes are shown in Table 2.12. We assume 10% of prefixes are in the second group and every prefix have the same hit rate.

We can see that the priority encoder can be reduced dramatically due to the small number of prefixes in the second group and the latency reduction is mainly because of the smaller priority encoder and fewer memory accesses.

(7). Benefits of parallel processing units

(i). Load Balancing between Different Blocks in the First Group

In order to evaluate the load balancing between different blocks in the first groups, we collect five different sets of prefixes in 2011. The percentages of the four blocks are shown in Figure 2.14.

From the Figure 2.14 we can see that our approach can provide good load balancing be-

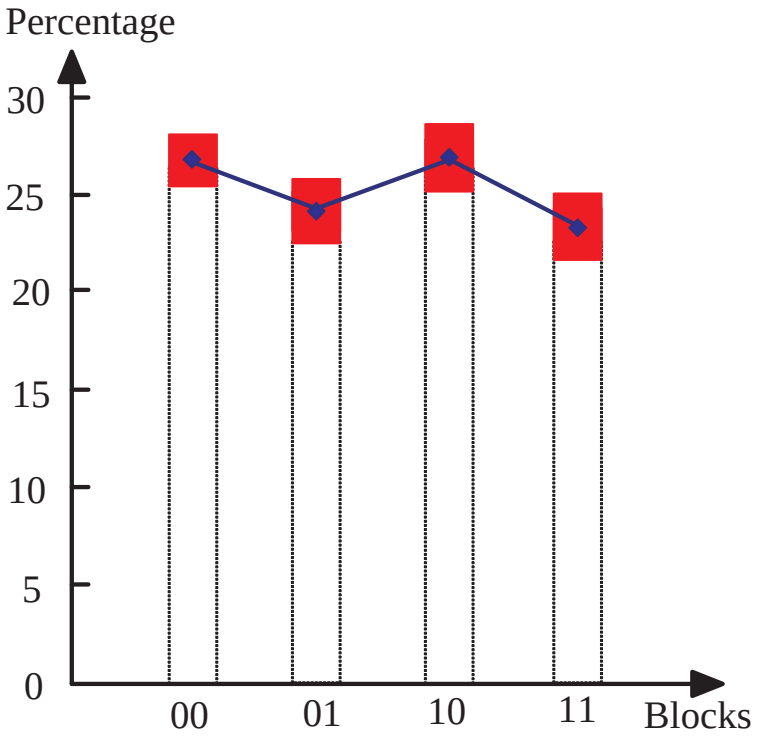


Figure 2.14: The percentage ranges of the four blocks

tween different blocks in the first group.

(ii). Throughput Improvement

We implement our approach as discussed above: the first IP prefix group is divided into four blocks and there are two sets of the second IP prefix group. We use selected IP prefix group in 2010 as shown in Figure 2.12 for our simulation. With the same clock cycle, original approach process one Destination IP address in a single clock cycle, and our approach process about 3.4 Destination IP addresses in a single clock cycle in average. So compared with original approach, our approach achieves 3.4 times as fast as before. Obviously, the maximum throughput is 4 times as fast as before, and the difference is caused by the different load balance in different blocks in the first IP prefix group and slower throughput in the second IP prefix group.

Our approach can be used together with existing algorithms that solve other problems of CAM-based IP route lookup, such as dealing with more routing table entries than TCAM entries. For example, our approach can be combined with software-based approaches to reduce the number of required CAM entries, because our approach is designed to be a drop-in replacement of TCAM-based longest prefix matching.

(8). Discussion

When the TCAM entries are run out, we should have an efficient back up strategy to deal with increasing number of IP prefixes in a system, but to the best of our knowledge, there are no such research on this issue and some products just use straightforward approaches. It can happen in some applications, for example, in network process based system, TCAM usually works as a coprocessor and have limited TCAM resources, so we should not rely on the TCAM to store all IP prefixes. Our algorithm provides a good way to use TCAM to cooperate with other approaches when the TCAM resource is no large enough. We can store the second group of prefixes in the TCAM, and we have shown that the percentage of these prefixes usually less than 10 percent, then we process the prefixes in the first group in other approaches, such as Trie-based approaches. The benefits of this approach is that the Longest Prefix Matching (LPM) problem becomes Prefix Matching (PM) prefix, which means for each destination IP address there is at most one path from the root node to the leaf node, so this reduces the task of the main processor, and the coprocessor works only when there is no result found in the main processor. We hope this first step can attract more researchers' attention on how to design efficient algorithm to solve limited TCAM resource problem.

Our approach can be easily extended to be used in IPv6 applications because the advantages of our approach are not affected by the new properties of IPv6 compared with IPv4.

The TCAMs support "don't care" state in any bit in a TCAM entry, however, some ap-

plications only use TCAMs to store prefixes, such as longest prefix matching problem. In such applications, the "don't care" state can only appear in continuous "don't care" state sequence from the least significant bit of a TCAM entry, and that will cause hardware waste. And a simplified TCAM can be designed to only support prefix storage and we call this kind of CAM "PCAM", where "P" represents $prefix$. The logic of original 8-bit TCAM and the proposed architecture of 8-bit PCAM is shown in Figure 2.15 and Figure 2.16 respectively, and a 8-bit prefix "$1110** **$" is stored in both of them. In the TCAM, each pair of value and mask represent a single bit value in Figure 2.15, and value and mask are stored in SRAM separately, but we can remove the storage of mask because we only need to store the boundary between binary bits and "don't care" bits. So we append an extra value storage after the least significant bit and store a value "1" in the value storage to separate binary bits and "don't care" bits in Figure 2.16. So we can figure out the prefix when finding out the least significant value "1".

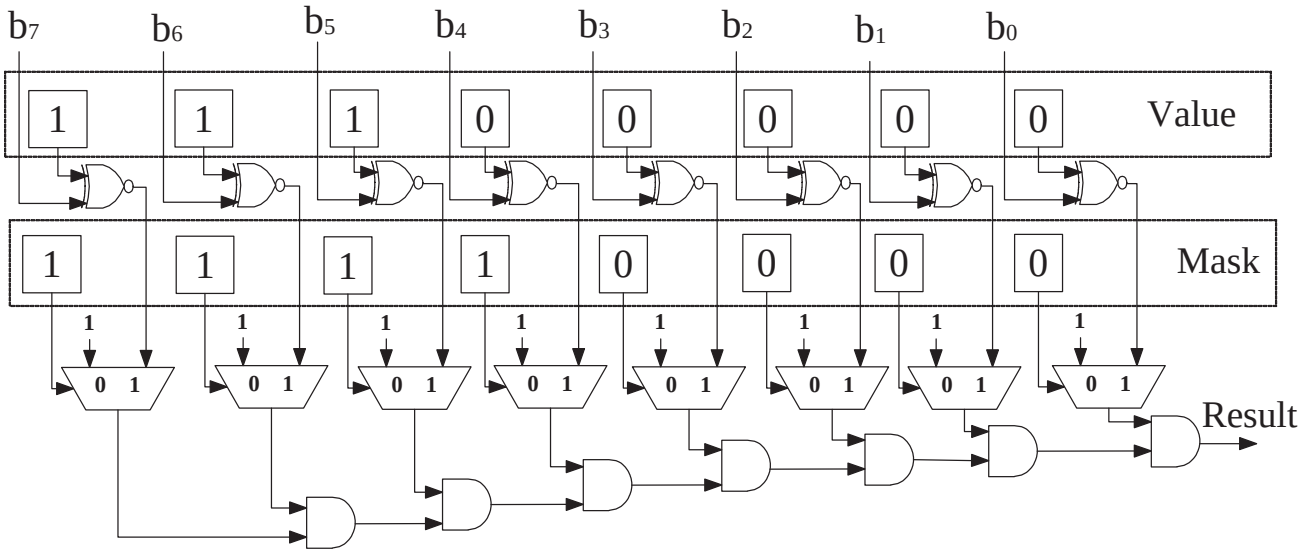


Figure 2.15: The logic of 8-bit TCAM storage

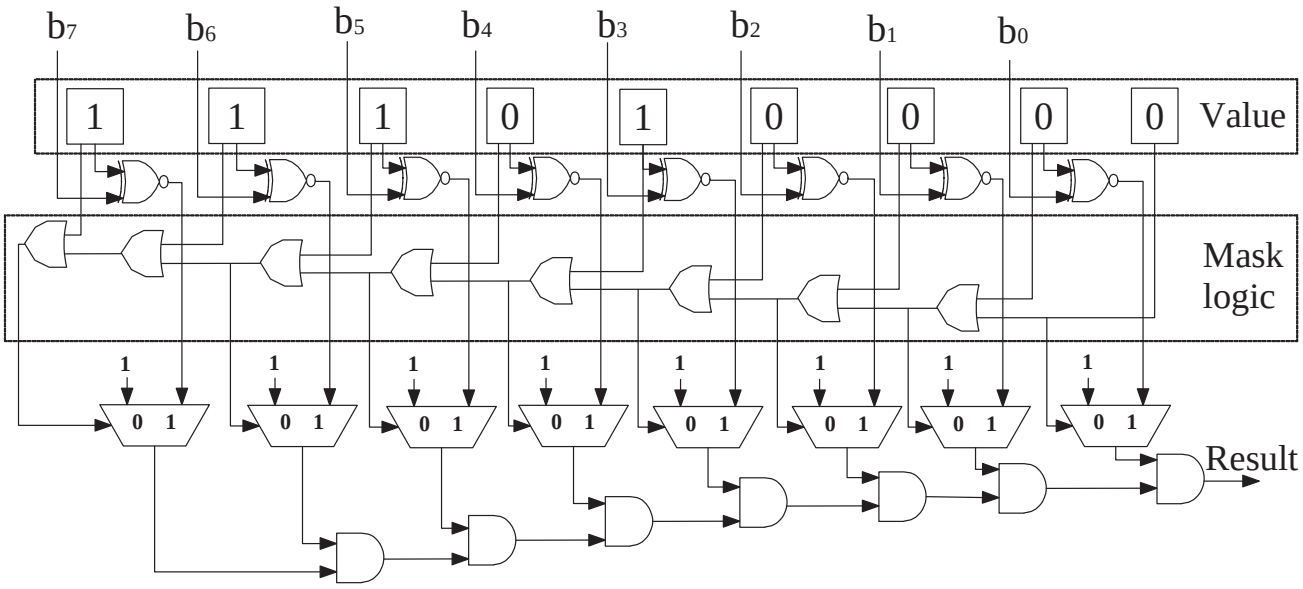


Figure 2.16: The architecture of proposed PCAM supporting prefix storage

2.4.4 Summary

In this section, we present a novel approach to IP route lookup using TCAMs to save memory usage, increase update speed and improve the throughput of longest prefix matching process. We discussed three important issues still have not been well studied in TCAM-based IP route lookup including large RAM usage and long memory accesses, slow update process of routing table and unscalable problem. Based on the observation that all of them result from the sorted storage in TCAM entries, we store IP prefixes can be stored out of order in the first group and the remaining ones in the second group, and deal with them separately. The simulation results show that our approach solve these problems efficiently: the usage of RAM has been reduced, the throughput has been increased and the update process speed also has been increased significantly compared with the traditional ways. Furthermore, the clock speed can also be increased because the memory access latency is reduced and the priority encoders are simplified. Our main contribution of this

section is to present an approach to solve multiple major problems in TCAM-based applications, not only try to solve one problem but usually worsen the others.

CHAPTER three

PACKET CLASSIFICATION

3.1 Introduction

There are a number of network services that require packet classification, such as policy-based routing, firewalls, provision of differentiated qualities of service, and traffic billing. In each case, it is necessary to determine which flow an arriving packet belongs to so as to determine, for example, where to forward it, whether to forward or filter it, what class of service it should receive, or how much should be charged for transporting it. As packet classification has been widely deployed on the Internet, demand for efficient packet classification grows. The function of a packet classification system is to map each packet to a decision according to a sequence of rules, which is called a packet classifier. The rules specified in a packet classifier may or may not be mutually exclusive; two rules may overlap in a packet classifier. When it happens with no explicit priorities specified, we follow the convention that a rule closer to the top of the list takes priority. Table 3.1 shows a simple packet classifier of four rules.

Perhaps the most popular method for high-speed packet classification in practice is to use Ternary Content Addressable Memory (TCAM) [36]. A TCAM is a memory chip where each entry

Table 3.1: A Simple Header Rule Set

Rule	Type	Src IP	Dst IP	Src Port	Dst Port	Decision
r_1	TCP	*	192.168.0.0/16	$<$1024	*	accept
r_2	TCP	*	192.168.14.1	*	139	discard
r_3	UDP	192.168.0.0/16	10.163.38.0/8	*	700:900	accept
r_4	TCP	*	*	*	*	discard

can store a packet classification rule in ternary form. It stores data patterns in the form of (value, bit mask) pairs. A query key can be simultaneously compared against all the patterns stored in a TCAM. A key q is said to match a stored pattern (v, m) if $q \quad \& \quad m = v \quad \& \quad m$, where "&" is the bit-wise logical AND operator. Given a packet, the TCAM hardware can compare the packet with all stored rules in parallel and then return the decision of the first rule that the packet matches through a priority encoder. Thus, it takes $O(1)$ time to find the decision for any given packet. Because of their high speed, TCAMs are widely used in prefix matching applications [26] and TCAMs have become the industrial standard for high speed packet classification [37]. The architecture of a TCAM used in the packet classification is shown in Figure 3.1. The input data compares with TCAM entries in parallel, and the priority encoder selects the matched entry with the highest priority, at last, memory access is performed to fetch corresponding result. Note that the priority encoder and memory access process may consumes much hardware resources and increase the overlay latency, and we will discuss these problems and solve them based on our approach.

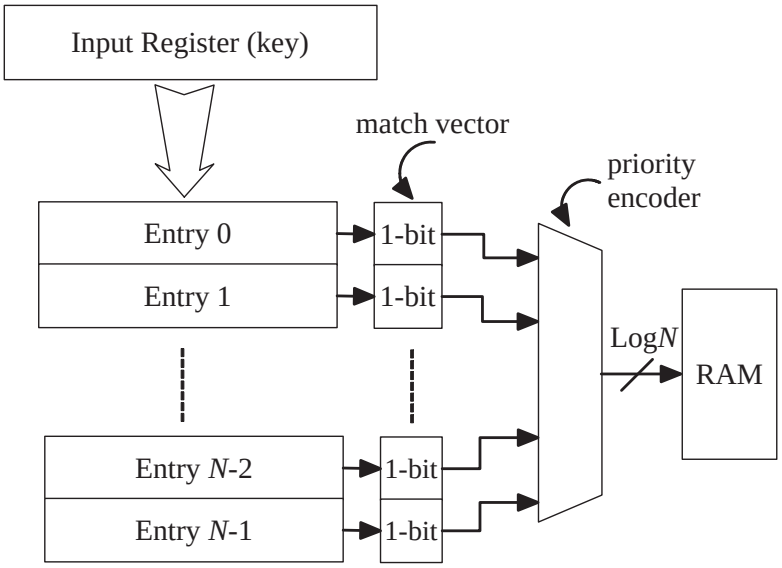


Figure 3.1: TCAMs used in the packet classification

A key (Protocol, Source IP address, Destination IP address, Source Port, and Destination Port) is stored in the input register and the rules are stored in the TCAM entries. The key compares with all the entries in parallel and the results are stored in the match vector, where 1's represent that the corresponding entries match the key, and the priority encoder chooses the match with the highest priority. At last, the output signal is used to find the corresponding action.

Despite their high speed, TCAMs have two major drawbacks when used in packet classifiers. First, they consume a large amount of power and have high hardware cost. Thus, their capacity in packet classifiers is often limited. Second, they are inefficient when applied to packet classifiers with port number ranges, because TCAMs can only store rules in ternary form, which means that port numbers need to be converted to one or more prefixes before being stored in TCAMs. This may lead to a significant increase in the number of TCAM entries needed to encode a rule. For example, 30 prefixes are needed to represent a single range [1, 65534], and 20 prefixes are needed to represent [1, 2046]. Overall, $30 \times 20 = 600$ TCAM entries are required to represent a single rule with these two ranges. We observe that packet classifiers typically have at most one port range in each rule, and rules specifying two port ranges are very rare. However, a small number of such rules can consume most of the TCAM entries and the number of such rules is increasing. Therefore, minimizing the number of required TCAM entries is crucial.

In this chapter, we first propose a tree-based algorithm to minimize the number of TCAM entries used by a packet classifier. An effective way to reduce the number of TCAM entries is to remove redundant rules and combine overlaying rules, yielding an equivalent set of rules without affecting the classification results. Based on the new non-overlapping rule set, we can remove a set of rules with the same decision from TCAMs to reduce the TCAM consumption. Furthermore, we can store rules in TCAMs out of order, then we can remove the priority encoder and memory access process. Our algorithm is based on such an equivalent transformation. It can reduce the number of

TCAM entries significantly while preserving the behavior of packet classifiers. A key advantage of reducing TCAM entries through this algorithm is that the algorithm can be easily deployed without any modification of existing packet classification systems. Our experiments show that we achieve a total reduction of 85.4% in the number of TCAM entries by removing overlaps between rules and removing the most rules with the same decision.

Our approach removes all redundancy in the original rule set to make sure each packet matches and only matches a single new rule. Based on the new non-redundant rule set, we further propose two range extension algorithms, one-directional range extension and bidirectional range extension, which require less TCAM entries to store them. Our algorithm is based on such an equivalent transformation. It can reduce the number of TCAM entries significantly while preserving the behavior of packet classifiers. A key advantage of reducing TCAM entries through this algorithm is that the algorithm can be easily deployed without any modification of existing packet classification systems. Our experiments show that we achieve a total reduction of 79.28 and 84.27% in the number of TCAM entries by removing redundant rules, combining overlaying rules and employing the range extension scheme.

Priority encoders are commonly employed with TCAMs together to detect the matching result with the highest priority among all matched entries [38–40]. The priority encoders usually consume considerable hardware area especially with large number of TCAM entries. So we also propose the Leading-zero counter to replace the priority encoder and the simulation shows significant area reduction introduced by our approach.

Based on the new non-redundant rule set, we also propose a fast TCAM update scheme which enable out of order storage in the TCAM and reduce the TCAM entries usage. Our experiments show that only 15.27 TCAM entries needed to be updated by adding a new rule in average, and our algorithm consumes approximately 29% and 36% TCAM entries update of original method

and existing compression method when removing a rule.

At last, we propose Comparator Content Addressable Memories (CCAMs), which can store ranges in CCAM entries besides ternary state numbers. In our algorithm, the hardware can be configured to store all kinds of port number ranges. If a rule contains any two-direction ranges, only two CCAM entries are needed to store it, otherwise, each rule can be stored in a single CCAM entry. Our simulations show that CCAMs consume about 27.6% entries compared with original TCAMs. Considering the transistor consumption, our approach saved about 66.4% transistors. Furthermore, the update process of CCAMs can be as fast as that of TCAMs usage without any compressions.

Our contributions are as follows:

1. We propose an overlapping removal algorithm to reduce the number of TCAM entries consumed and provide a good property: exact one matched TCAM entry for every searching, and we call that one-match property.

2. Based on the one-match property, we can remove a set of rules with the same decision from TCAM to further reduce the TCAM entries consumption. Of course, we remove the same decision rules consumes the most TCAM entries.

3. When there are only two decisions “accept” and “discard” in the rule set, based on the one-match property and the contribution (2), there is no priority encoder and memory access needed, because all the rules stored in TCAM has the same decision, and we know the decision by knowing whether there is a match in the TCAM entries.

4. When there are more than two decisions in the rule set, Based on the one-match property, we can store rules in TCAM entries out of order, so we store rules with the same decision

together and use Leading-zero counter to replace the priority encoder and use comparators to replace memory access process.

5. TCAM-based rule updating process usually takes long time and degrades the performance of packet classification because the packets must be buffered during the update. However,our approach provides fast TCAM update due to out of order storage in the TCAM.

The remainder of the chapter is organized as follows. Section 3.2 presents previous work related to this chapter. Section 3.3 proposes our tree-based algorithm and Section 3.4 presents range extension algorithm to reduce the number of TCAM entries. Section 3.5 and Section 3.6 present the fast rule update scheme and CCAM respectively. Finally, we conclude in Section 3.3.5.

3.2 Related Work

Previous work exploring solutions to deal with the range expansion problem falls into two major categories: hardware-based solutions, which require changing TCAM hardware circuits, range reencoding solutions, which reencode original ranges into other formats, and classifier compression solutions, which do not require such changes. Next, we review previous work in these three categories.

Hardware-based solutions The basic idea of hardware-based solutions is to modify TCAM circuits and architecture [37, 41–47]. For example, van Lunteren et al. proposed a method of adding comparators at each entry to better accommodate range matching in packet classifiers [43]. While this allows to use TCAMs more efficiently, any solution from this research line has some drawbacks such as the cost of hardware modification.

Range Reencoding solutions Many range reencoding compression solutions have been proposed [37, 48–52]. Their basic idea is to reencode field's value into more simple values. But this kind of approaches suffer from large RAM consumption and complex preprocessing of ranges. Furthermore, every income packet is need to be preprocessed before performing match operation. Therefore, we do not focus on this kind of solutions.

Classifier Compression solutions Many classifier compression solutions have been proposed [4, 45, 48, 52–61]. Their basic idea is to preprocess ranges that appear in a packet classifier or convert a given packet classifier to another semantically-equivalent packet classifier that requires fewer TCAM entries, and then store the new rule set in a TCAM. Hence, the TCAM circuits need not be modified to implement range storage, but the preprocessing is required. Although these methods can efficiently reduce the redundancy in the rule set, they may still miss some redundancy. Our work falls into this category and further optimizes existing approaches. Classifier compression solutions are more likely to be adopted by networking vendors and ISPs because they do not require changing TCAM hardware or existing packet classification systems and the income packets are not need to be preprocessed.

There are some other software-based packet classification, such as tree-based approaches [62–65] and hash-based approaches [66–68]. They usually require multiple memory access for each packet and we will not discuss them in detail because they are not our focus in this chapter.

3.3 Tree-Based Minimization of TCAM Entries for Packet Classification

Packet classification is a fundamental task for network devices such as edge routers, firewalls, and intrusion detection systems. Currently, most vendors use Ternary Content Addressable Memories (TCAMs) to achieve high-performance packet classification. TCAMs use parallel hardware to

check all rules simultaneously. Despite their high speed, TCAMs have a fundamental drawback in dealing with ranges efficiently. Many packet classification rules contain range specifications, each of which needs to be translated into multiple prefixes to store in TCAMs. Such translation may result in an explosive increase in the number of required TCAM entries. Furthermore, the priority encoder and memory access after TCAM match consumes too much hardware resources and increase the overall latency and we call them postprocess of TCAM matching. In this section, we propose an efficient TCAM-based packet classification algorithm, which consists of two parts: overlapping removal in a rule set using a tree representation of rules, and the postprocess optimization. The proposed algorithm first removes redundant rules and combines overlaying rules to build an equivalent, smaller rule set for a given packet classifier, and then removes priority encoder and memory access process based on the properties of non-overlapping rule set. Our experiments show a reduction of 85.4% in the number of TCAM entries based on Minimal Range Tree (MRT). Besides, we propose the leading-zero counter instead of priority encoder circuits and use comparators instead of RAM to reduce the hardware area and overall latency. It can also be used as a preprocessor, in tandem with other methods, to achieve further performance improvement. We first discuss the hyperrectangular partitioning problem in the packet classification.

3.3.1 Hyperrectangular Partitioning Problem

In d-dimensional packet classification, a packet is represented as a d-dimensional al vector, $p = (f_1, f_2, \ldots, f_d)$, where f_i is the value of the ith field of the packet. A d-dimensional packet classifier C is defined as a sequence of *condition-action* pairs:

$$C = ((c_1, a_1), (c_2, a_2), \ldots, (c_n, a_n)) \tag{3.1}$$

where c_i is a subset of the d-dimensional space and a_i is an action taken when a packet p is in c_i. Because conditions are usually given as prefixes and ranges, c_i represents a d-dimensional hyperrectangle. In an IPv4 packet, those d dimensions are usually source IP address, destination IP address, source port, destination port, and protocol type, of which the lengths are 32, 32, 16, 16, and 8 bits, respectively. If $\bigcup_{1 \leq i \leq n} c_i$ is the d-dimensional space itself, the classifier C is *complete*. Note that virtually all packet classifiers are complete because they have a "default" condition that covers the entire space.

TCAMs are usually used to store these rules to accelerate the searching process. In order to store c_i into a TCAM entry, every c_i must be represented as an exact binary value or a binary value with wildcard bits. For example, an IP address prefix 127.0.0.1/16 is converted to `0111111100000000****************`, where $\star$ is a wildcard bit. However, some fields such as source and destination port numbers are represented as integer ranges rather than exact values or prefix values. Thus, c_i with fields represented as integer ranges may need to be converted into more than one prefixes, which is called "range expansion." The process of range expansion consists of two parts. In the first part, each field of the condition is expanded independently. For example, if a condition for a 3-bit field is [1, 6], the corresponding minimum set of prefixes are `001`, `01`$\star$, `10`$\star$, and `110`. The worst-case range expansion of a w-bit integer range yields $2w - 2$ prefixes [69]. Then, the second part is to compute the cross-product of obtained prefix sets for all fields, resulting in an exponential increase of number of TCAM entries needed to store a single condition.

In order to mitigate this problem, we propose our approach based on removing overlaps between different rules. Two rules in a packet classifier may overlap, which means that one packet may match two or more rules. Besides, two rules in a packet classifier may conflict with each other. In other words, two overlapping rules may have different decisions. Packet classifiers typically

Table 3.2: Number of prefixes needed by a range

Range length	Possible numbers of prefixes required to represent the range	Frequency ratio of required prefix numbers	Average number of reqruied prefixes
1	1	1	1
2	1, 2	1:1	1.5
3	2	1	2
4	1, 2, 3	1:1:2	2.25
5	2, 3	1:1	2.5
6	2, 3, 4	2:1:1	2.75
7	3	1	3
8	1, 2, 3, 4	1:1:2:4	3.125
9	2, 3, 4	1:1:2	3.25
10	2, 3, 4, 5	2:3:1:2	3.375

resolve conflicts by employing the first match, which has higher priority. For firewalls, typical decisions include "accept," "discard," "accept with logging," and "discard with logging."

One motivation of removing overlaps is based on the observation that smaller range is likely to occupy fewer TCAM entries. For example, Table 3.2 shows the number of prefixes needed by ranges with different lengths varying from 1 to 10. We can see in the second column that ranges with the same length may need different numbers of prefixes. We compute the frequency of each number of prefixes in the third column, assuming that ranges are uniformly distributed. Using the frequency as weight, we obtain the average number of prefixes for each range length in the last column. The average number of prefixes increases as the range length increases. Note that the sum of two average lengths is larger than the average length of the sum of their range lengths. For example, the average number of prefixes for a range of length 3 is 2 and that for a range of length 4 is 2.25. However, if we combine two ranges into one of length 7, the corresponding average number of prefixes is 3. Thus, the compression ratio becomes $(2 + 2.25 - 3)/(2 + 2.25) = 29\%$. For longer ranges, the average number of prefixes grows logarithmically.

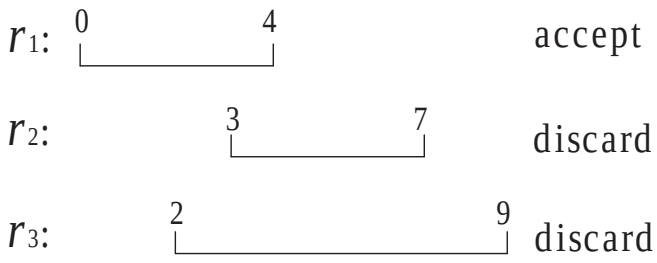


Figure 3.2: A simple packet classifier

One motivation of removing overlaps is based on the fact that overlaps between rules require the usage of priority encoder, and large number of rules will result in a very large priority encoder, which not only consumes a lot of hardware resource but also increase the processing latency significantly. So we can remove the large priority encoder by removing overlaps between rules.

We define redundancy in rules as follows:

- Redundancy in a rule set: A rule set contains redundancy if any packet matches more than one rules in the rule set.

For example, a simple packet classifier with three rules is shown in Figure 3.2. The rules r_2 and r_3 contain redundancy ranges of [3, 4] and [2, 7] respectively.

Compared with redundancy removal approach in [69], our approach further reduces the overlap redundancy and combine continuous ranges with same decision, which not only further reduce the number of TCAM entries consumption but also provide basic non-gap non-overlap rules for further optimization algorithms and we will discuss the details in the following sections.

So we define the hyperrectangular partitioning problem as follows:

- Given a $d-$dimensional packet classifier C, how to cut the whole $d-$dimensional space into a set of $d-$dimensional cubes to meet the following requirements:

▶ Each $d-$dimensional cube has a single decision.

▶ This set of $d-$dimensional cubes and packet classifier C are semantically-equivalent.

▶ Among all the possible sets of $d-$dimensional cubes, this set of $d-$dimensional cubes require minimum number of TCAM entries to store them when each dimension of a cube is presented by a prefix.

3.3.2 Proposed Algorithms

In order to reduce the number of TCAM entries consumption, we propose the minimal range tree to remove all redundancy in the original rules. In the new rules we get, there is no overlap or gap in all the five fields, which ensure any income packet matches and only matches one new rule. Based on the new non-overlapping rule set, we remove a set of rules with the same decision from TCAMs to reduce the TCAM consumption. Furthermore, we store rules in TCAMs out of order, then we remove the priority encoder and memory access process. We first consider rule set with two decisions in our approach and then extend it to rules with more decisions. In the latter situation, we propose the Leading-zero counter and comparator approach to replace the priority encoder and memory access process respectively to reduce consumed hardware area and the overall latency. In order to better describe our approach, we first consider rules only contain two decisions "accept" and "discard", and then describe rules with more than two decisions.

(1). Minimal Range Tree

Our goal is reducing redundancy in a given packet classifier, including redundant rules and overlapping parts. To achieve the goal, we build a minimal range tree as follows.

A range tree T for a packet classifier f: $(r_1, r_2, \ldots, r_n)$ over fields $F_1, \ldots, F_d$ is a tree that has the following properties:

- The height of the tree is equal to the number of fields in the packet classifier.
- Edges of each depth of the tree store the ranges of the corresponding field. All edges in the same depth cover the whole range of the field, and there is no overlap between any pair of them.
- A directed path from a leaf node to the root is called a *decision path.* For a given packet, the tree has exactly one matched decision path.
- Each leaf node is labeled with decision that is associated with the corresponding decision path.

Figure 3.4(a) shows a range tree for the simple packet classifier in Figure 3.3. In this example, we assume every packet has only two fields, F_1 and F_2, and the domain of each field is [0, 9].

$$
\begin{array}{lll}
r_1: & F_1 \in [0,4] \wedge F_2 \in [0,9] & \rightarrow \text{accept} \\
r_2: & F_1 \in [0,4] \wedge F_2 \in [4,9] & \rightarrow \text{accept} \\
r_3: & F_1 \in [5,9] \wedge F_2 \in [7,9] & \rightarrow \text{accept} \\
r_4: & F_1 \in [5,9] \wedge F_2 \in [0,2] & \rightarrow \text{discard} \\
r_5: & F_1 \in [0,9] \wedge F_2 \in [0,9] & \rightarrow \text{discard}
\end{array}
$$

Figure 3.3: A simple packet classifier with five rules

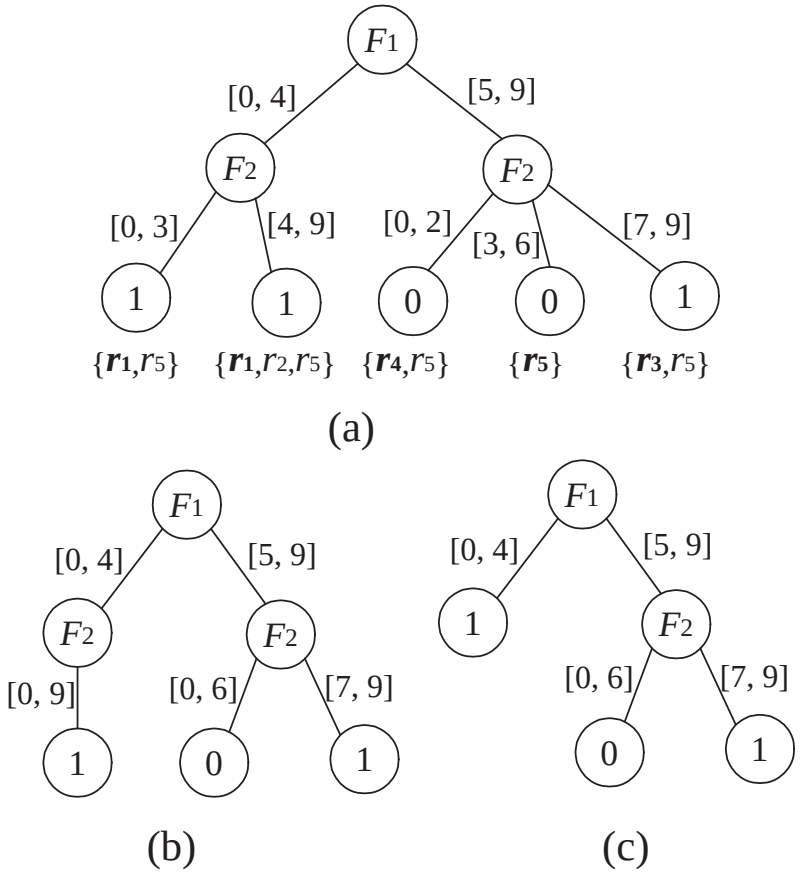


Figure 3.4: Constructing a minimal range tree for the packet classifier in Figure 3.3

We use number 1 as a shorthand for "accept" and number 0 as a shorthand for "discard" in labeling leaf nodes in Figure 3.4. In Figure 3.4, each edge represents a range in the corresponding field. We first build a tree as in Figure 3.4(a) according to the packet classifier in Figure 3.3, and the rules below each leaf node represent which rules satisfy the decision path. They are not a part of the tree. Then we combine two neighboring leaf nodes if they have the same decision and share the same parent node. The result is shown in Figure 3.4(b). Last, we move a leaf node to its parent node if this leaf node is the only child node as shown in Figure 3.4(c). As a result, we get the

minimal range tree in Figure 3.4(c). It corresponds to three new rules in Figure 3.5, which are equivalent to the rules in Figure 3.3.

$$
\begin{array}{lcl}
r'_1: & F_1 \in [0,4] & \rightarrow \text{accept} \\
r'_2: & F_1 \in [5,9] \wedge F_2 \in [0,6] & \rightarrow \text{discard} \\
r'_3: & F_1 \in [5,9] \wedge F_2 \in [7,9] & \rightarrow \text{accept}
\end{array}
$$

Figure 3.5: Three new rules according to Figure 3.4(c)

In this example, we remove all redundancies in the leaf level. However, there may be redundancies among internal nodes. Therefore, we also need to remove such redundancies. After removing redundancies in leaf nodes, some of their parents may become leaf nodes. We apply the same algorithm recursively from leaf nodes to the root to check the redundancy for them.

There is another type of redundancy that is not removed by the previous algorithm. After applying the previous algorithm, it is obvious that if two leaf nodes still have the same decision and have the same field range, then these two nodes must have different parent nodes. If these two nodes have the same grandparent node, it implies that there must be redundancy in their parent nodes, which we call parent redundancy. However, we cannot reduce this kind of redundancy through the algorithm we discussed above; we need to search for parent redundancy in a separate step. Figure 3.6 shows a simple example of removing parent redundancy.

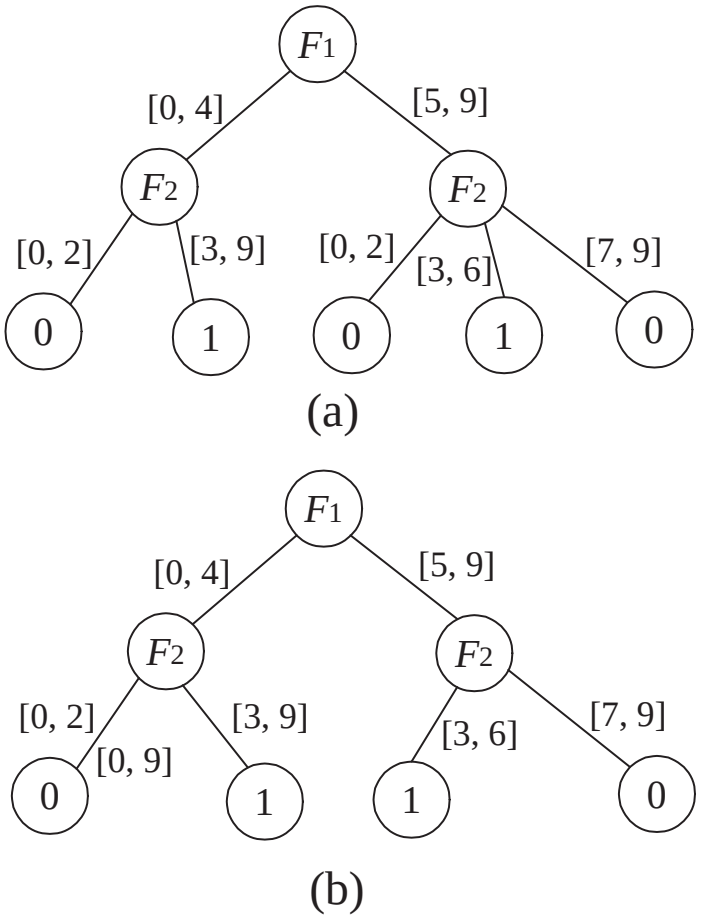


Figure 3.6: An example of removing parent redundancy

In Figure 3.6(a), the first leaf node and the third leaf node have the same decision (0), the same field range ([0, 2]), and the same grandparent node (F_1). Therefore, there is redundancy between their parent nodes (two F_2). In this case, we delete the second leaf node and combine their parent nodes' field ranges ([0, 4] and [5, 9]) so that it becomes the second field range of the first leaf node as shown in Figure 3.6(b). This new range overrides its parent's field range [0, 4], yielding the following decision path:

$$F_2 \in [0, 2] \wedge F_1 \in [0, 9] \rightarrow \text{discard}. \tag{3.2}$$

From the discussion above, in order to reduce TCAM entries, we have the following three strategies:

1. Reduce redundant rules,
2. Reduce overlapping between rules, and
3. Combine rules even there is no overlapping.

These three strategies are based on the following observations:

- The number of prefixes increases as the range length increases.
- For two ranges, [a, b] and [c, d], are overlapping with each other or have no gap between them ($b + 1 = c$ or $d + 1 = a$), the number of prefixes needed by them is larger than the number of prefixes needed by the combined range [mina, c, maxb, d].

We have the following theorem and corollary:

Theorem 2. *Given a range $\mathbb{R}_{12}$, when we randomly cut it into two ranges $\mathbb{R}_1$ and $\mathbb{R}_2$, the minimum number of prefixes needed to represent both $\mathbb{R}_1$ and $\mathbb{R}_2$ is larger than or equal to the minimum number of prefixes needed to represent $\mathbb{R}_{12}$.*

Proof. Given a range $\mathbb{R}_{12}$, when we randomly cut it into two ranges $\mathbb{R}_1$ and $\mathbb{R}_2$, assume the two minimum sets of prefixes needed to present $\mathbb{R}_1$ and $\mathbb{R}_2$ are $p_1 \ldots p_m$ and $q_1 \ldots q_n$ respectively. For the combined range $\mathbb{R}_{12}$, we can also use the same prefixes $p_1 \ldots p_m$ and $q_1 \ldots q_n$ to represent it, or we can combine multiple prefixes to reduce the number of prefixes.

Therefore, given a range $\mathbb{R}_{12}$, when we randomly cut it into two ranges $\mathbb{R}_1$ and $\mathbb{R}_2$, the minimum number of prefixes needed to represent both $\mathbb{R}_1$ and $\mathbb{R}_2$ is larger than or equal to the minimum number of prefixes needed to represent $\mathbb{R}_{12}$. □

Corollary 1. *For any two ranges and their relation belongs to one of the following three categories: (1). One range is a subset of another. (2). A part of a range is a subset of another. (3). There is no overlap and no gap between the two ranges and we can call them continuous ranges. Then we can combine these two ranges to reduce the number of prefixes to represent them or the number remains the same in the worst case.*

Proof. Given two ranges $\mathbb{R}_1$, $\mathbb{R}_2$ and combined range $\mathbb{R}_{12}$, assume the minimum sets of prefixes needed to present $\mathbb{R}_1$, $\mathbb{R}_2$ and $\mathbb{R}_{12}$ are $p_1 \ldots p_m$, $q_1 \ldots q_n$ and $c_1 \ldots c_w$ respectively, we approve it in three situations separately:

• If $\mathbb{R}_1$ is a subset of $\mathbb{R}_2$, we can use $q_1 \ldots q_n$ to represent $\mathbb{R}_{12}$, then $m + n > w$. If R_2 is a subset of R_1, we have the same result.

• If a part of $\mathbb{R}_1$ is a subset of $\mathbb{R}_2$, that also means a part of $\mathbb{R}_2$ is a subset of $\mathbb{R}_1$, we can use $p_1 \ldots p_m$ and $q_1 \ldots q_n$ to represent combined range $\mathbb{R}_{12}$, and we can further combine some prefixes to reduce the number of prefixes needed to present $\mathbb{R}_{12}$, then we get that $m + n \geq w$.

• If there is no overlap and no gap between the two range $\mathbb{R}_1$ and $\mathbb{R}_2$, based on Theorem 2, then we get that $m + n \geq w$.

Therefore, for any two ranges and their relation belongs to one of the following three categories: (1). One range is a subset of another. (2). A part of a range is a subset of another. (3). There is no overlap and no gap between the two ranges and we can call them continuous ranges. Then we can combine these two ranges to reduce the number of prefixes to represent them or the number remains the same in the worst case. □

Table 3.3: Analysis of Snort Rules

Field	Number of distinct values	Most frequently used values	Number of rules
Protocol type	4	TCP	7550
		UDP	490
		ICMP	135
		IP	39
Destination port	314	445	1574
		$HTTP_PORTS	1568
		any	1564
		139	1464
		$ORACLE_PORTS	291
Source port	196	any	7056
		$HTTP_PORTS	737
		1024	43
Destination IP	15	$HOME_NET	5519
		$EXTERNAL_NET	1220
		$HTTP_SERVERS	959
Source IP	11	$EXTERNAL_NET	6952
		$HOME NET	1198
		any	28

So based on the above theorem and corollary, we should combine all possible ranges we could.

In what order we should use the fields to construct the tree, starting from the root, is another issue we need to address. In order to reduce the redundancy effectively, the complexity of fields should increase as we move from the root to leaf nodes. As a case study, Table 3.3 presents the analysis of the complete set of the 8214 Snort rules [70].

While IP addresses have only a few distinct values, port numbers have hundreds of distinct values. In addition, destination IP addresses and port numbers have more distinct values than source IP addresses and port numbers, respectively. Therefore, in order to build the most efficient tree, the protocol type, which is the simplest, should be used in the root level, and source IP address,

destination IP address, source port number, and destination port number should follow in that order. The same tendency is observed in other rule sets we used in experiments. When processing every field of the five tuples in the rule set, for each node in upper level, our algorithm creates continuous and non-overlap ranges, which are represented by all the edges in the corresponding depth in the tree. So, for any income packet, every tuple in its header can match and only match a single edge in every depth of the tree from root to the leaf nodes. So we can assure any packet matches and only matches one path from root node to the leaf nodes, which shows that our algorithm removes complete redundancy in the rule set.

Overall, the non-overlapping rule set building algorithm is shown in Algorithm 1. We insert original rules into the tree one by one, and process different fields from the root to the leaf nodes. During this process, we partition the ranges in each field if needed and then combined all possible continuous branches. And the complexity of Algorithm 1 is $O(n^2)$.

Some applications may have very limited TCAM storage to store all the prefixes even after the number of prefixes is reduced by our algorithm. Even in such cases, our algorithm can still be used as a preprocessor, making actual mapping between prefixes and TCAM entries by other methods easier.

(2). Store Prefixes in a TCAM

After building the minimal range tree, we can extract the decision path from each leaf node to the root. The number of decision paths is equal to the number of leaf nodes. Note that these decision paths cover the entire ranges and there is no overlap. Then we convert the ranges along each decision path into prefixes and store them in a TCAM. When a packet enters the TCAM for comparison, the output bits contain only a single 1 indicating a match, and all the other bits are 0s indicating no match. Because there is exact one match for every incoming packet and there are two decisions “accept” and “discard,” we only need to store a set of rules with the same decision, for

Algorithm 7 Non-Overlapping Rule Set Building

```
Input : A rule set R: r_1, r_2, ..., r_n
Output: A non-overlapping rule set tree
for Each rule from R do
  for Each field from root to leaf do
    if this range exists in this level in the tree and this is not the leaf node level then
      mark the child node as the beginning of next level
    end if
    if this range does not exist in this level in the tree then
      add a new edge labeled with this range
      mark the new node as the beginning of next level
    else
      if this range contains existing ranges or parts of existing ranges in this level in the tree
      then
        split the ranges in the tree based on their overlap
        split the rule and deal with the first one, and insert remaining rules after the current
        rule in R
      end if
    end if
  end for
end for
for Each edge level from leaf node to root node do
  for Each edge from left to right do
    if the current edge and the next edge have the same parent node and have the same subtree
    or node then
      combine these two edges
    end if
  end for
end for
```

example, we store rules with decision "accept," therefore, the incoming packet will be accepted if there is a match in the TCAM entries, otherwise, the incoming packet will be discarded. By this way, at least half of TCAM entries can be further reduced based on the non-overlapping rule set. In the worst case, the number of prefixes consumed by rules with decision "accept" equals to the number of prefixes consumed by rules with decision "discard" and the number of TCAM entries can reduced half. Because we only need to check whether there is a match in the TCAM, therefore, priority encoder circuits (PEC) are not required in our algorithm and only a sequence of OR logic is needed, the incoming packet will be accepted if the result is 1, otherwise, the incoming packet will be discarded. Obviously, the memory accesses used to fetch corresponding decision can be eliminated, this not only reduce the memory hardware resource, but also reduce the overall packet classification process latency.

When storing a range in the TCAM, we need to convert the range into prefixes at first. And the pseudocode for generating the minimum set of prefixes needed to present a given range is shown in Algorithm 2. We showed that w-bit integer range yields at most $2w - 2$ prefixes, and w are 32 and 16 in IP address field and port field respectively, so the complexity of Algorithm 2 is $O(1)$.

And the minimum prefixes needed to present a given range is unique based on the following theorems and corollary:

Theorem 3. *For any two different prefixes, there are only two relationship between them: there is no overlap between them or one prefix is a subset of another.*

Proof. For any two different prefixes p_1 and p_2, we assume their represented ranges partial overlap with each other. So there must be a common value v in both ranges, and the value v matches both prefixes, then we can say p_1 and p_2 are two different prefixes of value v, so one prefix must be the

Algorithm 8 Generating the Minimum Set of Prefixes for A Range

Input : A range $\mathbb{R}$
Output: A set of prefixes P: $p_1, p_2, \ldots, p_n$
convert the low boundary of $\mathbb{R}$ into binary format $\mathcal{R}_l$
convert the high boundary of $\mathbb{R}$ into binary format $\mathcal{R}_h$
if $\mathcal{R}_l$ == $\mathcal{R}_h$ **then**
  add $\mathcal{R}_h$ to P
  return P
end if
while $\mathcal{R}_l \leq \mathcal{R}_h$ **do**
  if $\mathcal{R}_l == \mathcal{R}_h$ **then**
    add $\mathcal{R}_h$ to P
    return P
  end if
  if $\mathcal{R}_h$ is even **then**
    add $\mathcal{R}_h$ to P
    $\mathcal{R}_h - -$
  else
    find the longest continuous 1s from the least significant bit of $\mathcal{R}_h$ and satisfy the condition:
    after converting these 1s in $\mathcal{R}_h$ into 0s, $\mathcal{R}_h \geq \mathcal{R}_l$
    convert these 1s in $\mathcal{R}_h$ into $*$s and add it to P
    convert these $*$s in $\mathcal{R}_h$ into 0s
    if $\mathcal{R}_l == \mathcal{R}_h$ **then**
      return P
    end if
    $\mathcal{R}_h - -$
  end if
end while

prefix of another one and one prefix must be a subset of another, that means one represented range contains another one, not partial overlap, which is a contrary.

Therefore, for any two different prefixes, there are only two relationship between them: there is no overlap between them or one prefix is a subset of another. □

Theorem 4. *For any range, the minimum set of prefixes needed to represent it is unique.*

Proof. In order to present a range $\mathbb{R}$ with minimum prefixes, we can not use two prefixes p_m and p_n when p_m is a prefix of p_n, because p_n is redundant in this situation. Based on the Theorem 1, we know we can not use two prefixes p_m and p_n and they partial overlap with each other. So all the prefixes we need must not overlap with each other. Furthermore, our prefixes need to cover the whole range $\mathbb{R}$, so all the prefixes must continuous and do not overlap with each other. So when converting range $\mathbb{R}$ to prefixes from low boundary and high boundary will result in the same prefix set.

Therefore, For any range, the minimum set of prefixes needed to represent it is unique. □

Corollary 2. *In our non-overlapping rule set tree, the minimum set of prefixes in fields needed to represent it is unique.*

Proof. Because each branch in our non-overlapping rule set tree represent a new rule, and each branch consists of five ranges $\mathbb{R}_1 \ldots \mathbb{R}_5$, and any range $\mathbb{R}_i$ ($1 \leq i \leq 5$) can be a exact value here. Based on Theorem 4, prefixes for each range $\mathbb{R}_i$ is unique, so prefixes needed to represent this branch is unique and the number of prefixes consumed by this branch is the multiplication of numbers of prefixes in these five ranges. Because all the branches are independently when converted into prefixes, the total prefixes in fields to represent our non-overlapping rule set is also unique.

Therefore, in our non-overlapping rule set, the minimum set of prefixes in fields needed to represent it is unique. □

Until now, we have shown our approach to solve the hyperrectangular partitioning problem. Based on the minimal range tree, we propose further optimization in the following subsections.

(3). Extend Our Approach to Deal with Rules with More Decisions

Most packet classification optimization approaches only consider rules with two decisions. In order to make our approach more scalable, we simply modify our approach to deal with unlimited decisions. Our approach can be easily extended to deal with more than two decisions. For example, assume we have decisions $D_1 \ldots D_n$ in our rule set. When building the Minimal Range Tree, we can remove the redundant rules and overlap parts as before, and we also combine possible rules with the same decision as before. However, the approach proposed in the previous subsection does not work when dealing with more than two decisions in the rule set. Therefore, we propose modified approach.

Because there is no overlap in the new rule set and there is exactly one match for each incoming packet, which means we can store the rules in TCAMs out of order, and that also means we can store rules with the same decision together and then we use comparators to replace priority encoder and the memory access can be eliminated. In this situation, we can remove a set of rules with the same decision for TCAMs. Of course, we would like to remove rules with the same decision consume the least TCAM entries. For example, if we do not store rules with decision "discard" in TCAMs, if there is no match in the TCAM, that means the corresponding decision is "discard," otherwise, we need to find out that which block the matched entry belongs to. The architecture is shown in Figure 3.7 and we assume there are seven decisions and we do not store the rules with decision "discard", which is also the decision of the default rule. We store rules

with the same decision together and the comparators store the boundaries used to compare with matched result in TCAM. For example, there are 500 TCAM entries are used to store the first group of rules with the same decision, the first comparator is used to find out whether the matched result is smaller than 500.

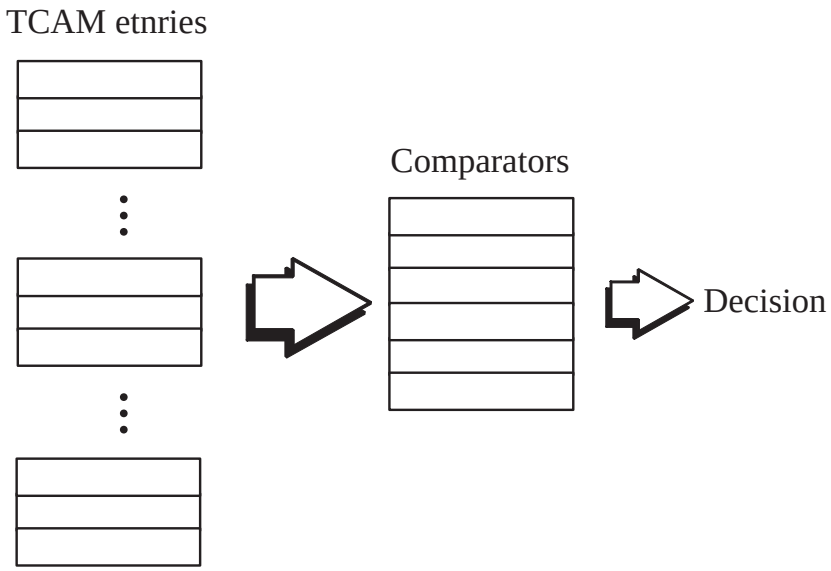


Figure 3.7: Rules stored in TCAMs in groups

As we discussed above, existing TCAM-based approaches need priority encoder and memory access after TCAM matching process, and the priority encoder consumes too much hardware resources and increase the overall latency. In [33], the 32 bits priority encoder consumes 1106 transistors and a maximum power consumption of 13.8 mW, and the latency is about 1.5 ns using 0.15 μm TSMC CMOS technology. For thousands of TCAM entries, the latency of the priority encoder will be longer than the TCAM matching process, which makes the priority encoder the bottleneck of the packet classification. Based on our approach, there is at most one match in the TCAM, so we can simply replace the priority encoder with an ordinary encoder. In order to further

reduce this bottleneck in the packet classification, we use leading-zero counter circuit to replace encoder and we will show the leading-zero counter circuit in the next subsection.

(4). Leading-zero Counter Circuit

Usually, there are thousands of TCAMs entries which make the priority encoder very large, so how to design efficient priority encoder is also important when using TCAMs. So we design our encoder as follow to replace commonly used encoders. We only need to calculate the maximum number of continuous zeros start from the highest end. The logic is shown in Figure 3.8.

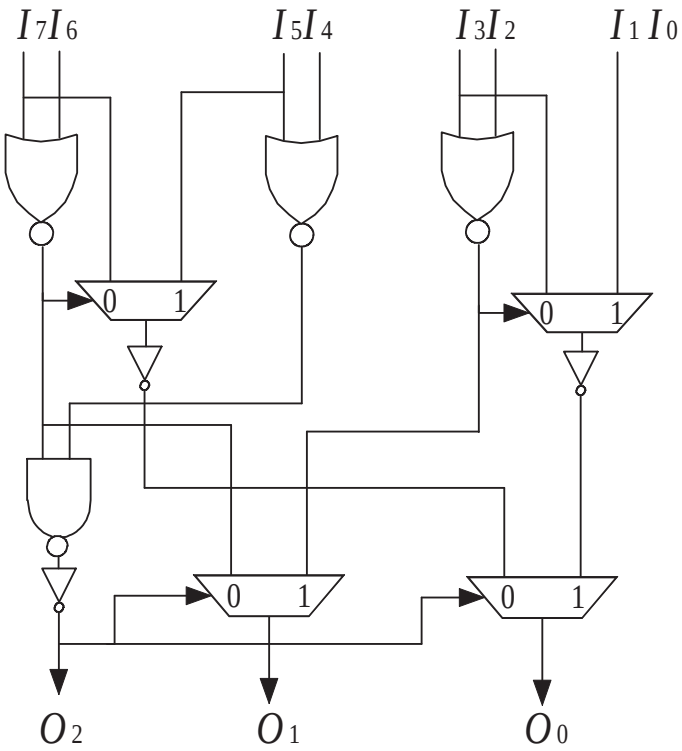


Figure 3.8: The 8-bit leading-zero counter

In Figure 3.8, there are only 8-bit input represents the 8-bit TCAM entries matching output, and it can be easily extended to meet the length of TCAM entries output. We assume that there are

Table 3.4: Gates used by our leading zero counter and the ordinary priority encoder with 8-bit input

Gates	Number of Transistors(CMOS)	Our Algorithm	Ordinary priority encoder
2-input NOR	4	3	1
2-input NAND	4	1	1
3-input NAND	6	0	2
4-input NAND	8	0	3
2-input Multiplexer	2	4	0
Not	2	3	8
3-input AND	8	0	2
4-input AND	10	0	1
4-input OR	10	0	3

at least one entry matching so we do not need to consider the lowest input bit I_0. And the output $O_2O_1O_0$ is the number of leading zeros. We compare the logic gates used by our algorithm and the ordinary priority encoder shown in Table 3.4.

This extension scheme can reduce the TCAM entry requirement, but it can only be used in one field to reduce the entries, we use it in the source port in our implementation. In order to further reduce the number of entries, we must consider another field with many ranges, which is destination port in our implementation. The number of entries used is equal to the multiplication of different fields as we discussed above, so we can reduce the entries significantly if we can separate these two fields to convert the multiplication to addition.

In our implementation, we separate the destination port field from other fields and call them part2 and part1 respectively. After the matching lookup and encode in part1, we will get the address of the matching address of part1, we use this address to find the corresponding address in part2 after access a pre-computed address converter, which stored in a SRAM chip or on chip. The output of the converter is a encoded bit mask, which is used to filter out rules higher than the matching rule. So we can get the final encoded address from part2, and then access the memory to

get the result. This process is designed in pipeline to accelerate the speed as shown in Figure 3.9.

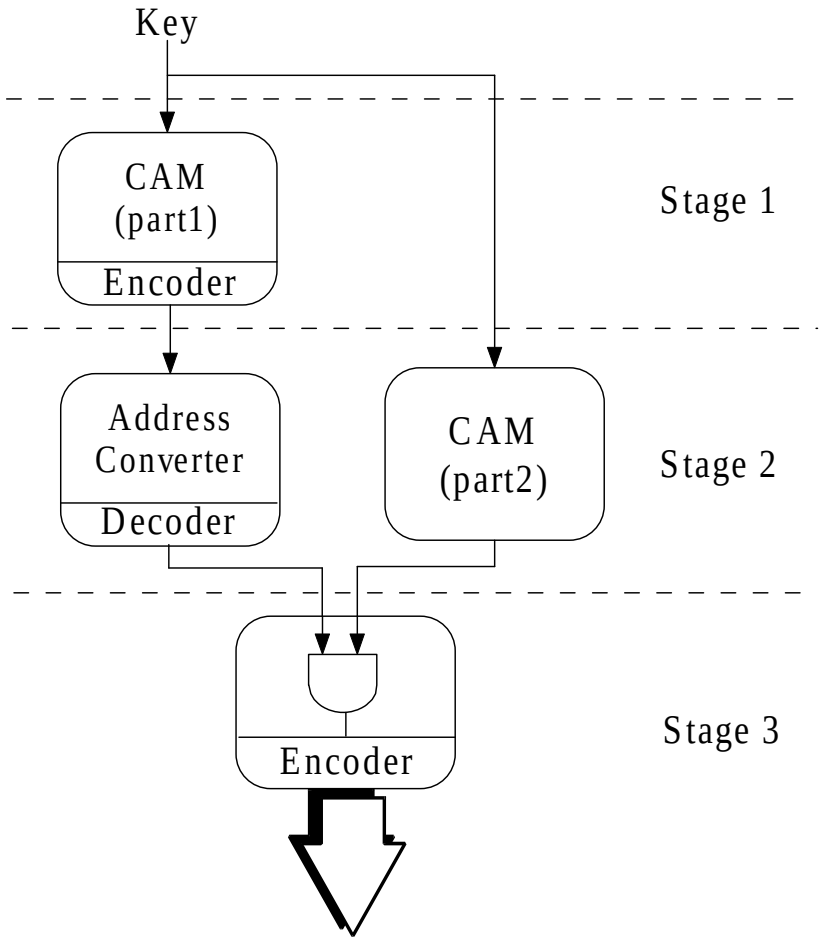


Figure 3.9: Pipelined architecture of separated fields

The logic used to generate mask is shown in Figure 3.10.

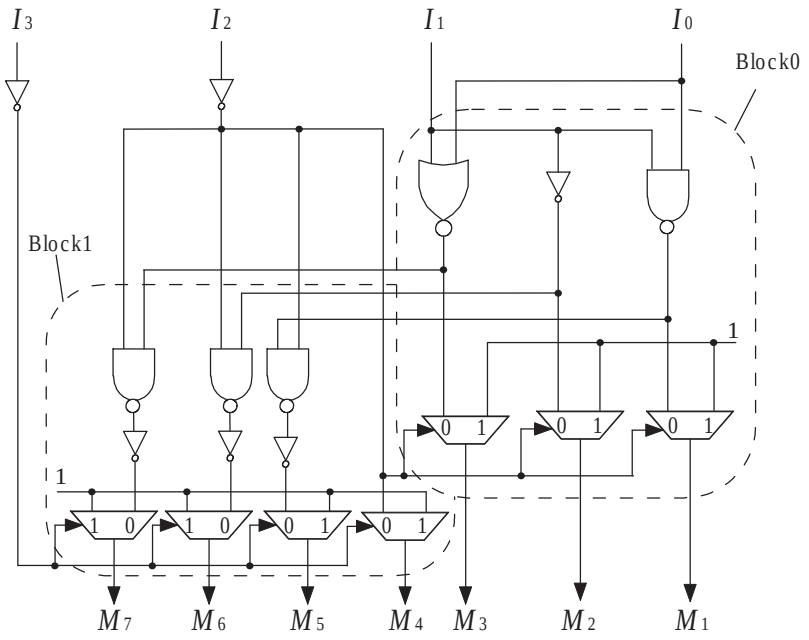


Figure 3.10: The 8-bit mask generater

In Figure 3.10, we use 3-bit input I_2 to I_0 to generate the 8-bit mask M_7 to M_0. The 3-bit input represents the number of leading zeros in the mask and all the other bits are set to value 1. We assume that there are at least one entry matching so the value of M_0 must be 1 and is omitted in Figure 3.10. The input I_3 is only used to extend our scheme to meet longer mask requirement. Actually, we use the radix-4 scheme to extend the logic, the block0 is used to generate the least 4-bit output, the block1 is used to generate higher 4-bit output, and it can be easily to extend the length of mask.

(5). Comparators

Because there are usually very limited decisions in a rule set, only a small number of comparators, actually adder circuit, are needed, and we need adders which achieve a good balance between calculation speed and hardware resource consumption. Because adders have been well

studied, so we will not discuss too much about this, we use the adder proposed in [71] to meet our requirement, and all the details can be found there.

3.3.3 Simulation Results

Because our main goal is reducing the TCAM entries consumption and optimize post TCAM match process, so we first simulate the number of TCAM entries can be reduced by the minimal range tree approach. Finally, we simulate the area reduction can be achieved by our leading-zero counter approach.

(1). Experimental Results on Minimal Range Tree

For experiments, we collected rules generated from [72]. Because rule sets vary across different applications, we gathered as many rules as we could and then randomly selected one thousand rules from them. We compared the number of TCAM entries used by original rules and the number of TCAM entries used after reducing redundancies with our algorithm. The result is shown in Figure 3.11.

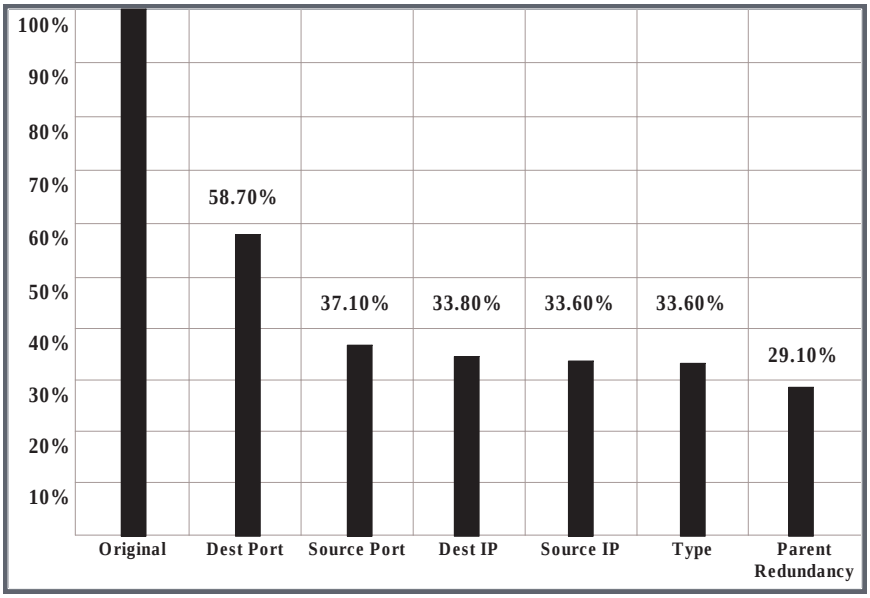


Figure 3.11: Ratio of TCAM entries left after redundancy removal

After reducing redundancies for each level of the tree, starting from leaf nodes, we get the percentage of TCAM entries left. We can see that the redundancy of the destination port field alone affects the number of required TCAM entries most; 41.3% of TCAM entries are reduced. On the other hand, the IP address fields have only a small impact, and the reduction by the protocol type field is almost negligible. Finally, the parent redundancy removal reduces 4.5% TCAM entries. Overall, our algorithm reduces 70.9% of TCAM entries in this experiment.

As we discussed above, because there is exact one match for every incoming packet and there are two decisions “accept” and “discard,” we only need to store a set of rules with the same decision in the TCAM. By this way, at least half of TCAM entries can be further reduced based on the non-overlapping rule set. In the worst case, the number of prefixes consumed by rules with decision “accept” equals to the number of prefixes consumed by rules with decision “discard” and the number of TCAM entries can reduced half. In this situation, TCAM entry reduction can

achieve 85.4%.

For comparison with other existing approaches, we selected the latest redundancy removing algorithm, which was proposed by Liu et al [73]. Compared with their approach, our approach not only removes the redundant rules but also removes some parts of other rules, which we called partial redundant rules, furthermore, we combine continuous rules with the same decision and our approach only needs to store a set of rules with the same decision in the TCAM. Using the same rules, only removing redundant rules can reduce 51.6% TCAM entries. Therefore, our approach reduces 33.8% ($85.4\% - 51.6\%$) more TCAM entries.

When there are two decisions, priority encoder and the memory accesses used to fetch corresponding decision can be eliminated entirely, so the hardware resource and latency reduction depend on the number of TCAM entries used. So we do not need to simulate them and we discuss them in discussion section.

(2). Area Reduction by Leading-Zero Counter

When dealing with more than two decisions in the rule set, our leading-zero counter will be applied. Based on the information of transistor consumption in Table 3.4, we can know that our leading-zero counter circuit only consumes 32.8% (38/116) transistors of the consumption of an ordinary priority encoder. The simulation in Xilinx Virtex-5 FPGA shows that the area consumption of our leading-zero counter circuit is about 46.2% of the area consumption of an ordinary priority encoder.

3.3.4 Discussion

Because we eliminate the usage of priority encoder and RAM, so the critical path is the delay of TCAM itself. And the merit of RAM removal is significant, and we discuss this in two situations. First, if we use off-chip memory, which means the TCAM and RAM are in different chips. In

this situation, we have enough RAM resource but the latency is much longer than on chip memory access, so our approach can remove such latency. Second, if we use on-chip memory, which means the TCAM and RAM are in the same chip. In this situation, the memory access is much faster than off-chip memory, but the RAM resource is very limited due to the high complexity and higher power of TCAM circuitry. With large number of TCAM entries, it is difficult to store all decisions in the on-chip RAM. So our approach can remove such RAM consumption.

Due to the original rules should be stored in the TCAM in priority descending order, so updating a rule in the TCAM entry usually affects other TCAM entries, and the whole TCAM entries needed to be rewrite in the worst case. Even though the TCAM search process is very fast and usually costs a single clock cycle, the update process is much slower than the search process; for example, a single entry update process consumes 16 times as many clock cycles as a search operation for TCAM entries in FPGA [74]. This update process degrades the performance of packet classification because the packets must be buffered during the update. Therefore, a fast update scheme becomes more and more important with the increasing number of rules in a rule set and the increasing frequency of rule updates. Our approach can provide fast TCAM update due to the non-overlapping property. As we discussed above, we can store the non-overlapping rules in the TCAM out of order, so when updating a rule, only related TCAM entries needed to be modified, and all others remain the same.

The advantage of our approach compared with complete removal of redundant rules [73] is that, our approach not only removes the redundant rules but also removes some parts of other rules, which we called partial redundant rules, furthermore, we combine continuous rules with the same decision. And these lead to two major different results: first, there is no overlap in our new rule set and there is only one match for each incoming packet, but approach in [73] may also have multiple matches for each incoming packet because there may still have overlaps between remain-

ing rules; second, redundancy removal algorithm in [73] is complete, but the removed set of rules is not guaranteed to be maximal because there may be different new rule set when processing rules in different sequences. Our approach does not have such problem because we do not only remove redundant rules but also remove partial redundant rules and combine continuous rules, and different processing sequences of rules will result in the same result as we have proved above. We have proved that our approach can achieve the minimum number of TCAM entries when using prefixes in different fields to represent a non-overlapping rule set tree, so we can achieve the optimal hyper-rectangular partitioning using prefixes based on our non-overlapping rule set tree. However, our approach can not guarantee the minimum number of TCAM entries consumed by a given rule set due to two reasons and we analyze them separately.

First, we have proved that our approach can achieve the minimum number of TCAM entries when using prefixes in different fields to represent a non-overlapping rule set tree, but we can build different non-overlapping rule set trees with different sequences of fields as we discussed in section 4.1. If we convert the original rule set into prefix format directly, different sequences of fields will result in the same number of TCAM entries because all the original rules are independent and any change of field sequence will only result in fields relocation in TCAM entries. However, we will reduce TCAM entry consumption when we build the non-overlapping rule set tree and different sequences of fields will result in different TCAM entry reduction. To solve this problem, the brute force approach is to calculate all the possible TCAM entry consumptions and choose the minimum one, but this will consume much more time. As we discussed the sequence we selected, our strategy is to reduce redundancy and combine ranges as many as possible because they can reduce the number of prefixes needed to represent the rule set.

Second, we have proved that our approach can achieve the minimum number of TCAM entries when using prefixes in different fields to represent a new non-overlapping rule set, but we

cannot guarantee the minimum usage of TCAM entries because the wildcard can happen anywhere in a TCAM entry, not only happens in the suffixes in each field. The problem of finding the minimum number of TCAM entries to store a rule set is a 104-hypercube problem, where 104 is the total length of 5-tuples. Each node connects with nodes with only one different bit, so each node connects with other 104 nodes, and we can combine connected two nodes with the same decision, and the problem is that we should find the minimum nodes. The different combination sequences can result in different results. Similarly, based on our non-overlapping rules, we can combine every two entries with exact one different bit and with the same decision. By this way, we will not introduce new overlap but different combination sequences may result in different results. For example, in Figure 3.12, we have two combination options: first, combine r_1 and r_2, then combine r_4 and r_5; second, combine r_1 and r_3, then combine r_4 and r_5, at last, combine them together. So different combine sequences will result in different results.

$r_1 : 0000 * * \rightarrow \text{accept}$ $r'_1 : 00 * 0 * * \rightarrow \text{accept}$ $r''_1 : * * 00 * * \rightarrow \text{accept}$
$r_2 : 0010 * * \rightarrow \text{accept}$ $r'_2 : 0100 * * \rightarrow \text{accept}$ $r''_2 : 0100 * * \rightarrow \text{accept}$
$r_3 : 0100 * * \rightarrow \text{accept}$ $r'_3 : 1 * 00 * * \rightarrow \text{accept}$
$r_4 : 1000 * * \rightarrow \text{accept}$
$r_5 : 1100 * * \rightarrow \text{accept}$

Figure 3.12: Combine rules with different sequences

Therefore, there are still two problems unsolved: 1) Is there any algorithm that can be used to bound the difference between our prefix-based minimum number of TCAM entries approach and the minimum number of TCAM entries without overlap? 2) Is the problem of detecting minimum number of TCAM entries without overlap in a rule set NP-hard?

3.3.5 Summary

In this section, we propose a tree-based overlapping removal algorithm to remove redundant rules and combine overlaying rules to build new rule sets in packet classifiers, and then remove a set of rules with the same decision based on the non-overlapping rule set. This equivalent transformation can significantly reduce the number of TCAM entries needed by a packet classifier and our experiments show a reduction of 85.4% in the number of TCAM entries after performing the two steps. Based on the properties of non-overlapping rule set, we optimize the post TCAM match process. When there are only two decisions "accept" and "discard" in the rule set, the priority encoder can be removed and the memory access can be eliminated, because all the rules stored in TCAM has the same decision, and we know the decision by knowing whether there is a match in the TCAM entries. When there are more than two decisions in the rule set, we can store rules in TCAM entries out of order, so we store rules with the same decision together and use Leading-zero counter to replace the priority encoder and the simulation result shows about 53.8% area reduction in FPGA by using Leading-zero counter. And we also replace the SRAM memory with comparators to reduce processing latency. Furthermore, our approach provides fast rule updating process. In some applications, our algorithm can be used as preprocessor, which combined with other algorithms to achieve better results.

3.4 Range Extension for TCAM-Based Packet Classification

3.4.1 One-Directional Range Extension Algorithm

In order to future reduce the TCAM consumption after redundancy removal proposed in previous section, we first propose the one-directional range extension approach as following.

Based on the minimal range tree, there is no overlap and no gap between different rules, and all new rules have been sorted. So we can extend the range towards the same direction and we extend ranges towards "0" in this section. We extend the ranges because the minimum range does not guarantee the minimum number of represented prefixes, which means extended large range may consumes less TCAM entries than the original one, so we find the extended range which consumes the minimum number of TCAM entries for each range. There will be overlap between different rules after using range extension approach, which means there maybe multiple matches for a single income packet, but the priority encoder will choose the correct result, and that's why we must extend the ranges towards the same direction. For example, a port range [15, 1023] will be extended to [0, 1023], then the corresponding TCAM entries will be reduced from 10 to 1, and the matching result will be unchanged.

A simple example is shown in Fig. 3.13. The original scheme with five rules is shown in Fig. 3.13 (a), and the corresponding new scheme is shown in Fig. 3.13 (b). We can see that the new rule set has different ranges and the entries requirement is reduced from 12 to 7, note that we do not need to store the rule in TCAM. Actually, the range extension will introduce redundancy to the rule sets, when a packet with a value "0010", it matches r_1 based on Fig. 3.13 (a), and it will match r_1, r_2 and r_3 in Fig. 3.13 (b), but the priority encoder will chose r_1 as the final result, which is the same as in Fig. 3.13 (a). This is also the reason we extend all rages towards the same direction instead of two directions.

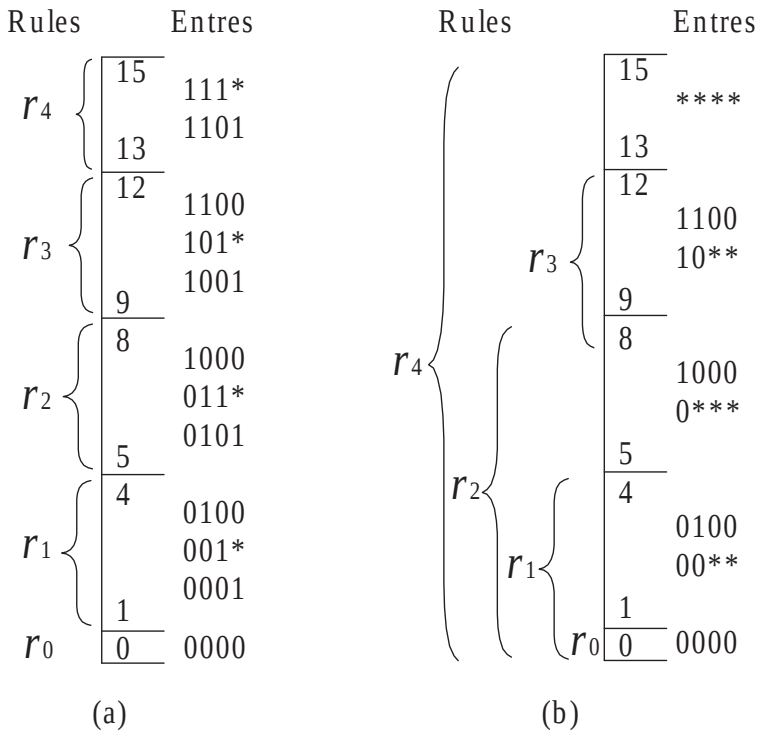


Figure 3.13: TCAM entries left after different redundancy removal

So we use the following algorithm to find the minimum number of entries for each rule. For each range, name the lower boundary B_l and the upper boundary B_u.

(1) Find the longest same prefixes of B_l and B_u, called P_m with length of m-bit.

(2) The remainder part of lower boundary and upper boundary is called R_l and R_u respectively with length of n-bit, and $m + n = 16$. It is obvious that the highest bit of R_l and R_u is 0 and 1 respectively.

(3) The extended prefix is P_m followed by a "0" and $n - 1$ "don't-care" states.

(4) The remaining entries are generated from P_m followed by n zero to B_u.

The algorithm using C-code is shown in Fig. 3.14. For the worst case, the single interval [1, 65534] requires only 16 TCAM entries in our algorithm instead of 30 entries, which reduces

about 46.7%.

The priority encoder will choose the highest matching entry, which has the result bit of 1 from all the matching entries, which is the correct result.

In Fig. 3.15, a simple example with the range from to is shown to compare the ordinary approach in Fig. 3.15 (a) and our approach in Fig. 3.15 (b). In Fig. 3.15 (a), the dash line is used to separate the same prefix and remainder parts, and three TCAM entries are generated: E_0, E_1, and E_2. In Fig. 3.15 (b), only two TCAM entries are generated: E_0 and E_1, and the low boundary of the range will be ignored, which can increase the processing speed.

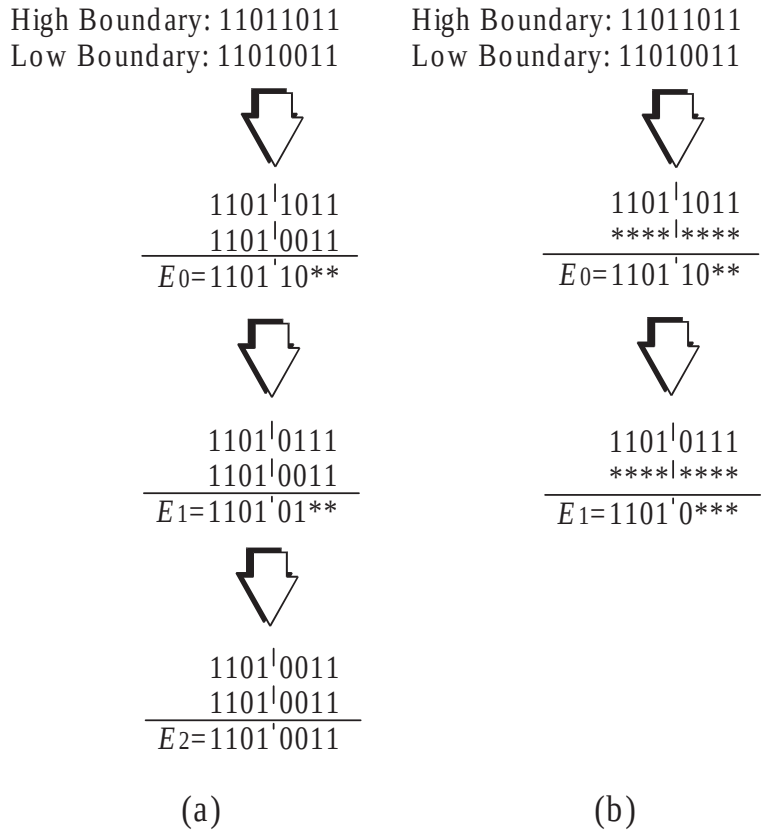


Figure 3.15: A simple example of TCAM entries generations using ordinary approach and our approach

3.4.2 Bidirectional Range Extension Algorithm

After removing redundancies with the minimal range tree, there remains no overlap between different rules, and thus we can sort all the new rules by their ranges in a field. However, this does not solve the range explosion problem; it only mitigates it. Still, a single range may need multiple TCAM entries to represent itself. In the worst case, $2m - 2$ entries are needed to store a single m-bit range. For a 16-bit port range, it may need 30 TCAM entries. In order to further reduce the

number of TCAM entries, we propose a bidirectional range extension, which only needs at most two TCAM entries to represent a single range regardless of the length of the range.

Given a list of ranges, we assume that every value belongs to exactly one range, that a range appearing earlier in the list has a higher priority than a range appearing later, and that the output of range matching for each range is binary, either 1 or 0. Note that applying the minimal range tree algorithm described in Sec. 3.3 and sorting resulting ranges by the boundary values yield a list of ranges satisfying these assumptions. Assuming these, we take the following steps to reduce the number of required TCAM entries.

We divide each range into two parts, each of which is represented as a single prefix. One part is called an upward extension and the other a downward extension. It is demonstrated in Fig. 3.16. Given a range, e.g., [100001, 110101], find a number N that belongs to the range and has the most consecutive 0's starting at the least significant bit (LSB). In our example, N will be 110000. Convert all the rightmost consecutive 0's of N into "don't care" bits. The result will be the upward extension (master entry). If N is equal to the lower bound of the original range, this range does not need a downward extension. Otherwise, the numeric prefix (excluding "don't care" bits) of the upward extension minus 1 (10 in our example), followed by the same number of "don't care" bits will be the downward extension (slave entry) as shown in Fig. 3.16(a). We use the set of all upward extensions as the master set and the set of downward extensions as the slave set. Note that the master set has the same number of entries as the number of ranges in the original list. On the other hand, the slave set only contains entries for those ranges where the original range is not a subset of the upward extension.

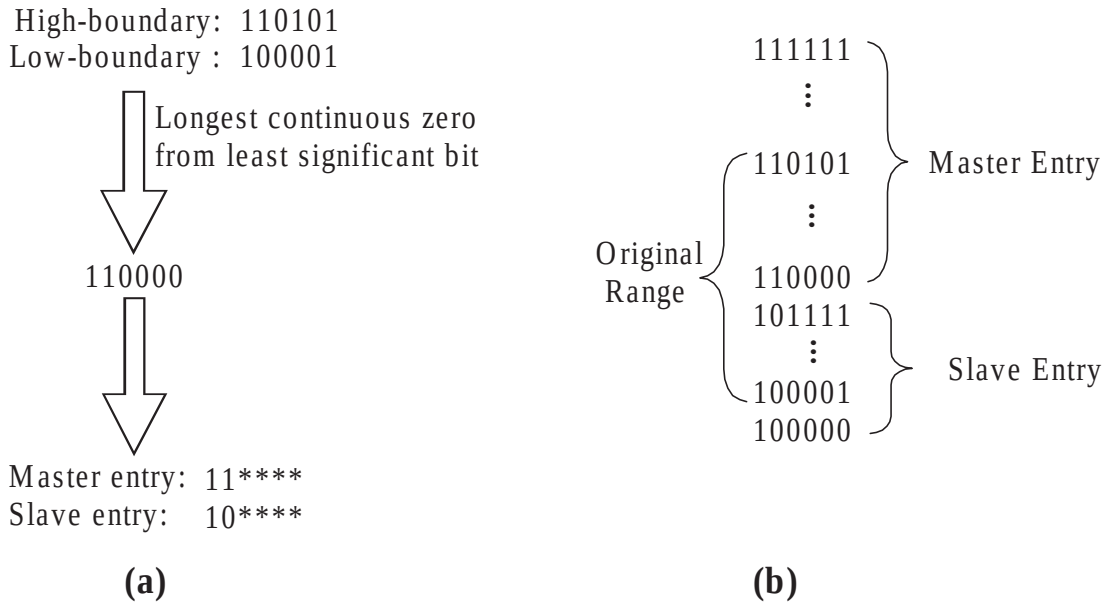


Figure 3.16: An example of how to divide a range and generate master/slave entries

Below, We prove that the original range is a subset of the master and slave entries combined.

Proof. Assume that the range is $[N_{\mathrm{L}},\ N_{\mathrm{H}}]$. Suppose $N_{\mathrm{L}} = \sum_{i=0}^{n-1} b_i^{\mathrm{L}} 2^i$ and $N_{\mathrm{H}} = \sum_{i=0}^{n-1} b_i^{\mathrm{H}} 2^i$, where $b_i^{\mathrm{L}}, b_i^{\mathrm{H}} \in \{0, 1\}$. Let N be the number in $[N_{\mathrm{L}},\ N_{\mathrm{H}}]$ that has the longest consecutive zeros (n_0 bits) starting at LSB. Thus, we have $N = 2^{n_0} + \sum_{i=n_0+1}^{n-1} b_i 2^i$, where $b_i \in \{0, 1\}$. Then the master entry is a range $[N,\ N_{\mathrm{master}}]$, where $N_{\mathrm{master}} = N + \sum_{i=0}^{n_0-1} 2^i$ with at least consecutive $n_0 + 1$ 1's starting at LSB. Therefore, $N_{\mathrm{master}} + 1 = 2^{n_0+1} + \sum_{i=n_0+1}^{n-1} b_i 2^i$, which has at least $n_0 + 1$ consecutive 0's starting at LSB, or more 0's than N, and thus should not be in $[N_{\mathrm{L}},\ N_{\mathrm{H}}]$. Therefore,

$$N_H \leq N_{\mathrm{master}}\,. \tag{3.3}$$

Similarly, the slave entry, if we have one, is a range $[N_{\mathrm{slave}},\ N - 1]$, where $N_{\mathrm{slave}} = N - 2^{n_0} = \sum_{i=n_0+1}^{n-1} b_i 2^i$. Note that N_{slave} has at least $n_0 + 1$ consecutive 0's starting at LSB, or

more 0's than N, and thus should not be in [N_{L}, N_{H}]. Therefore,

$$N_L > N_{\mathrm{slave}}\,. \tag{3.4}$$

By Eq. 3.3 and Eq. 3.4, [N_{L}, N_{H}] is a subset of [N_{slave}, N_{master}].

□

Note that each of the master and slave entries can be represented by a single prefix, and thus by a single TCAM entry.

A field of an incoming packet is compared against both the master and slave sets; each set is stored in a separate TCAM. After knowing the decision from the master set, we need to find out whether the matched entry has a corresponding entry in the slave set. If there is no such entry, the decision in the master set becomes the final decision; otherwise, we need to consider the corresponding slave entry. In the slave set, we obtain the decision and the low-boundary (N_{L}) of the corresponding entry. If both decisions, one from the master set and the other from the slave set, are identical, that becomes the final decision; otherwise we compare the low-boundary from the slave set with the input key. If the low-boundary is smaller than the key, we choose the decision from the slave set; otherwise we choose the decision from the master set.

For the worst case, the single interval [1, 65534] requires only 2 TCAM entries in our algorithm instead of 30 entries, which reduces about 93.3%.

3.4.3 Simulation Results

Because our main goal is reducing the TCAM entries consumption, so we first simulate the number of TCAM entries can be reduced by the minimal range tree approach, and then simulate the number of TCAM entries can be reduced by the range extension approach based on the new rules generated

Table 3.5: Analysis of a single range in one-directional range extension

The Range in the field (bits)	Total Entries used by Our Algorithm	Total Entries used by Normal Algorithm	Entries Saved (%)
1	1	1	0.00
2	8	10	20.00
3	48	65	26.15
4	256	355	27.89
5	1280	1831	30.09
6	6144	9103	32.51
7	28672	43935	34.74
8	131072	206911	36.65
9	589824	955007	38.24
10	2621440	4335871	39.54
11	11534336	19421695	40.61
12	50331648	86033407	41.50
13	218103808	377595903	42.24
14	939524096	1644400639	42.87
15	4026531840	7114039295	43.40
16	17179869184	30602706943	43.86

from minimal range tree. Finally, we simulate the area reduction can be achieved by our leading-zero counter approach.

(1). Analysis of Range Extension

In this subsection we analyze the performance of our range extension algorithm on a single range, such as the destination port range. First, we assume every possible range has the same possibility 1 appear in the rule set, so the rule set contains ranges [0, 1], [0, 2], ..., [0, $2^n - 1$], [1, 2], [1, 3],...,[$2^n - 2$, $2^n - 1$] in average, where is the number of bits of this field. And the simulation results are shown in Table 3.5.

The percentages of entries saved with different field widths are shown in Fig. 3.17. We can see that the percentages of entries saved by our algorithm increase with the field width increase,

and we can save 43.86% entries of a 16-bit port number, and for the rule sets we collected from real world show that our algorithm can save 33.75% percent and 31.98% entries for the destination port ranges and source port ranges respectively in average. For a single 16-bit port number, our approach reduces the average number of entries from 14.25 by ordinary approach to 8.

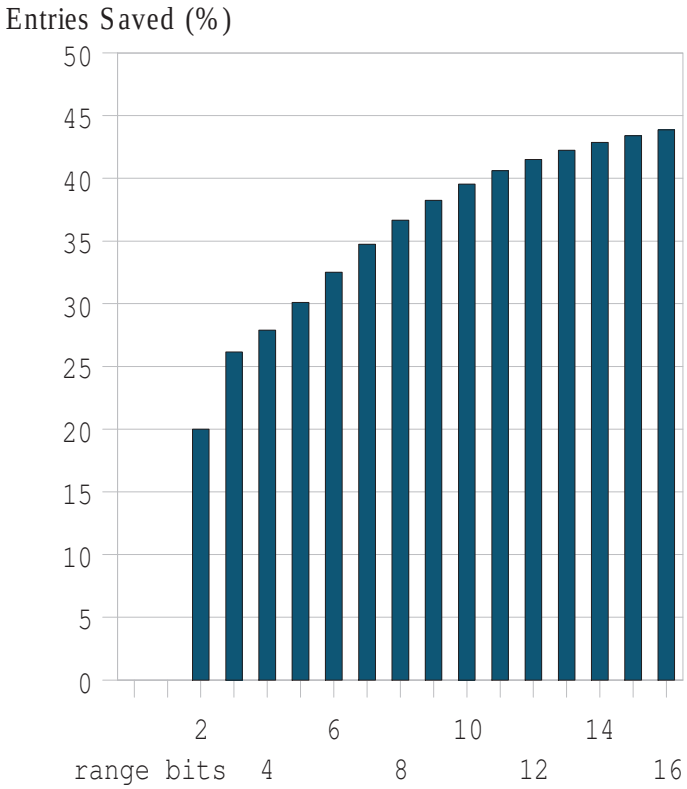


Figure 3.17: The entries saved with different field width

For both destination port and source port our algorithm can further save the entries because of the multiplication has been replaced by addition. So for the situation shown in Table 3.5 our algorithm can save 88.35% for the ranges in both Source port and Destination port in average, and for the rule sets we collected from real world can save 28.8% based on our non-redundant rules.

Table 3.6: Analysis of a single range in bidirectional range extension

Range length (bits)	# of TCAM entries with range extension	# of TCAM entries w/o range extension	Saving (%)
1	1	1	0.00
2	9	10	10.00
3	49	65	24.62
4	225	355	36.62
5	961	1831	47.52
6	3969	9103	56.40
7	16129	43935	63.29
8	65025	206911	68.57
9	261121	955007	72.66
10	1046529	4335871	75.86
11	4190209	19421695	78.43
12	16769025	86033407	80.51
13	67092481	377595903	82.23
14	268402689	1644400639	83.68
15	1073676289	7114039295	84.91
16	4294836225	30602706943	86.00

So combine the minimal range tree algorithm with the range extension algorithm, we can reduce 79.28% (1-29.1% $\times$ 71.2%) entries for the real world rule sets we collected.

(2). Analysis of Bidirectional Range Extension

In this subsection we analyze the performance of the proposed bidirectional range extension algorithm on a single range, such as the destination port range. Given the number of bits for the field, we generate every possible range whose length is greater than 1 with the same probability. Thus, the generated rule set may include ranges [0, 1], [0, 2], . . . , [0, 2^n-1], [1, 2], [1, 3],. . . ,[2^n-2, $2^n - 1$], where n is the number of bits of this field. And the simulation results are shown in Table 3.6.

The percentages of entries saved with different field widths are shown in Fig. 3.18. We can see that the percentage of entries saved by our algorithm increases as the field width increases,

and we can save 86.00% of TCAM entries of the 16-bit port number field. For the rule sets we collected from [72], our algorithm can save 77.47% and 74.81% entries for the destination port field and source port field, respectively, excluding non-range rules. For a single 16-bit port number, our approach reduces the average number of entries from 14.25 to 2.

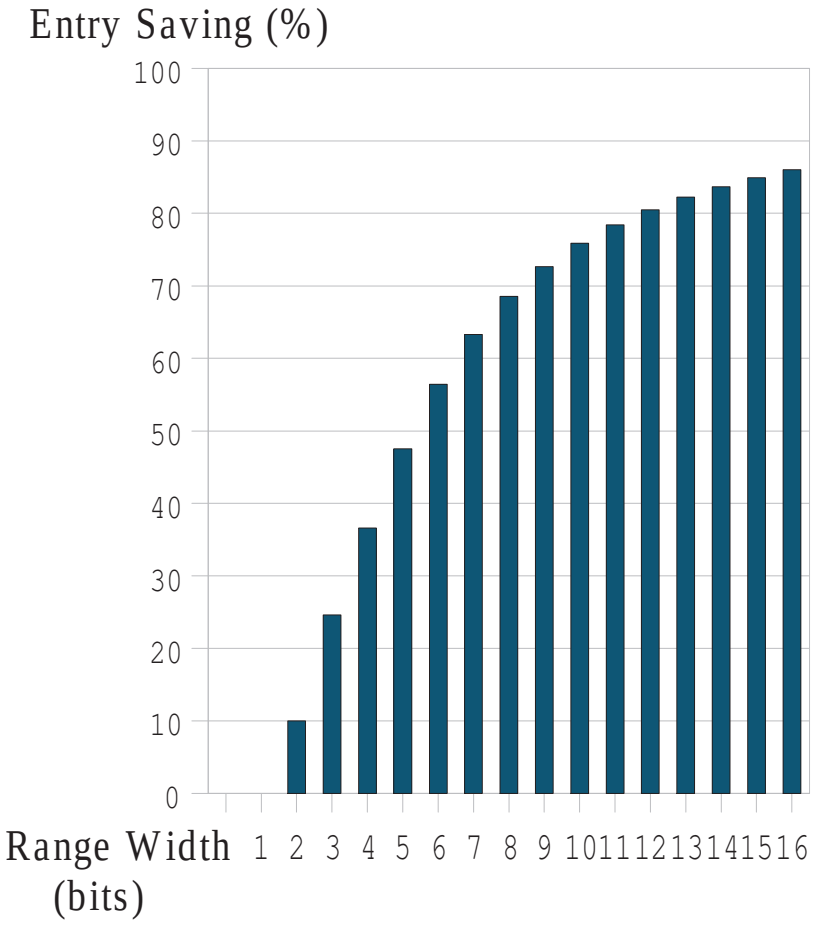


Figure 3.18: The Entries saved with Different Field Width

For a rule with both destination port and source port ranges, our algorithm can further save TCAM entries because of the multiplication effect. For the scenarios shown in Table 3.6, our

algorithm saves 98.04% of TCAM entries for randomly generated rules with both destination port and source port fields, and saves 82.78% for the ranges included in the real-world rule sets. The difference between them is mainly caused by the fact that some popular ranges, such as [0, 1023], can be represented by a single prefix in real-world rule sets. If we include rules without ranges in the simulation, our algorithm can save 53.18% TCAM entries for the real-world rule sets.

From both redundancy removal using the Minimal Range Tree and the bidirectional range extension, we can reduce 84.27% of entries for the real-world rule sets with additional circuits such as two adders and a little latency.

3.4.4 Discussion

The one-directional range extension algorithm and bidirectional range extension algorithm have some different properties. The one-directional range extension algorithm can only extend the ranges toward the same direction to maintain the priority sequence of original rules. And the bidirectional range extension algorithm first divides a range into two ranges, and then extends the ranges upward and downward to make each extended range consumes at most one TCAM entry. The result is that each range consumes at most two TCAM entries. Therefore, the bidirectional range extension algorithm can reduce more TCAM entries, but the rule set cannot be stored in a single TCAM, and it is more suitable to be pipelined in ASIC or FPGA. And the advantage of the one-directional range extension algorithm over bidirectional range extension algorithm is that the one-directional range extension approach does not need to modify the original TCAM-based packet classification architecture because all the five fields can be stored in a single TCAM.

3.4.5 Summary

In this section, we first propose a tree-based redundancy removal algorithm to remove redundant rules and combine overlaying rules to build new rule sets in packet classifiers, and then propose one-directional range extension algorithm and bidirectional range extension algorithm based on the new rule set. This equivalent transformation can significantly reduce the number of TCAM entries needed by a packet classifier and our experiments on random selected rules from real packet classifiers. The final experiments show a reduction of 70.9% in the number of TCAM entries after performing non-overlapping algorithm, Furthermore, 79.28% and 84.27% reductions in the number of TCAM entries after performing the one-directional and bidirectional range extension algorithms respectively. Furthermore, we replace priority encoding circuit with leading-zero counter circuit and the simulation result shows about 53.8% area reduction in FPGA. In some applications, our algorithm can be used as preprocessor, which combined with other algorithms to achieve better results.

3.5 A Fast Update Scheme for TCAM-based Packet Classification

In this section, we propose a fast update scheme to meet the fast matching requirement with small TCAM entries requirement. Our scheme eliminates the requirement of sorted rules in the TCAMs by removing all the redundancy in the rule set to make sure at most one entry matches a incoming packet, which allows out of order storage in the TCAMs. So updating a rule only affect the corresponding TCAM entries, which reduce the update latency significantly. And our scheme also reduces the number of TCAM entries used by a packet classifier.

Because TCAMs need to store a sorted list, so the slow update problem also happens in TCAM-based routing lookup tables, and some algorithms are designed to optimize this prob-

lem [75–77]. In [75], the authors claim they propose the first algorithmically algorithm to optimize update process on a TCAM, and most TCAM vendors live with an $O(N)$ worst case update process. They propose two algorithms to constrain the update process latency small in the worst case. [76] proposes a TCAM-based fast update scheme for IPv6 routing lookup architecture, it uses multiple TCAMs to store prefixes with different lengths and the TCAM space management controls four storage space and two empty spaces. And the simulation shows good result on the update process. In [77], the authors analyse the requirements for inserting a new entry into TCAM, and design a TCAM management scheme based on the properties of prefixes to eliminate unnecessary sorting such as only the overlapping prefixes need sorting. All these approaches are capable to deal with the destination IP address problem in routing lookup tables, but can not be used in the much more complex five-tuple packet classifiers, which includes Protocol, Source IP address, Destination IP address, Source port, and Destination port. Furthermore, the ranges in Source port and Destination port usually cause the range expansion problem as we discussed, so the compression approaches should be used before stored in the TCAM, which makes the update optimization more difficult. Our work first reduces the redundancy in the rule sets and then store the new rule set in TCAMs out of order to reduce the TCAM entry shift frequency.

To the best of our knowledge, there is no previous algorithm to optimize the TCAM update for packet classification. Ordinary TCAM chips need $O(N)$ update time in the worst case, where N is the depth of TCAM.

3.5.1 Proposed Algorithms for Fast Packet Classifiers

Our approach is based on the previous minimal range tree approach. In order to reduce the update latency, we must reduce the number of TCAM writing. That because the writing time is usually 16 times of reading time for a TCAM entry. Furthermore, the bandwidth is limited especially for

the wide TCAM entries. So our algorithm only modifies related TCAM entries based on the out of order storage.

We use the same example to show our approach and we assume there are five original rules in the rule set shown in Figure 3.19.

```
int boundary_l;    //lower boundary
int boundary_u;    //upper boundary
int boundary_nl;   //new lower boundary
int prefix;        //the same prefix
int m;             //length of the same prefix
int remainder;     //remainder part of upper boundary
int shift_cnt = 0;  //shift counter
int entry_cnt = 0; //entry counter
int entry[1024];    //TCAM entry
int mask[1024];     //corresponding mask

prefix = ~ (boundary_l ^ boundary_u);
m = prefixlength(prefix);
prefix = prefix & (0xFFFF << (16 - m));
entry[0] = prefix;
mask[0] = 0xFFFF << (15 - m);
boundary_nl = prefix | (1 << 15 - m);
remainder = boundary_u & (0xFFFF >> m);
if(remainder%2 == 0) {
     entry[entry_cnt] = prefix | remainder;
     mask[entry_cnt ++] = 0xFFFF;
     remainder -= 1;
 }
 while(remainder != 0){
    if(remainder % 2 == 0) {
       entry[entry_cnt] = prefix | (remainder << shift_cnt);
       mask[entry_cnt ++] = 0xFFFF << shift_cnt;
       remainder -= 1;
    }
    else{ //the last bit is one
       do{
          remainder >>= 1;
          shift_cnt ++;
        } while(remainder % 2 != 0);
       entry[entry_cnt] = prefix | (remainder << shift_cnt);
       mask[entry_cnt ++] = 0xFFFF << shift_cnt;
    }
}
```

Figure 3.14: The C code of proposed entries generater

$$
\begin{aligned}
r_1: &\quad F_1 \in [0,4] \wedge F_2 \in [0,9] \quad \rightarrow \text{accept} \\
r_2: &\quad F_1 \in [0,4] \wedge F_2 \in [4,9] \quad \rightarrow \text{accept} \\
r_3: &\quad F_1 \in [5,9] \wedge F_2 \in [7,9] \quad \rightarrow \text{accept} \\
r_4: &\quad F_1 \in [5,9] \wedge F_2 \in [0,2] \quad \rightarrow \text{discard} \\
r_5: &\quad F_1 \in [0,9] \wedge F_2 \in [0,9] \quad \rightarrow \text{discard}
\end{aligned}
$$

Figure 3.19: Five rules in a simple packet classifier

Now we need to add three new rules: r_6, r_7 and r_8, and all other old rules shift down accordingly. The three new added rules are shown in Figure 3.20.

$$
\begin{aligned}
r_6: &\quad F_1 \in [3,4] \wedge F_2 \in [4,9] \quad \rightarrow \text{discard} \\
r_7: &\quad F_1 \in [5,7] \wedge F_2 \in [8,9] \quad \rightarrow \text{accept} \\
r_8: &\quad F_1 \in [5,9] \wedge F_2 \in [3,8] \quad \rightarrow \text{accept}
\end{aligned}
$$

Figure 3.20: Three added rules

When adding a new rule, we compare the first field of the new rule with existing entries one by one, and we only need to compare the next field if and only if there is overlapping between this field and they also have different decisions. There are four situations for each field comparison, which are shown in Figure 3.21.

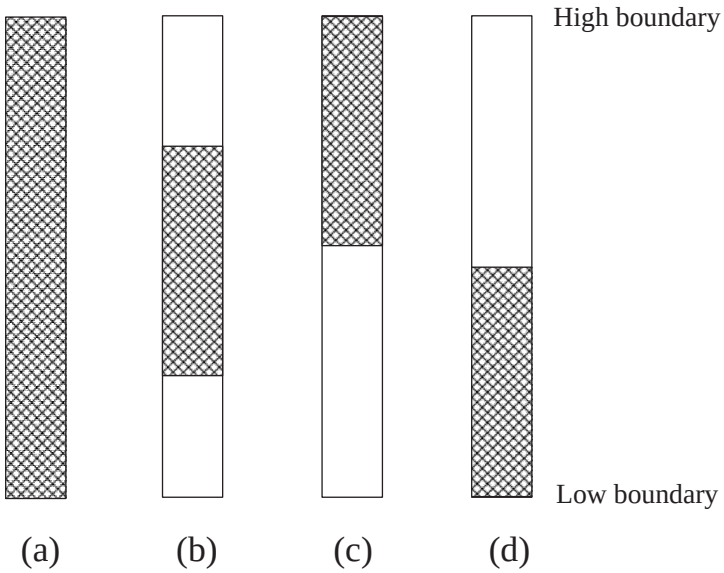


Figure 3.21: Four overlapping situations in a single field

The high boundary and low boundary of the field in the existing field is shown in Figure 3.21 and the shadow areas are overlapped ranges with the adding rule. Note that we only need to compare the rules with different decisions. So for the situation shown in Figure 3.21(a), just simply convert the decision and combine the new rule with nearby rules if they can. For the situation shown in Figure 3.21(b), convert the existing high boundary with new low boundary minus one, and generate two same new rules: one with the same range and the same decision with the new range, and another replace the low boundary with new high boundary plus one. For the situation shown in Figure 3.21(c), convert the existing high boundary with new low boundary minus one, and generate a same new rule: one with the same range and the same decision with the new range, then combine it with the following rule if they can. For the situation shown in Figure 3.21(d), convert the existing low boundary with new high boundary plus one, and generate a same new rule: one with the same range and the same decision with the new range, then combine it with the pre-

$$
\begin{aligned}
r'_1: &\quad F_1 \in [0,2] \rightarrow \text{accept} \\
r'_2: &\quad F_1 \in [3,4] \wedge F_2 \in [0,3] \rightarrow \text{discard} \\
r'_3: &\quad F_1 \in [3,4] \wedge F_2 \in [4,9] \rightarrow \text{accept} \\
r'_4: &\quad F_1 \in [5,9] \wedge F_2 \in [0,2] \rightarrow \text{discard} \\
r'_5: &\quad F_1 \in [5,9] \wedge F_2 \in [3,9] \rightarrow \text{accept}
\end{aligned}
$$

Figure 3.22: New rules after adding three rules

vious rule if they can. During deal with one field, make sure other fields are unchanged. In order to perform efficient new rule insertion, affected rules searching, we store the new non-redundancy rules sorted, and this task can be done after the TCAM update. The psudocode of adding a new rule is shown in the following algorithm.

So r_6 separates r'_1 into three parts, r_7 does not affect the new rules. r_8 separates r'_2 into two parts, and does not affect r'_3. After adding the new rules, we combine two continuous rules with the same decision and the same parent node. The final new rules are shown in Figure 3.22.

The removing rule scheme is usually more complex than the adding rule scheme, especially for the compressed rule sets. In order to remove an old rule, we must keep the information of every original rule. So our algorithm stores the original and new non-redundancy rule sets in the following format: each new rule does not only store the final decision but also the previous decisions which has been covered by other rules with higher priority. Each decision is presented by a single bit, and an extra bit "1" indicates the beginning of the decision chain, and the least significant bit . For example, "00101101" indicates the decision sequence is "0", "1", "1", "0" and "1", and the recent final decision is "1". When adding a new rule with a "discard" decision to this decision chain, the new decision chain becomes "01011010", and the final decision will be changed. When removing an original rule corresponding to the third decision, then the decision

chain becomes “00101010”, and the corresponding TCAMs do not need to be changed due to the final decision is unchanged. The decision chain becomes “00010101” when removing an original rule corresponding to the fifth (last) decision, and the corresponding TCAMs need to be changed. If the second decision is the same as the first one, we can remove the second decision record, because remove such rules does not affect the final decision, so the first decision and the second decision must be different.

And each original rule records the new rule number and corresponding bit in the decision chains. So the format is:

$rule$ (new rule number, bit number)

When removing an original rule, just update the corresponding new rules. And we also need to update the bit numbers in the original rules which have the same “new rule number”.

We take the same example in Fig. 3.19. In this example, we add the rules one by one, and the final original rules are shown in Figure 3.23 and the new rules before combination are shown in Figure 3.24.

Algorithm 9 Pseudocode of adding a new rule

```
1: i = 0;
2: n = 0;
3: while no overlap between new rule and rule n do
4:    n := n + 1;
5: end while
6: while overlap between new rule and rule n do
7:    for i := 2 to 5 do
8:       if Overlap in Figure 3.21(b) then
9:          insert two rules the same as rule n
10:         set rule n's high boundary to new rule's low boundary minus one
11:         set rule (n + 1)'s low boundary to new rule's low boundary
12:         set rule (n + 1)'s high boundary to new rule's high boundary
13:         set rule (n + 2)'s low boundary to new rule's high boundary plus one
14:         n := n + 1;
15:      else
16:         if Overlap in Figure 3.21(c) then
17:            insert one rule the same as rule n
18:            set rule n's high boundary to new rule's low boundary minus one
19:            set rule (n + 1)'s low boundary to new rule's low boundary
20:            set rule (n + 1)'s high boundary to new rule's high boundary
21:            n := n + 1;
22:         else
23:            if Overlap in Figure 3.21(d) then
24:               insert one rule the same as rule n
25:               set rule n's high boundary to new rule's low boundary
26:               set rule (n + 1)'s low boundary to new rule's high boundary plus one
27:            end if
28:         end if
29:      end if
30:   end for
31: end while
```

When we remove r_2, the r_2' needs to be modified. So the decision chain becomes "00000101", and we can combine r_1' and r_2'.

So our algorithm has three main components: original rule set, non-redundancy rule set and TCAM entries. And the architecture is shown in Figure 3.25.

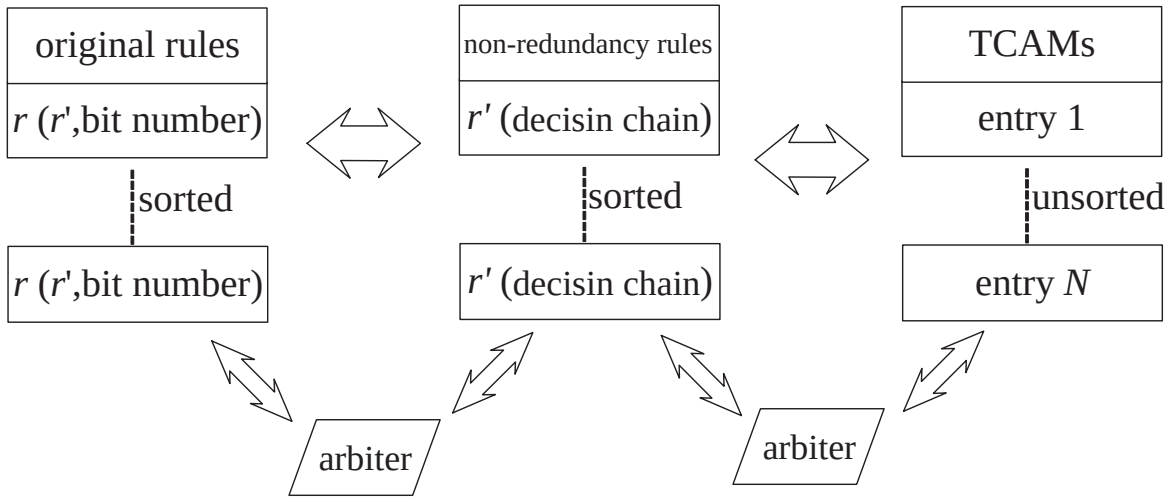


Figure 3.25: The architecture of our update algorithm

The last part configures the non-redundancy rules into TCAM entries, which has been well studied.

In order to reduce the TCAM entries consumption and eliminate the memory access of

$$
\begin{aligned}
r_1: &\quad F_1 \in [0,4] \wedge F_2 \in [0,9] &&\rightarrow \text{accept}(r_1', 0), (r_2', 0)\\
r_2: &\quad F_1 \in [0,4] \wedge F_2 \in [4,9] &&\rightarrow \text{accept}(r_2', 1)\\
r_3: &\quad F_1 \in [5,9] \wedge F_2 \in [7,9] &&\rightarrow \text{accept}(r_4', 0)\\
r_4: &\quad F_1 \in [5,9] \wedge F_2 \in [0,2] &&\rightarrow \text{discard}\\
r_5: &\quad F_1 \in [0,9] \wedge F_2 \in [0,9] &&\rightarrow \text{discard}
\end{aligned}
$$

Figure 3.23: Original rules with removing information

$$
\begin{aligned}
r'_1: &\quad F_1 \in [0,4] \wedge F_2 \in [0,3] &&\rightarrow \text{accept}(00000101)\\
r'_2: &\quad F_1 \in [0,4] \wedge F_2 \in [4,9] &&\rightarrow \text{accept}(00001011)\\
r'_3: &\quad F_1 \in [5,9] \wedge F_2 \in [0,6] &&\rightarrow \text{discard}(00000010)\\
r'_4: &\quad F_1 \in [5,9] \wedge F_2 \in [7,9] &&\rightarrow \text{accept}(00000101)
\end{aligned}
$$

Figure 3.24: New rules before combination

final result fetching, we only need to configure rules with "discard" decision into the TCAM entries because there is no overlap between rules with "accept" decision and rules with "discard" decision.

In order to enable the out of order storage in TCAMs, each rule records the TCAM entry numbers it has been stored, and the arbiter allocates and recycles TCAM entries as a Memory Manage Unite (MMU). In order to reduce the frequency of TCAM entry update, we use a valid mask to recycle TCAM entries, just set the corresponding bit in valid mask to "0" when we need to recycle a TCAM entry. So we only update TCAMs when we need to add or modify TCAM entries.

3.5.2 Experimental Results

The cost of adding a rule vary dramatically on the property of the new rule, in the best case, there is nothing needed to be done to the TCAM component. In this situation, the new added rule is redundant and it does not change any decision. In the second best case, we only need to modify the valid mask. In this situation, the new rule with an "accept" decision and no TCAM entries needs to be modified. In the normal case, we just do as discussed above. In the worst case, when the new rule with an "accept" decision, we need to modify two rules and remove some rules in the non-redundancy rule set, and the modified rule's corresponding TCAM entries need to be modified, and the removed rules' corresponding valid mask bits should be set; when the new rule with an

Table 3.7: The number of entries need to be updated for a new rule with an "accept" decision

Low\High boundary	Situation(a)	Situation(b)	Situation(c)	Situation(d)
Situation(a)	14.25	0	0	14.25
Situation(b)	0	14.25	0	0
Situation(c)	14.25	0	14.25	29
Situation(d)	0	0	0	14.25

Table 3.8: The number of entries need to be updated for a new rule with an "discard" decision

Low\High boundary	Situation(a)	Situation(b)	Situation(c)	Situation(d)
Situation(a)	14.25	0	0	14.25
Situation(b)	0	14.25	0	0
Situation(c)	14.25	0	14.25	14.25
Situation(d)	0	0	0	14.25

"discard" decision, we need to modify one rule and remove some rules in the non-redundancy rule set, and the the modified rule's corresponding TCAM entries need to be modified, and the removed rules' corresponding valid mask bits should be set.

For a single range, such as the destination port range, we assume every possible range has the same possibility 1 appear in the rule set, so the rule set contains ranges [0, 1], [0, 2], ..., [0, $2^n - 1$], [1, 2], [1, 3],...,[$2^n - 2$, $2^n - 1$] in average, where is the number of bits of this field. The simulation result shows 14.25 TCAM entries are needed to store a single port range. Because we observe that packet classifiers typically have at most one port range in each rule, and rules that specify two port ranges are very rare, so we assume that each rule in the non-redundancy rule set consumes 14.25 TCAM entries in average. The update cost of a new rule with an "accept" decision and an "discard" decision are shown in Table 3.7 and Table 3.8 respectively. We consider all the situations in Fig. 3.21 for new rule's low boundary and high boundary

So in average, 15.27 TCAM entries needed to be updated by adding a new rule, which does

not include the best and near best cases. And this number does not increase with the number of total TCAM entries used. For example, if we have 1024 TCAM entries, only 1.5% TCAM entries needed to be updated if needed for a new rule.

And how to locate the affected rules in the non-redundancy rule set by the new rule is simple, we just need to locate the two (or one) rules in the non-redundancy rule set corresponding to the high boundary and low boundary of the new rule, and the timing complexity is $O(N)$, where N is the number of rules in the non-redundancy rule set, which is much smaller than the number of TCAM entries used. The timing complexity of locating affected rules in the non-redundancy rule set is the same when removing a rule.

The TCAM entry update process of removing a rule is much more difficult to be evaluated than adding a rule due to the number of affected rules in the non-redundancy rule set is vary dramatically, removing of old rules is more likely to affect more rules in the non-redundancy rule set, then more TCAM entries are needed to be modified. In the best case, no TCAM entries and no valid mask need to be modified. In this situation, the removing rule is a redundant rule, it has the same decision with previous rule(s) or latter rule(s). In the second best case, no TCAM entries need to be modified and we only need to set the valid mask. In this situation, the removing rule must be with an "discard" decision, and all the corresponding rules in the non-redundancy rule set must followed by rules with "accept" decision. In the worst case, $M - 1$ TCAM entries and valid mask need to be modified, where M is the number of TCAM entries used after removing the rule. In average, $(M - 1)/2$ TCAM entries need to be modified, fortunately, removing rules is rare.

Many algorithms have been proposed to reduce the TCAM entries consumption in packet classifiers and usually can achieve 40% to 60% compression ratio. These approaches store sorted compressed rules in TCAM entries, so the update process needs to modify almost all used TCAM entries. We compare our algorithm with both original method without compression and a com-

monly used compression approach which can achieve 60% compression ratio. The comparisons of average numbers of updated TCAM entries when adding a new rule or removing a rule are shown in Fig. 3.26 and Fig. 3.27. When removing a rule, our algorithm consumes approximate 29% and 36% TCAM entries update compared with original approach without compression and existing compression approach. From both figures, we can see that our algorithm performs much better than existing approaches during the update process.

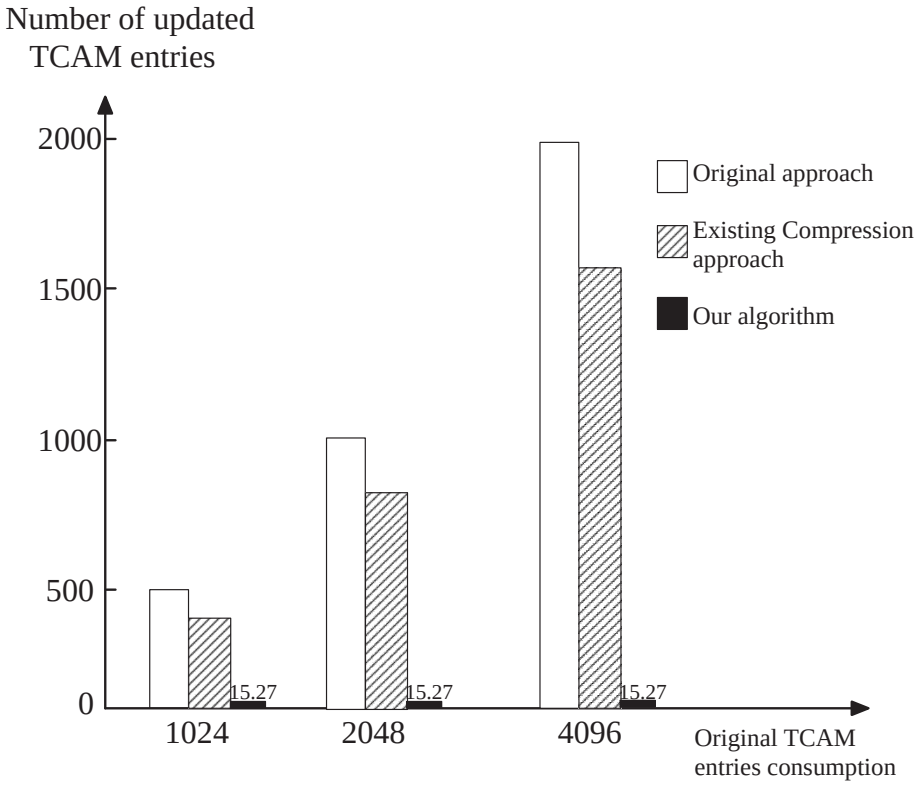


Figure 3.26: Comparisons of average numbers of updated TCAM entries when adding a new rule

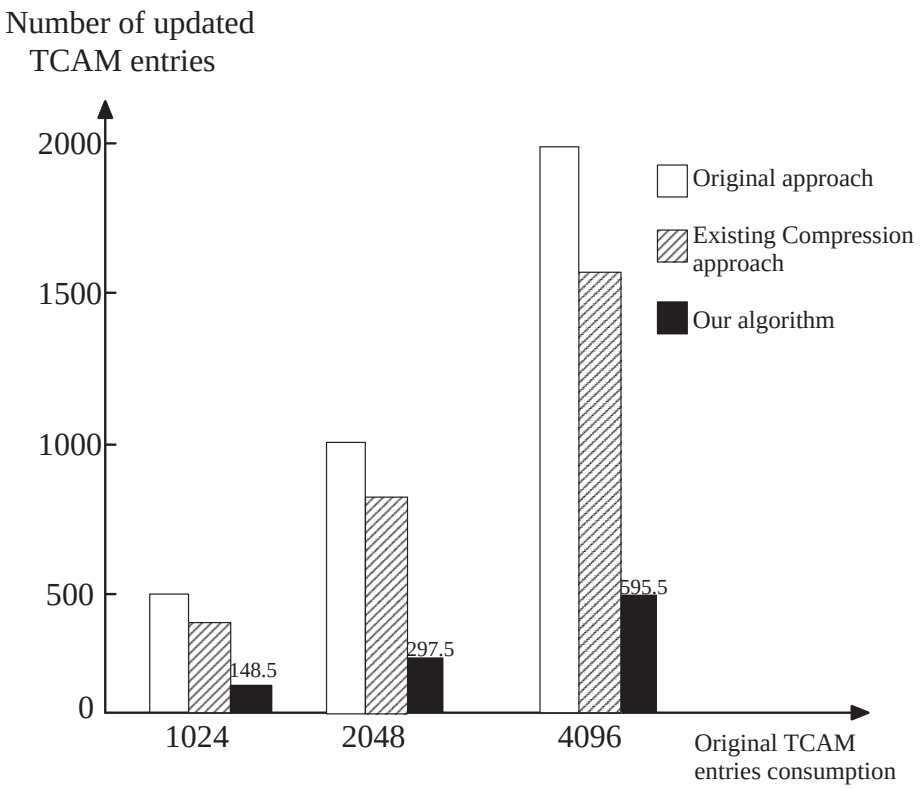


Figure 3.27: Comparisons of average numbers of updated TCAM entries when removing a rule

When adding a new rule, we need to find the two affected boundary rules in the non-redundancy rule set, and we only need to modify the rules between them. The complexity of the search is $O(2\log N)$, where N is the number of rules in the non-redundancy rule set. And the complexity of compression of the non-redundancy rule set is $O(N)$, and the complexity of converting them into TCAM entries really depends on the property of the new rule.

3.5.3 Summary

In this section, we propose a fast TCAM update scheme which enables out of order storage in the TCAM and reduces the TCAM entries usage. In order to optimize the update process, we need to

reduce both TCAM entries usage and TCAM entries shift during update process. So we remove the redundancy in the rule set, which not only reduce the TCAM entries usage but also enable the out of order storage in the TCAM due to at most one match during all the TCAM entries. According to our experiments using randomly selected rules from actual packet classifiers, it shows a reduction of 70.9% in the number of TCAM entries after removing the redundancy in the rule set, and only 15.27 TCAM entries are needed to be updated by adding a new rule in average, and our algorithm consumes approximately 29% and 36% TCAM entries update of original method and existing compression method when removing a rule.

3.6 Comparator CAMs for Packet Classification

Approaches we proposed above do not modify the TCAM entry circuit itself to achieve TCAM consumption reduction. In this section, we propose Comparator Content Addressable Memory (CCAM), which is modified TCAM and can store ranges in CCAM entries besides ternary state bits. In our algorithm, the hardware can be configured to store all kinds of port number ranges. If a rule contains any two-direction ranges, only two CCAM entries are needed to store it, otherwise, each rule can be stored in a single CCAM entry. Our simulations show that CCAMs consume about 27.6% entries compared with original TCAMs. Considering the transistor consumption, our approach saved about 66.4% transistors. Furthermore, the update process of CCAMs can be as fast as that of TCAMs usage without any compressions.

3.6.1 Proposed Comparator CAMs Architecture

In order to reduce the TCAM entry consumption caused by the range expansion problem and avoid the compression of rules for fast update, we propose the Comparator Content Addressable Memory (CCAM) architecture. We first analyze ranges in the port number fields in rules, and group them into four categories. Based on this categorization, we propose a CAM entry configurable to accommodate all four kinds of ranges. We first define two kinds of ranges in port numbers:

- *Bounded* range: a range covers at least two port numbers, but it does not cover port number 0 or port number 65535. For example, range "$700 : 900$."
- *Half-bounded* range: a range covers at least two port numbers, and it covers either port number 0 or port number 65535. For example, range "< 1024."

If a rule contains any *bounded* range, two CCAM entries are needed to store it; otherwise, the rule can be stored in a single CCAM entry. In order to increase the throughput of packet classification, we pipeline the whole procedure into four stages.

Unlike previous approaches designed for ranges, our approach embeds the range comparators within CAM entries and provides an efficient and flexible circuit design to store different types of ranges. In addition, our approach does not require any rule set compressions before storing them, which is crucial for fast rule set updates.

the Comparator Content Addressable Memory (CCAM) architecture. It adds the comparison fields to TCAMs to store port ranges. We store Protocol, Source IP, and Destination IP in original TCAM cells, and store Source Port and Destination Port in CCAM cells. In the remainder of the paper, we refer to this combination of TCAM and CCAM cells as a *combined CCAM entry.*

Every new combined CCAM entry gets the match result "1" if and only if the Protocol, Source IP Address, and Destination IP Address fields of an input key match the corresponding

fields in TCAM cells, and both the Source Port and Destination Port fields of the input key match the corresponding ranges in CCAM cells. We use pipelining to increase the throughput and clock speed. In our pipelined architecture, the first stage processes 72 bits (Protocol, Source IP Address, and Destination IP Address) with the ordinary TCAM circuit, the second stage compares the Source Port, and finally the third stage compares the Destination Port.

Below we discuss the comparison parts. The proposed comparator logic implements a "$\geq$" comparison as shown in Figure 3.28.

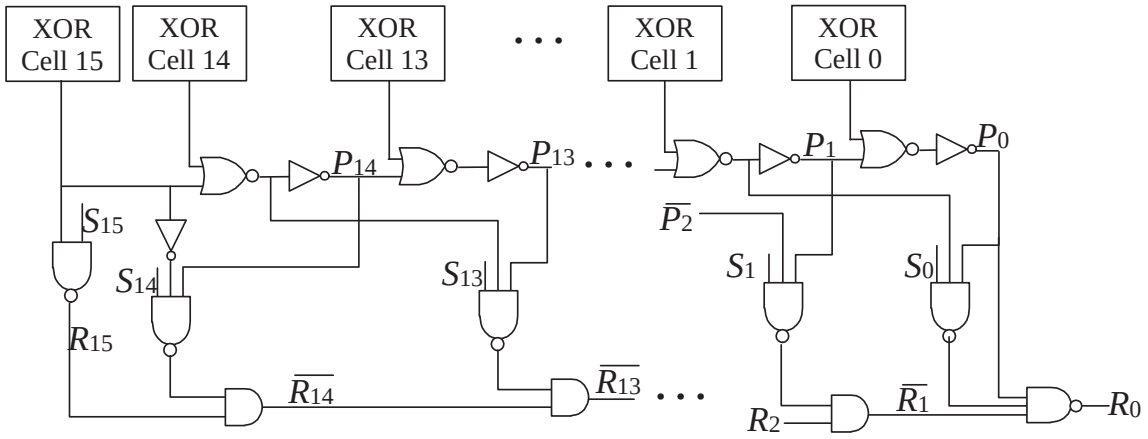


Figure 3.28: Proposed range comparator of CCAM architectures

In a range comparison, S_i denotes the ith stored bit and Q_i the ith query bit. The logic

functions in Figure 3.28 are equivalent to the following operations:

$$
\begin{aligned}
P_{15} &= S_{15} \oplus Q_{15} \\
P_i &= S_i \oplus Q_i \vee P_{i+1} && 0 \leq i \leq 14 \\
R_{15} &= P_{15} \wedge S_{15} \\
R_i &= R_{i+1} \vee (\neg P_{i+1} \wedge P_i \wedge S_i) && 1 \leq i \leq 14 \\
R_0 &= R_1 \vee (\neg P_1 \wedge P_0 \wedge S_0) \vee \neg P_0
\end{aligned}
$$

where P_i represents whether there is a different value pair of stored bit and query bit in the ith bit or higher bit. And R_i represents whether current query data is larger or equal to current stored date regardless of the values in bits smaller than ith bit, then R_0 indicates the final result of the comparison in this entry.

A range with both lower and upper bounds needs two CCAM entries to store them separately. That means each rule should consume two CCAM entries because we do not know which rules contain ranges before storing them. That may cause a lot of waste because many of the rules can be stored in a single entry without any range expansion. Hence, we design our logic to be more programmable to reduce such waste. First, we divide port ranges into four categories: exact numbers, wild-cards (*), half-bounded ranges, and bounded ranges. Note that only those in the last category need two CCAM entries. Then, we make the CCAM entries configurable so that they can perform the "$=$", "$\geq$", and "$\leq$" operations. In addition, each CCAM entry has three different usages: as an independent entry for a half-bounded range, and a lower bound and an upper bound of a bounded range. Figure 3.28 shows the "$\geq$" operation, and the "=" and "$\leq$" operations can be

implemented by modifying R_0 as follows:

$$\begin{aligned} R_0 &= \neg P_0 && \text{for } = \\ R_0 &= \neg(R_1 \vee (\neg P_1 \wedge P_0 \wedge S_0)) && \text{for } \leq . \end{aligned}$$

Therefore, only a small part of the logic circuit needs to be configurable. To make two adjacent CCAM entries a pair to store a bounded range, we configure the usage type of CCAM entries to be a lower bound and a upper bound. Usually, the rules listed above in a rule set have higher priority than the rules listed below. Given multiple matching entries, i.e., those with all 1's as the results, the priority encoder chooses the matching entry with the highest priority. When storing a bounded range, we store the upper bound in the CCAM entry above the one with the lower bound, and set the bounded range hint bit in the upper entry.

To implement this, every 16-bit comparison logic requires three extra bits: T, the hint bit, indicating whether the current and next CCAM entries contain a bounded range, and C indicating the operation type, `00` for "$=$", `10` for "$\geq$", and `01` for "$\leq$". The logic circuit of the kth CCAM entry output is shown in Figure 3.29.

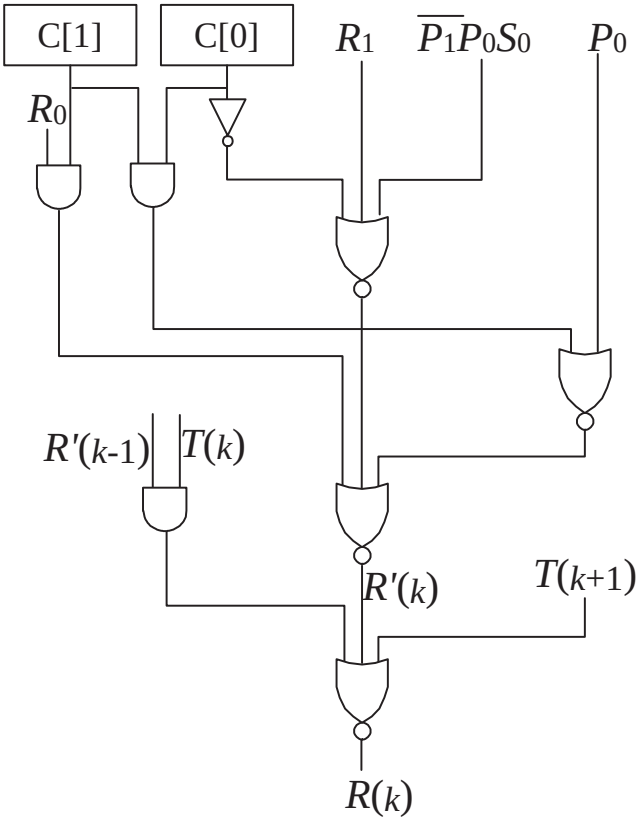


Figure 3.29: The output logic of CCAM entries

In Figure 3.29, $R(k)$ is the final output of the kth entry. And we can see that the final result is affected by not only the upper CCAM entry ($T(k+1)$) but also the lower CCAM entry ($R'(k-1)$). If the upper entry is the upper bound of a bounded range ($T(k+1) = 1$), the output of this CCAM entry must be set to 0 to ensures at most one "1" output for each rule. If the current CCAM entry is the upper bound of a bounded range ($T(k) = 1$), the range matches only when both current and the lower entries match the query key. $R'(k)$ is the inverse of the current CCAM entry match result.

Then, among all outputs with $R = 1$, the priority encoder chooses the highest-priority

matching entry, and fetches the corresponding action defined in the packet classifier.

In order to increase the throughput of packet classification, we pipeline this process as shown in Figure 3.30.

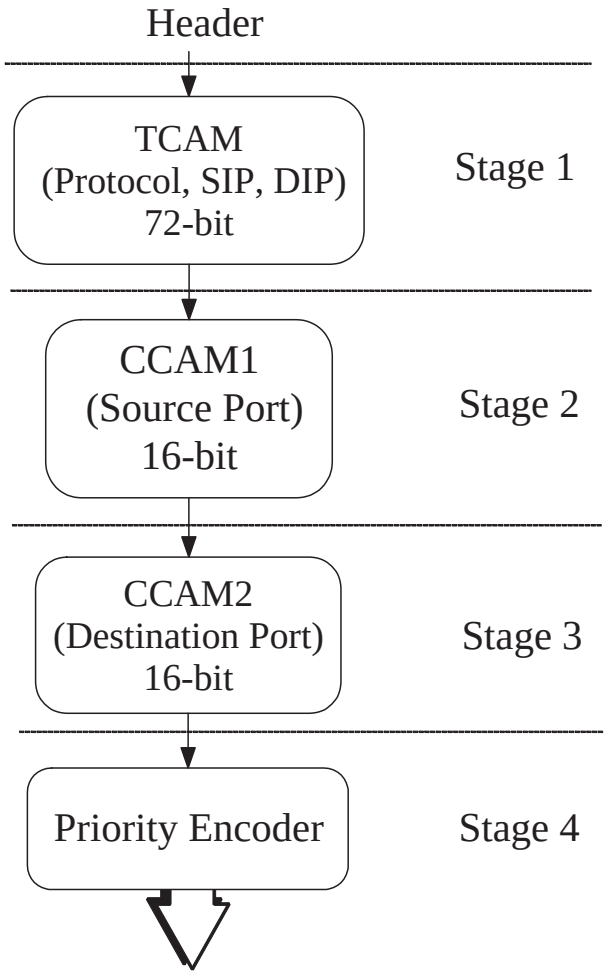


Figure 3.30: Pipelined Architecture of separated fields

3.6.2 Simulation Results

First, we compare the number of transistors consumed by an ordinary TCAM entry with that by a proposed CCAM entry. In our approach, each CCAM cell needs 24 transistors except for the left-most bit, which needs 18 transistors. and a TCAM cell needs 16 transistors. Each output logic requires 60 additional transistors in CCAMs. The details are shown in Table 3.9.

Table 3.9: Number of transistors used by a TCAM entry and a CCAM entry

Type	Ordinary Entry	Proposed Entry
TCAM Cell	16×104	16×72
CCAM Cell	0	$24 \times 30 + 18 \times 2$
Output Logic	0	60×2
Total	1664	2028

Although a CCAM entry consumes 1.22 times as many transistors as a TCAM entry does, the overall number of transistors may be different because the CCAM-based approach consumes fewer entries than TCAM-based approaches because of TCAM-based ones' range expansion problem.

To compare the number of CAM entires between our approach and TCAM-based ones, we used rules generated with ClassBench [78], a widely-used packet classification benchmark, and rules from real-world rule sets. We randomly selected one thousand rules among them, and computed the number of CAM entries used by each approach. The simulation results are shown in Table 3.10.

From the numbers in Table 3.10, we can see that our approach consumes about 27.6% entries of what the TCAM-based approach does. Combining this with the results in Table 3.9, we find that our approach saves 66.4% of transistors.

Compared with other approaches also modifying TCAM entries, our approach consumes fewer entries and achieves lower latency. For example, DRES requires 1.23 entries per rule, while

Table 3.10: Entry consumption by TCAM and CCAM

Range Type	Percentage (%)	TCAM Entry #	CCAM Entry #
Exact number	32.5	325	325
Wild-Card	48.4	484	484
Half-bounded	13.8	1623	138
Bounded	5.3	1389	106
Total	100	3821	1053

our approach requires only 1.05 entries per rule. The simulation on a Xilinx Virtex-5 FPGA shows that the clock speed can also increase by forty percent.

RMB is another hardware-based approach, where ranges in Port number fields are stored in Range Match Blocks and compared, while other fields are compared using an ordinary TCAM chip. The results are combined together to get the final verdict. However, each cell in RMB needs two comparison logics for every range while our approach needs only one for half-bounded ranges. Our approach also simplifies the logic further with a minimal configurable part. Moreover, since our approach matches all the five fields simultaneously in CCAM entries, it eliminates the latency between chips. Besides, our approach uses only one priority encoder instead of two with RMB.

Software-based approaches typically achieve a 40% to 60% compression ratio, which is close to what our approach does. However, the compression itself is complex and time-consuming, making the update process slow.

3.6.3 Summary

In this paper, we proposed energy efficient Comparator Content Addressable Memories (CCAMs) to solve the range expansion problem and reduce the rule set update delay. Our approach can store ranges in CCAM entries besides ternary state bits. In our algorithm, the hardware can be configured to store all kinds of port ranges. If a rule contains any bounded range, two CCAM

entries are needed to store it; otherwise, the rule can be stored in a single CCAM entry. Simulations show that CCAMs consume about 27.6% of what the original TCAMs do in terms of the number of entries. As for transistor consumption, our approach saves about 66.4% of transistors compared with TCAM-based approaches, and this reduces power consumption significantly. Furthermore, the rule update process with CCAMs is comparable to that with TCAMs without any rule set compression. Compared with the existing approaches optimized for ranges, our approach performs better because of its fast update, compact circuit logic, and pipelined architecture.

CHAPTER four

DEEP PACKET INSPECTION

4.1 Introduction

Many security applications in today's networks are based on deep packet inspection. These applications, such as traffic monitoring, layer-7, and network intrusion detection compare the headers and payloads of data packets to predefined databases describing potential attack traffic. These predefined databases are populated by numerous string patterns that, when found, imply that the packet under question is possibly malicious. The inspection is performed using exact matching algorithms. However, the number of patterns has continued to grow in order to describe more and more payloads. For example, Snort [79], an open-source Network Intrusion Detection System (NIDS), has seen its rule set double in size in the last six years from roughly 3,000 rules to more than 5,000. On a similar note, the signature database for the open source ClamAV antivirus software had about 27,000 patterns in 2010 [80]. Further, the volume of network traffic is continuing to increase such that open-source NIDSs like Bro [81] and Snort [79] expend all the resources, both CPU time and memory, and halt immediately when they are deployed under high-speed network environments [82]. Thus, efficiency in matching, especially for large pattern sets, has become a major concern. These regular expressions are also used in commercial firewalls and other networking equipments including Cisco's IOS [83].

The most popular method to implement pattern matching is to use finite automata [84–87]. A finite automaton is built as a composite of all the patterns in the rule set and is run with the packet payload as input. The exact automaton used is dependent on implementation considerations and

may either be Deterministic Finite Automata (DFA) or Non-deterministic Finite Automata (NFA). NFA offer the best efficiency in terms of memory required to store the automaton, however NFA can have multiple transitions per state and thus may need to maintain multiple states as the automaton is traversed resulting in less efficiency in throughput. This makes NFA more suitable for Application-Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) implementations which can provide wide bandwidth but small amount of on-chip memory. Conversely, DFA tend to be quite large in terms of memory, but only have a single transition between states. Therefore, DFA are more suitable for general-purpose processors and network processors.

The remainder of the chapter is organized as follows. In Section 4.2, related work in regular expression matching is presented. Section 4.3.1 describes our hierarchical NFA-based pattern matching, Section 4.4.2 and Section 4.5.2 describe our hybrid regular expression matching for deep packet inspection on multi-core architecture and DFA-based regular expression matching on compressed traffic respectively.

4.2 Related Work

Nowadays, regular expressions are the language of choice in NIDS (network intrusion detection systems) from commercial vendors such as 3Com's TippingPoint X505 [88] and Cisco IPS [83], as well as open source NIDS such as Bro [81] and Snort [89]. Those regular expressions are typically implemented using finite automata, either NFA or DFA. Because a DFA requires at most one state transition per input character, it is faster than a NFA in many cases, and thus has been a preferred way to implement regular expression matching. For the thousands of complex regular expressions as found in Snort's rule sets, the DFA implementation consumes prohibitive amount of memory. We divided the approaches to regular expression matching into the following three categories:

ASIC-based Several commercial network equipment vendors, including 3Com [88] and Cisco [83] have supplied their own NIDS and a number of smaller players have introduced pattern matching ASICs which go inside these NIDS. ReCPU is a fast ASIC-based regular expression matching approach [84]. In fact, many have argued that peep packet inspection should happen in ASICs. Developing ASICs for NIDS, however, has several disadvantages; it requires a large investment and a long development cycle, and it hard to upgrade.

FPGA-based There is a body of literature advocating FPGA-based pattern matching [90–99]. It can provide not only fast matching cycle but also parallel matching operations. NFAs are well-suited for FPGA-based matching because of its wide bandwidth requirement and low memory consumption. Pre-decoded CAMs [97] and Bitwise optimized CAM [98] are FPGA-based architectures that use character pre-decoding coupled with CAM-based patterns to accelerate the matching speed. This type of approaches, however, is not flexible enough for general-purpose regular expression matching. Besides, it is still expensive and power-consuming.

Software-based The software-based approaches are also called general-purpose approaches, and they are based on general-purpose processors or network processors [85–87, 100–107]. Our work falls into this category. DFAs are more popular in software-based approaches because they only need one state transition per input character, which causes at most one memory access for each character input. Therefore, they are often desirable at high network link rates. However, As we mentioned earlier, the practical use of DFAs is limited because of their excessive memory usage. In order to mitigate this issue, many methods have been proposed [101–105, 108]. They develop several memory compression techniques for DFAs, focusing on reducing the number of transitions between states, and in some cases, 99% transitions can be eliminated. Although this can reduce

the memory consumption significantly, unfortunately it is hard to reduce the number of states in DFAs with complex regular expressions.

Ternary Content Addressable Memory (TCAM) based TCAM has become a popular approach as exhibited in recent research like [80, 109, 110]. Both [109] and [110] used TCAMs to store the transition rule table but even with transition compression approaches they still consumed a large amount of TCAM and SRAM resources. The research in [80] stored states instead of transitions in TCAMs to reduce the usage of both TCAM and SRAM, and their simulation results showed a significant improvement because there are usually many fewer states than transitions in a DFA. Unfortunately, the computational complexity of the approach is very high. We note that the high speed of TCAM is often offset by not only the cost of the TCAM, but also a slow clock speed and slow memory accesses. Thus, rather than using TCAM, we see BCAM as an attractive alternative to not only reduce cost, but to increase the clock speed. Finally, our approach eliminates the usage of SRAM and thus removes that as a factor in pattern matching.

Actually, all the previous works focus on how to use NFA/DFA more efficiently, without considering compressed traffic. Bremler-Barr and Koral focuses on this issue but only considers simple fixed-length pattern matching [111]. We further explore efficient algorithms to perform deep packet inspection on compressed traffic based on regular expressions, which are more complex and commonly used in today's Intrusion Detection Systems.

4.3 Hierarchical NFA-Based Pattern Matching for Deep Packet Inspection

Many security applications in today's networks are based on deep packet inspection, examining not only the headers but also the payloads of data packets. Traffic monitoring, layer-7

filtering, and network intrusion detection classify traffic by identifying predefined patterns within packet payloads that are specific to certain classes of attacks. Pattern matching is the primary task in deep packet inspection. The most common and efficient implementations of pattern matching are based on Non-deterministic Finite Automata (NFA) and Deterministic Finite Automata (DFA). In this section, we propose an efficient NFA-based pattern matching in Binary Content Addressable Memory (BCAM) which uses multi-bit, binary search words. Our approach can process multiple characters at a time using limited BCAM entries, providing for greater parallelism and potential scalability through examining larger numbers of characters per cycle. Furthermore, we build a hierarchical pattern matching architecture which filters most of the packets from full evaluation using a small number of BCAMs and leaving only a minor percentage of packets to be checked in the full pattern matching process. Such filtering greatly improves throughput for expected traffic as demonstrated in our simulations. We evaluate our algorithm using patterns provided by Snort, a popular open-source intrusion detection system. The simulation results show that our approach outperforms existing TCAM-based and software-based approaches.

4.3.1 Proposed Approach

Ternary Content Addressable Memories (TCAM) have been widely adopted by network applications, such as routers, to improve the speed of the longest prefix matching. Deep packet inspection systems have begun to use TCAM to perform pattern matching. TCAM matching is extremely fast because it allows the input key to compare against all patterns in parallel and return a result in a single clock cycle. Unfortunately, TCAM suffer from high circuit complexity which results in high cost and increased power consumption. The increase in power consumption has become a major concern as the cost of electricity continues to rise. Thus, Binary Content Addressable Memories (BCAM) are offered as an alternative to TCAM solutions. BCAM require fewer transistors and

exhibit less complex circuitry which translates into reduced power consumption. This reduction, however, comes at the cost of the "don't care" state that is present in TCAM that allows a comparison to match either a 0 or a 1. The absence of this state is problematic in pattern matching as the "don't care" state is common to many patterns.

Our approach is threefold. First, we reduce the complexity of each storage cell so that BCAM may be employed rather than TCAM. Secondly, we utilize NFA rather than DFA in order to reduce the number of required entries stored in the BCAM. Finally, we implement a scalable, parallel processing architecture in order to achieve high throughput. Under our approach, it becomes possible to implement high throughput pattern matching in BCAM and garner the benefits of BCAM's reduced cost and power consumption.

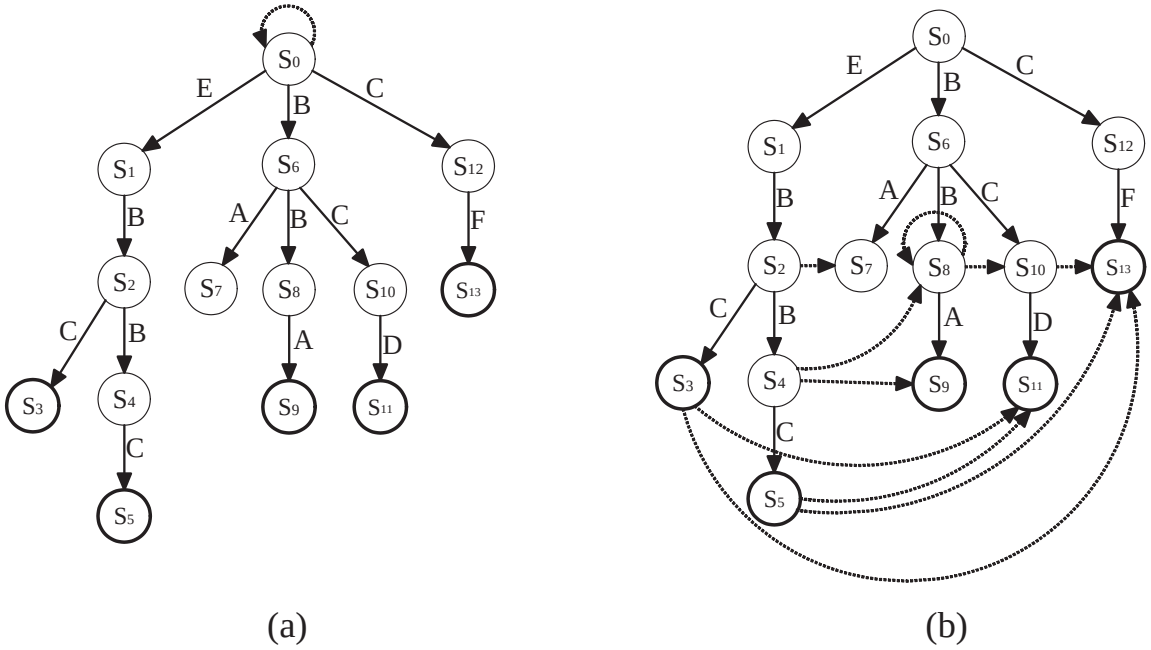


Figure 4.1: NFA in (a) and DFA in (b) for the patterns {EBC,EBBC,BA,BBA,BCD,CF}. Failure and restartable transitions are omitted for clarity in (b)

(1). Proposed Pattern Matching

As stated earlier, most pattern matching solutions utilize NFA or DFA. We observe that for any single pattern an input string must match every character in that pattern in order to register a match. From this, we conclude that at any point in matching the current match state is based on the previous match state and the current matching result. Thus, it is possible to chain matching decisions and that an input string will not match unless the chain is followed to the final state. To illustrate we employ the same example finite automata as in [80] utilizing a simple pattern set of: CF, BCD, BBA, BA, EBBC, and EBC. The corresponding NFA and DFA for this set are shown in Figure 4.1. We note that NFA use ε transitions to represent the possibility of moving to a new state without consuming any input character. This can result in multiple active states (concurrent traversals of the NFA) while processing an input string. It also means that if an input string fails to match, then that active state halts. Thus, if the chain of matching characters is broken, then traversal is restarted at the beginning of the NFA. We utilize this aspect, and the following two properties to simplify our matching circuitry:

- Only the start state of the NFA has incoming or outgoing ε transitions.
- Each state of the NFA has only one incoming transition.

Figure 4.1(a) illustrates an NFA created under the above constraints. The DFA is shown to the right in Figure 4.1(b), and most of the transitions are omitted for clarity here. We can see that both the NFA and DFA have 14 states but the DFA has many more transitions. Furthermore, every state in the NFA has only one incoming transition except for the initial state.

The final piece to our approach concerns BCAM memory. A single BCAM entry may store some number of BCAM cells each containing a 0 or 1. A TCAM entry, however, contains a third, "don't care," state that matches both 0 and 1. To support this a TCAM entry stores content as a (value, mask) pair, where value and mask are W-bit numbers, requiring W storage cells for

the value and an additional W storage cells for the mask. Moreover, the matching circuitry is more complicated than that of a BCAM entry. A typical TCAM cell requires two SRAM cells plus circuitry for the matching logic. Each SRAM cell typically requires six transistors and the matching logic requires 4 transistors. Thus, a single TCAM cell is 2.7 times larger than the typical SRAM cell, though different techniques used by CAM manufactures can result in different sizes for SRAM cells [31]. Conversely, a single BCAM cell requires only a single SRAM cell and simple matching logic. As a result, we assume that the number of transistors and power consumption of a TCAM cell are roughly two times that of a BCAM cell.

The basic logic of the proposed approach is shown in Figure 4.2. We assume there are n patterns and the sum of all their lengths, in characters, is N. Each BCAM entry stores the active state of the previous BCAM (labeled V_i in Figure 4.2) as well as the next corresponding character from the composite of all patterns (labeled C_i Figure 4.2). For the first BCAM entry, the active state will always match due to the epsilon transition in the NFA. From the second BCAM entry on, the active states depend on the matching result of the previous BCAM entry from the previous clock cycle combined with the matching result of the current BCAM entry in the current clock cycle. A complete match occurs after the final state for a pattern has successfully matched, implying that all previous states also matched as illustrated by r_1 in Figure 4.2.

The real architecture of the proposed approach is shown in Figure 4.3. There are still n patterns and the sum of all their lengths, in characters, is N. Each BCAM entry stores a character from the patterns, in sequence, so that all the BCAM entries are 8 bits. Registers here are used to record the current matching result for usage in the next clock cycle, and the "AND" logics are used to determine when both previous and current matches happen. For example, imagine that an incoming packet matches the first pattern r_1. In this example, C_0 matches in the first clock cycle and a "1" is stored in the first register. In the next clock cycle, a logical "AND" is performed

on the result from the first clock cycle, stored in the first register, with the output of C_1. That result is stored in the second register. This continues with the result of each stage stored in its respective register. At C_3, the final character for r_1 is matched, and since all successive characters have matched, the last register (the third in this case) contains a "1" so when a logical "AND" is applied to the match result for C_3 and the previous register a "1" is returned indicating a complete match to r_1. Further, since the last stage has matched a complete pattern there is no need to store the result in a register. Likewise, a failure to match at any stage will of course make matching the complete pattern impossible.

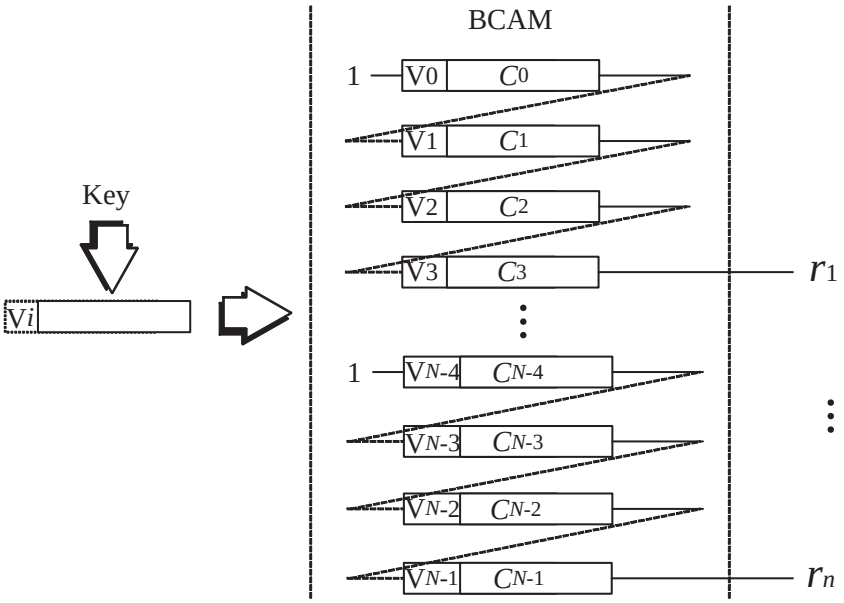


Figure 4.2: The logic of proposed approach based on BCAM

In order to increase the throughput, we designed our approach to process multiple characters at a time. In this section we process four characters at a time though the architecture can easily be extended to process more characters at a time. The architecture of the example is shown in Figure 4.4. In this example, we consider the pattern "ABCD" and the input string "CABCDFE".

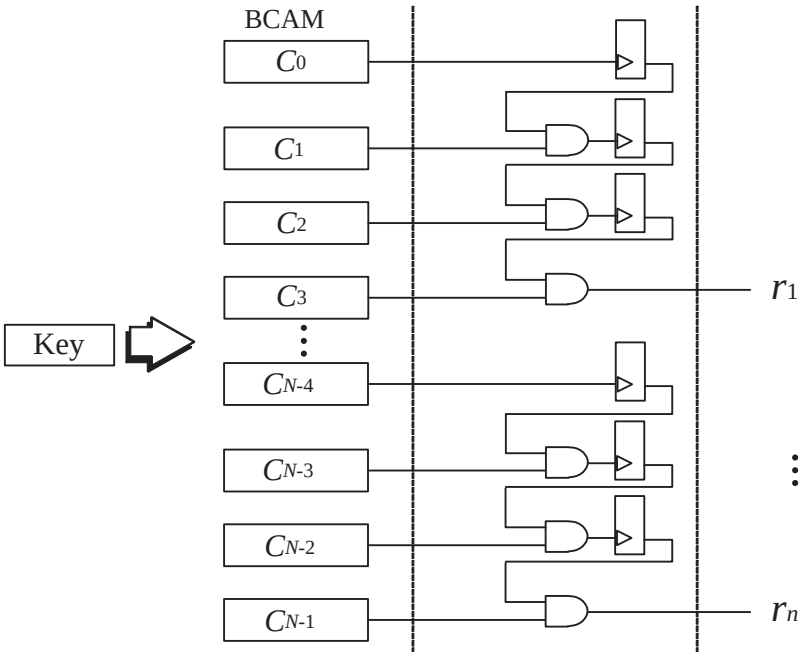


Figure 4.3: The main architecture of proposed approach based on BCAM

In order to process four characters in a single clock cycle, we need to consider four beginning points in the incoming string, so we need four sets of patterns for “ABCD” to be stored in the BCAM entries. The dashed lines in Figure 4.4 show the comparison relationships between incoming characters and BCAM entries. The four “AND” logics mean the incoming string must match all characters in the pattern in a single clock cycle, and the “OR” logic means the pattern can appear anywhere in the incoming string. The incoming string must shift four characters every time, but all seven characters in the incoming string must be examined. So in the example, the second set of BCAMs store pattern “ABCD” and match the incoming string. Further, the four BCAM entries in the same column in Figure 4.4 can be combined together to save hardware logic as they store the same content. The combined architecture is shown in Figure 4.5 illustrating how the BCAM entries with the same content can share data storage and utilize their own comparator logic.

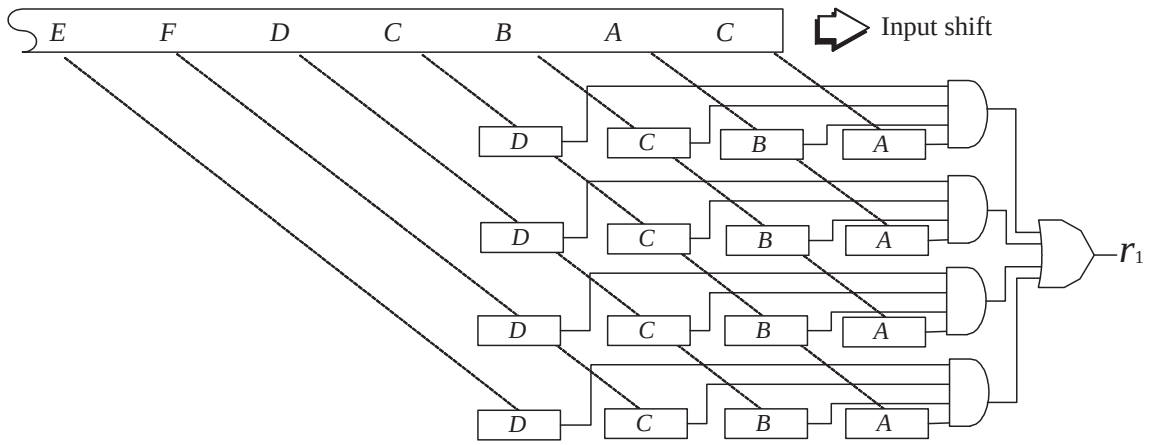


Figure 4.4: Proposed parallel processing unit for pattern matching

(2). Hierarchical Pattern Matching

In our simulations, we observe that most of the packets do not contain any matching string patterns nor do they contain any matching substrings of the rule set. Furthermore, many of the string patterns in the rule set contain common substrings. Based on these observations, we propose a hierarchical architecture and use a substring set of the rule set patterns as a pre-filter of the string pattern matching. We select all two-character substrings within all the patterns and store multiple sets of the whole substrings in BCAMs of 16-bit length. Multiple packets can be checked in parallel against the substrings, the output is "1" if current continuous two characters in the packet payload matches any substring and "0" if no match found, only matched substrings in the packet payload need to be checked against string patterns and the whole packet payload doesn't need to be checked against string patterns if there is no match in the first stage. Another benefit of this approach is that if the length of continuous "1" in the output is n, we only need to check the patterns of length no shorter than n, then we can divide patterns into multiple blocks to perform pattern matching in parallel. However, this could result in poor load balancing and the short pattern matching may become the bottleneck because the short patterns will be checked more often.

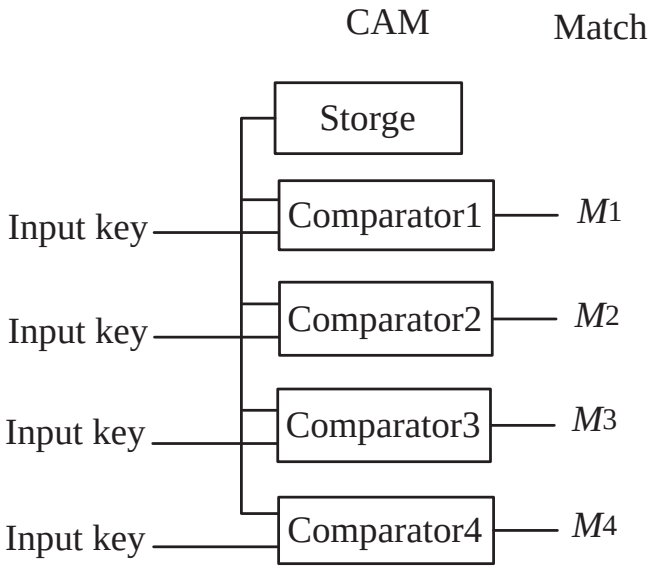


Figure 4.5: Combine multiple BCAM entries with the same content

Fortunately, we can build multiple short pattern blocks to work in parallel because they consume far fewer resources than long patterns. The hierarchical architecture is shown in Figure 4.6. We use 32 sets of substrings in the first stage in this section.

4.3.2 Evaluation

To evaluate the proposed approach, we collected the pattern sets from Snort [79] and ClamAV [112] which are the same as used in [80] for comparison. We build a single NFA, for each set, with only prefix merging because we need to make sure each state in the NFA only has one incoming transition; excepting the beginning state. We implement the NFA in NetFPGA [34], which is a network hardware accelerator that augments network functions of a standard computer. We use the Xilinx Virtex-II Pro FPGA on the NetFPGA and implement our algorithm on it for the simulation. The NetFPGA card has four Gigabit Ethernet ports, SRAM and DRAM chips on board, and the NetFPGA communicates with the host PC through a Peripheral Communication Interconnect (PCI)

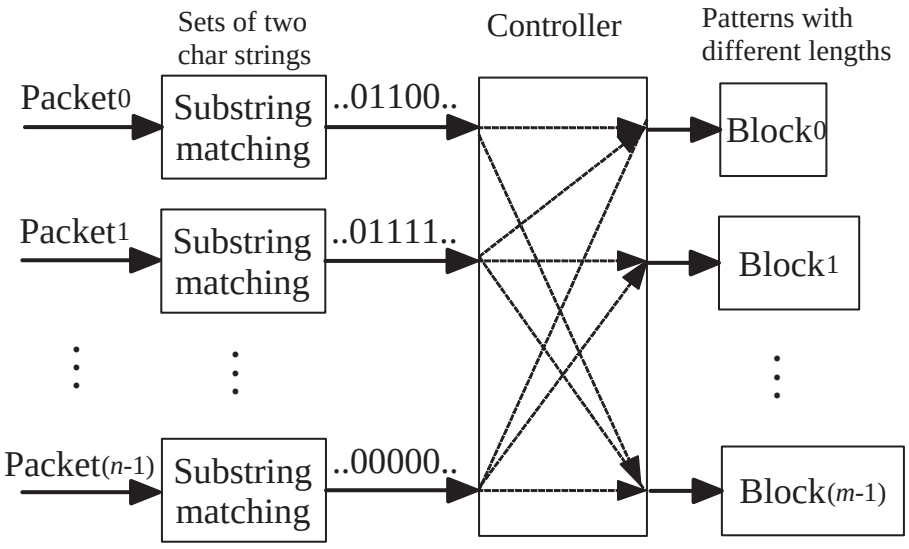


Figure 4.6: Proposed hierarchical pattern matching architecture

Table 4.1: Statistics of the patterns collected from Snort and ClamAV

Pattern Set	Approaches	Patterns	States in NFA	States in DFA
Snort	In [80]	6,423	-	75,256
Snort	Our approach	6,423	75,256	-
ClamAV	In [80]	26,987	-	1,565,874
ClamAV	Our approach	26,987	1,565,874	-

bus. All the BCAM entries are 8-bit wide and each character is stored in a BCAM entry. So the number of BCAM entries used is the number of states in the NFA, as well as total number of characters in patterns when only counting common prefixes once.

We compare our approach with the latest and the most efficient TCAM-based pattern matching approach "CompactDFA" [80] to the best of our knowledge. The rule sets collected from Snort and ClamAV are shown in Table 4.1, and simulation results are shown in Table 4.2.

From the simulation results we can see our algorithm outperforms "CompactDFA" proposed in [80] in almost all aspects. For power consumption by CAM, as we mentioned the power

Table 4.2: Comparison between CompactDFA and Hierarchical NFA

Items	Pattern set	CompactDFA	Hierarchical NFA
CAM type	Both	TCAM	BCAM
CAM length	Both	36-bits	8-bit
CAM Size (MB)	Snort	0.36	0.08
CAM Size (MB)	ClamAV	8.18	1.57
SRAM Size (MB)	Snort	0.32	0.00
SRAM Size (MB)	ClamAV	7.37	0.00
Need memory access	Both	Yes	No
Construction speed	Both	Slow	Fast
Pattern update speed	Both	Slow	Fast
Platform	Both	TCAM chip	FPGA

consumption of a TCAM cell is two times as large as a BCAM cell, our approach consumes 11.1% and 9.6% power of "CompactDFA" based on the Snort and ClamAV pattern sets respectively. However, the disadvantage of our approach is that it needs to be implemented on a configurable hardware platform such as FPGA. Fortunately, our approach is small enough to be configured into FPGAs. Further, checking four characters in a single clock cycle, allows our approach to easily achieve a throughput of 16 Gbps.

Increasing the number of parallel processing units will increase the CAM entry consumption. We illustrate how our approach scales in this increase as opposed to other approaches in Figure 4.7. As is evident, our approach scales better due to resource sharing by parallel processing units.

For the hierarchical approach, we focus on the first stage of substring matching. We selected 5,261 unique patterns with length of at least two characters from the Snort rule set as published by the Sourcefire Vulnerability Research team [113] for December 20, 2010. From these 5,261 patterns we identified only 333 unique substrings with length of two characters. Thus, a single substring set for this rule set would consume only 666-bytes of BCAM, (at two bytes per

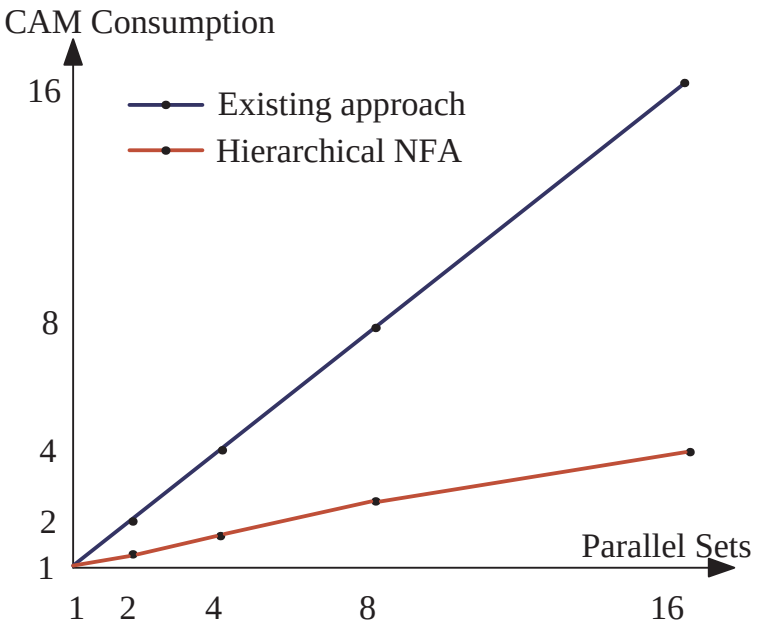


Figure 4.7: CAM entry consumption increases as the number of parallel processing units increases.

substring pattern) which is less than 1% of the total memory the BCAM consumes for the proposed pattern matching approach. We can extrapolate from these results and estimate that 32 such sets would only consume about 21% more memory than the approach as outlined in this section. This implies that such an approach is suitable for parallel processing and further implies that the substring matching is a valid pre-filter. In our simulation, we collected a total of 5,203 packets from attack exercises performed locally which exhibited several attacks against networks, primarily cross-site scripting attacks. In the substring pre-filtering stage, there are a total of 51,635 matched substrings and 1,887,078 unmatched substrings in all the packets' payload, which means only 2.7% of the aggregate packet payload matched any substring. More interestingly, the few matches that occurred typically happened in the same packets. In other words, only a few packets matched and those that did matched multiple times. About 90.3% of the packets contain no substring matches and do not need further checking. That means the majority of the traffic will not

require further processing than the pre-filter. With the clock speed of 500 MHz and an optimal balance of output from the pre-filter to pattern matching then our approach can achieve up to 128 Gbps throughput.

4.3.3 Summary

In this section, we studied more effective techniques for pattern matching in deep packet inspection. We built an efficient NFA generator based on BCAMs, which consumes fewer transistors and which has reduced latency because of shorter BCAM entries used in our approach. As a result, our approach demonstrates improved matching speed for every character while reducing the amount of SRAM resources needed. Also, our approach can process multiple characters at a time while using only a limited number of BCAM entries, which improves potential scalability scalable. Furthermore, we built a hierarchical pattern matching architecture which serves to exclude most packets from full pattern matching leaving only a small percentage to be fully checked in the pattern matching process. This hierarchical pattern matching improves throughput for the average case. In all, our evaluation demonstrates that our approach outperforms existing similar approaches.

Our approach here may offer a key lever for improving the throughput of regular expression matching through extracting exact-match strings from the regular expressions and preprocessing the matching dependent on those strings.

4.4 Hybrid Regular Expression Matching for Deep Packet Inspection on Multi-core Architecture

Many network security applications in today's networks are based on deep packet inspection, checking not only the header portion but also the payload portion of a packet. For example,

traffic monitoring, layer-7 filtering, and network intrusion detection all require an accurate analysis of packet content in search for predefined patterns to identify specific classes of applications, viruses, attack signatures, etc. Regular expressions are often used to represent such patterns. They are implemented using finite automata, which take the payload of a packet as an input string. However, existing approaches, both non-deterministic finite automata (NFA) and deterministic finite automata (DFA), have limitations; NFAs have excessive time complexity while DFAs have excessive space complexity. In this section, we propose an efficient algorithm for regular expression matching to implement deep packet inspection on multi-core architecture. A regular expression is split into NFA-friendly components and DFA-friendly components, which are then assigned to different cores. This hybrid method combines the merits of NFA and DFA implementations, and efficiently takes advantage of multi-core architecture. To the best of our knowledge, this is the first effort to design an efficient deep packet inspection algorithm for multi-core platforms. We evaluate our algorithm using rule sets provided by Snort, a popular open-source intrusion detection system. The evaluation results show that our approach is faster than NFA implementations while consuming less memory than DFA implementations.

4.4.1 Regular Expression Matching and Finite Automata

A network intrusion detection system (NIDS) classifies packets using a predefined rule set to determine whether packets are malicious or not by searching packet payloads for any signature in the rule set. Because of the increasing amount of network traffic and threats, intrusion detection systems become very resource-intensive. For instance, open-source NIDSs such as Bro [81] and Snort [89] expend all the resources, both CPU time and memory, and halt immediately when they are deployed under high-speed network environment [82]. Therefore, achieving high-throughput in pattern matching and reducing memory access frequency are crucial for overall intrusion detection

performance.

DFAs are more suitable for general-purpose processors and network processors because there is at most one transition for each incoming character. When checking a particular packet's payload, only a very small subset of rules are examined, which leaves a large portion of the memory occupied by the DFA unused. Moreover, with the complexity of regular expressions increasing, the amount of memory required by DFAs grows exponentially. For these reasons, DFA-based solutions do not seem to be ideal in deep packet inspection applications that require to process packets at the line speed with the growing number of regular expressions.

According to the Moore's Law, a trend that happened during the past decades in the semiconductor industry, the number of transistors in the integrated circuits (ICs) is doubling approximately every two years. The physical barriers such as clock frequency and chip temperature, however, make it impossible to follow such a law; the clock-speed increase no longer increases the performance of the processors. An alternative is to place multiple processor cores in a single chip. Nowadays, most processor vendors are increasing the number of cores in a single chip [114, 115]. The multi-core trend is observed not only in general-purpose processors but also in embedded processors [116, 117], such as network processors, digital signal processors, and cores embedded in FPGAs and GPUs. Furthermore, the number of cores in a single processor continues to increase. The performance gain achieved by the use of a multi-core processor depends on the algorithms used and their implementations in software. Hence, how to design efficient algorithms and implement them to take advantage of available parallelism in multi-core processors receives more attention. Since deep packet inspection is often a bottleneck in packet processing, exploiting parallelism in multi-core architecture is a key to improving overall performance. In this section, we propose an efficient algorithm for regular expression matching to implement deep packet inspection on multi-core architecture. For simplicity, we demonstrate the implementation of our algorithm on dual-core

architecture. The main idea in our algorithm is to classify regular expressions into two categories: NFA-friendly and DFA-friendly. Then we assign regular expressions or subexpressions to multiple cores according to their categories. NFA-friendly regular expressions are implemented as NFAs and DFA-friendly ones as DFAs. The evaluation results show that our approach is faster than NFA implementations while consuming less memory than DFA implementations.

In this section, we first examine NFA and DFA implementations for regular expression matching, which become basis of the hybrid approach we propose in Section 4.4.2.

(1). Types of Regular Expression Components

In order to increase parallelism in regular expression matching, we first need to study types and characteristics of regular expression components. Although the regular language itself is a well-defined and well-understood language, there are many variants adopting additional notations to make the language more human-friendly. In this section, we consider some of the main types of regular expression components frequently found in Snort rule sets. In the following, we present them in the increasing order of their complexity.

1. Exact-match strings are the simplest kind of regular expressions. They are fixed-size patterns, and thus the number of states in a finite automaton (DFA or NFA) can be kept less than the number of characters in the regular expression (string). The Aho-Corasick algorithm [118] or the Boyer-Moore algorithm [119] can be used without modification, and hashing can be used for optimial performance. However, this type of regular expressions is not expressive enough, and cannot detect malicious packets if an attacker inserts padding in them. So, the percentage of this kind is dropping fast.

2. Character sets and single wildcards such as "$[c_i{-}c_jc_k]$" and "$.$". For this type of regular expressions, the exact-match algorithms such as the Aho-Corasick and Boyer-Moore

algorithms or hashing schemes cannot be used directly. Instead, exhaustive enumeration of exact-match strings should be used. These regular expressions are more expressive than exact-match strings, but require larger finite automata with more transitions.

3. Simple character repetitions such as "c?", "$c\star$", and "c+". For this type of regular expressions, exhaustive enumeration of exact-match strings are inapplicable because the length of a matched string may be infinite. However, it can be efficiently implemented as a loop transition in a finite automaton.

4. Bounded repetitions such as "c\{n,m\}", "[^c_i-c_j]{n,}". For this kind of regular expressions, the number of states of a finite automata grows fast as the counting constraints increase. However, we can introduce counters as an augmentation to finite automata to alleviate this problem.

5. Character sets and wildcards repetitions such as "[^c_i-c_j]$\star$", ".$\star$", "[^\r\n]*". If multiple such regular expressions are implemented as a single DFA, the size of the DFA can grow exponentially. We demonstrate it through experiments in the next part.

In practice, most regular expressions in NIDS have more than one kind of regular expression patterns mentioned above in a single regular expression.

(2). Size of Finite Automata with Complex Regular Expressions

With the evolution of network threats and evation techniques, the length and complexity of regular expressions in rule sets are increasing fast. Building a DFA of a set of complex regular expressions potentially results in an exponential number of DFA states and exponential build time. To investigate the impact of the complexity of regular expressions on the size of finite automata, we perform experiments with hundreds of real attack signatures. Furthermore, the number of this kind

of complex regular expressions is growing fast. For the number of states explosion problem, we can divide the regular expressions into several small sets and build several DFAs, and our experiment shows this is an efficient method to reduce the number of states, but multiple DFAs may have active states at the same time and work in parallel, which increase the bandwidth requirement. For the building time explosion problem, we can build the DFA "on-the-fly" during the actual matching instead of pre-computing the huge DFA. By this way, we only need to store DFA states that are actually needed. However, this will cause more time to build the DFA before matching malicious traffic in practice.

In our experiments, we implement the NFA-based algorithm and DFA-based algorithm proposed by Ficara et al. [101]. Regular expressions are selected from Snort rule sets. Since the goal of these experiments is to understand the effect of complex regular expressions on finite automata, we use the last type of rules defined in the previous part, which can cause exponential growth of the size of build time of finite automata. So, the selected regular expressions are long and complex, containing many wildcard repetitions. Note that this type of regular expressions exhibits an increasing trend in terms of the number of rules in NIDS. The other types are not included becuase they can be implemented efficiently either by augmenting finite automata or by other algorithms without finite automata. All the experimental results reported in this section were obtained on an Intel 3.0 GHz Dual-Core machine with 4 GB main memory.

We first measure the size of NFA for different rule sets to estimate space complexity. We choose three sets of regular expressions from Snort rule sets, calling them R_1, R_2, and R_3. They contain 98, 185, and 298 regular expressions, respectively. For comparison with simple regular expressions, we alse create three rule sets, E_1, E_2, and E_3, each of which has the same number and average length of exact-match regular expressions as the corresponding R_i.

Table 4.3 shows the number of states and the number of transitions for each configuration.

Table 4.3: Number of NFA states and transitions

Rule set	# of states before/after compression	# of transition before/after compression
E_1	2746/1387	3596/1892
R_1	2348/1206	15006/7220
E_2	6044/3024	7124/3549
R_2	5136/2631	28756/13398
E_3	9554/4683	11264/5797
R_3	8048/4176	41155/20014

For completeness, each column contains the number before applying the compression algorithm by Ficara et al., as well as the number after compression. For visual comparison between the simple regular expression sets, E_i, and the complex sets, R_i, we plot the numbers in Figures 4.8 and 4.9.

We can see from those Figure 4.8 that, in NFA, the number of states grows proportionally to the number of regular expressions, and there is only a slight difference between the simple regular expressions and the complex regular expressions. On the other hand, the difference between the numbers of transitions is much larger, as shown in Figure 4.9. This result indicates that the copmlexity of regular expressions mainly affect the number of transitions, not the number of states.

In Table 4.4, we compare the build time of NFA and DFA. The build time of NFA is expected to exhibit the linear tendency as we observed in Figures 4.8 and 4.9, because creating states and transitions is the main task in building finite automata. The first column of Table 4.4 confirms this. However, the second column shows a very different trend. For DFAs, while the build time of the simple regular expressions grows linearly with the increase of the number of expressions, the build time of the complex regular expressions grows exponentially with the increase of the number of regular expressions. Therefore, it would be desirable to avoid DFA-only implementations for today's complex rule sets.

We also compare the number of states between NFA and DFA. The results are shown in Table 4.5, and the corresponding chart is presented in Figure 4.10.

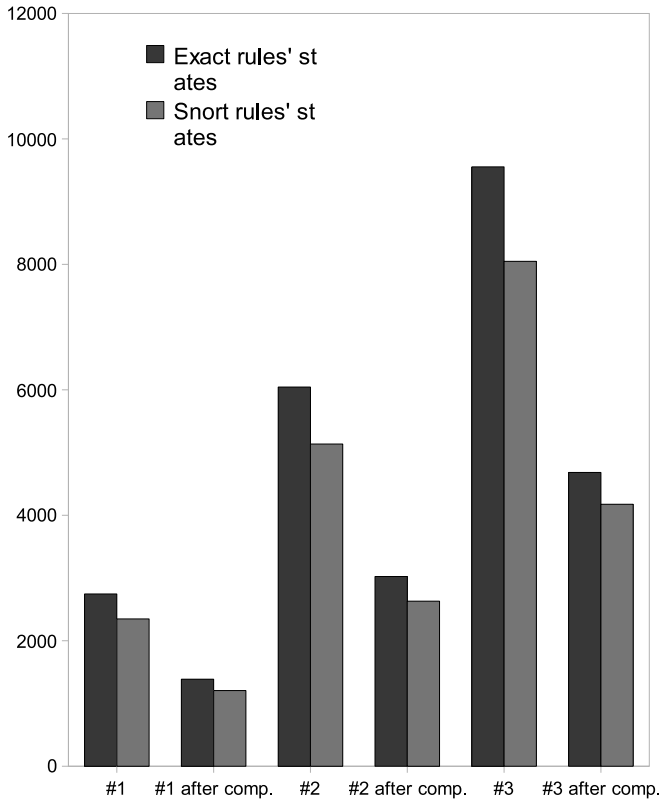


Figure 4.8: Number of NFA states

Table 4.5 and Figure 4.10 show the exponential growth of DFA more clearly. While NFA still demonstrates linear growth, regardless of the complexity of regular expressions, the number of states of a DFA exceeds 200000 for R_3, which has fewer than 300 rules.

Note that the last row in Table 4.5. It corresponds to a finite automaton implementing R_3 with three DFAs. It is obtained by dividing R_3 into three components, building a DFA for each component, and then combining them using a NFA. The numbers of states of DFAs are 9304, 15782, and 22149, respectively. This means that we can use several small DFAs to replace a single huge DFA to reduce the number of states. Of course, such an approach will increase the memory

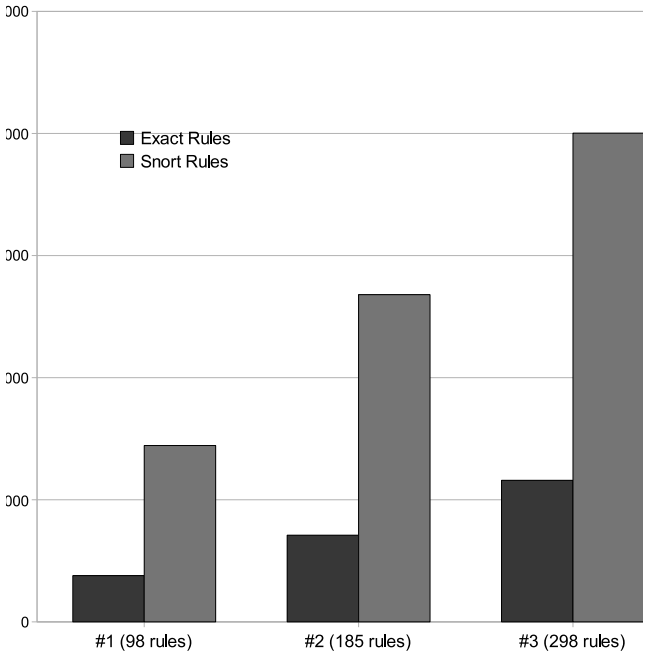


Figure 4.9: Number of NFA transitions

bandwidth requirement, resulting in more memory accesses when there is not enough bandwidth.

To summarize, we observe in the experiments that DFA is more efficient for the regular expressions with less complexity, such as exact-match, character sets, and simple character repetitions. For more complex regular expressions, however, NFA is more efficient and the cost of DFA implementation is prohibitive. This finding leads us to a hybrid approach, which is presented in Section 4.4.2.

4.4.2 Hybrid Approach

(1). DFA-Friendly Regular Expressions and NFA-Friendly Regular Expressions

For better utilization of multi-core architecture, we need to split regular expressions in an

Table 4.4: Build time of NFA and DFA

Rule set	Build time of NFA (sec)	Build time of DFA (sec)
E_1	0.36	1.98
R_1	0.81	103.98
E_2	1.24	4.57
R_2	1.93	432.56
E_3	2.08	9.05
R_3	3.93	2928.92

Table 4.5: Number of states comparison between NFA and DFA

Rule set	# of states in NFA	# of states in DFA
E_1	1381	1370
R_1	1207	16673
E_2	3024	3012
R_2	2630	64297
E_3	4781	4779
R_3	4176	> 200000
R_3 with 3 DFAs	–	47235

efficient manner. As we observed in Section 4.4.1, a regular expression can be implemented using multiple DFAs combined with a NFA. To apply this idea, we need to be able to tell which part of a regular expression is better implemented as a DFA and which as a NFA. These are what we call DFA-fridnely regular expressions and NFA-friendly expressions, explained in the following.

DFA-friendly regular expressions Regular expressions suitable to be implemented as a DFA. Such regular expressions must keep the number of states and build time of DFAs reasonably small. Implementing them as a NFA would not bring any gain in terms of memory bandwidth.

NFA-friendly regular expressions Regular expressions suitable to be implemented as a NFA. The DFA implementation of such regular expressions will have an exponentially large number of states and build time. By implementing them as NFAs, memory consumption can be

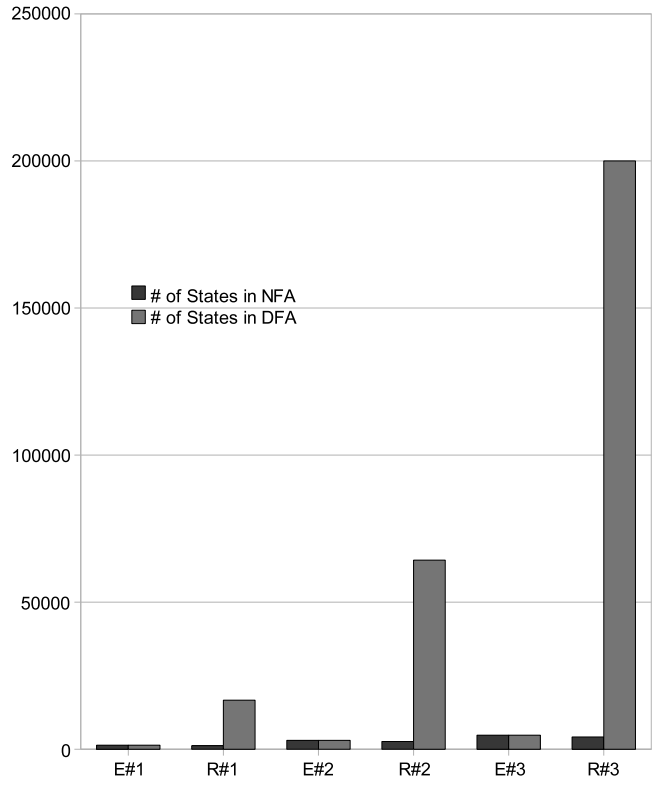


Figure 4.10: Number of states comparison between NFA and DFA

reduced significantly.

Actually, there is no clear boundary between these two groups of regular expressions. When we categorize regular expressions in practise, we also need to consider many other factors such as the actual regular expression set and payload balance between cores.

(2). Proposed Multi-Core-Based Algorithm

The hybrid approach we propose in this section is based on the following observations in our experiments:

1. For simple regular expressions, DFA is more efficient (DFA-friendly). Such regular expres-

sions include exact match, character sets, and simple character repetitions.

2. For complex regular expressions, NFA is more efficient (NFA-friendly). Such regular expressions include wildcards repetitions.

3. Majority of packets match only short prefixes of regular expressions.

4. In typical complex Regular expressions, a DFA-friendly expression and a NFA-friendly expression appear in an alternating manner as in "`abc.*defg.*hijklmn.*opq`".

The first step in our approach is to identify all DFA-friendly components (substrings), $d_1, d_2, \ldots, d_m$, from the regular expressions included in a given rule set. Each DFA-friendly substring d_i is converted into a DFA D_i, which works as a coprocessor in our implementation.

Every instance of d_i in the original rule set is replaced with a reference to D_i. Then this modified rule set is converted into a single NFA, which becomes the main processor, as shown in Figure 4.11. Since all DFA-friendly substrings were removed, we expect this NFA is of reasonable size.

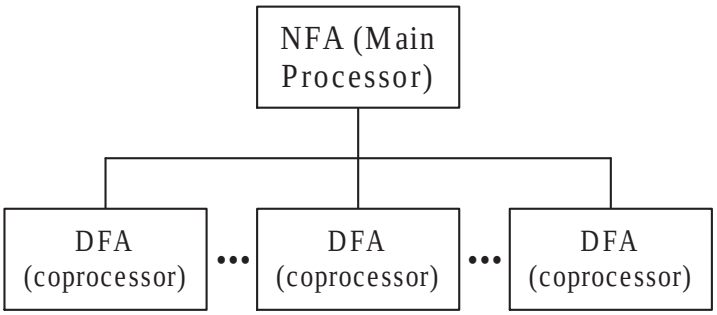


Figure 4.11: Hybrid architecture of deep packet inspection on multi-core platform

For better utilization of multi-core architecture, we identify the followings as key issues in designing the algorithm.

1. Memory access patterns of cores are critical in multi-core architecture. Different cores may use the same area in the main memory, and the frequency and locality of memory accesses will affect the overall performance significantly. Thus, each core should minimize the number of accesses to shared contents in the main memory, which may cause bandwidth contention and consistency problems.

2. Even with a single core, increasing locality is an important factor in reducing memory access latency. In multi-core architecture, this becomes more crucial because higher cache miss rates of multiple cores will aggravate bandwidth contention, resulting in poor performance.

3. To minimize idle time, the load should be evenly distributed among cores.

The architecture in Figure 4.11 addresses all of these issues. First, each processor uses its own data (states and transitions). Second, because of the reduced memory consumption demonstrated in Table 4.5, the overall cache hit ratio should be higher than DFA-only or NFA-only approaches. Third, although the main processor (NFA) needs more memory bandwidth, which may decrease the overall performance, for most of the time a branch of the NFA is waiting for coprocessor's responds (DFA matching), or doing repetition on the same state, or being inactive, none of which consume memory bandwidth. Furthermore, we find that the size of the NFA in the main processor is small enough for most rule sets that the states and transitions can reside in its own cache. All of these contribute to reducing memory accesses by the main processor, alleviating memory bandwidth contention. Finally, for even distribution of loads, we may classify character sets and wildcards repetitions as DFA-friendly regular expressions.

A simple example for this approach is illustrated in Figure 4.12. Four original regular expressions are used in this example, and eight DFA-friendly substrings are identified and converted into DFAs (left-hand side). The remaining parts of regular expressions are converted into a NFA

(right-hand side).

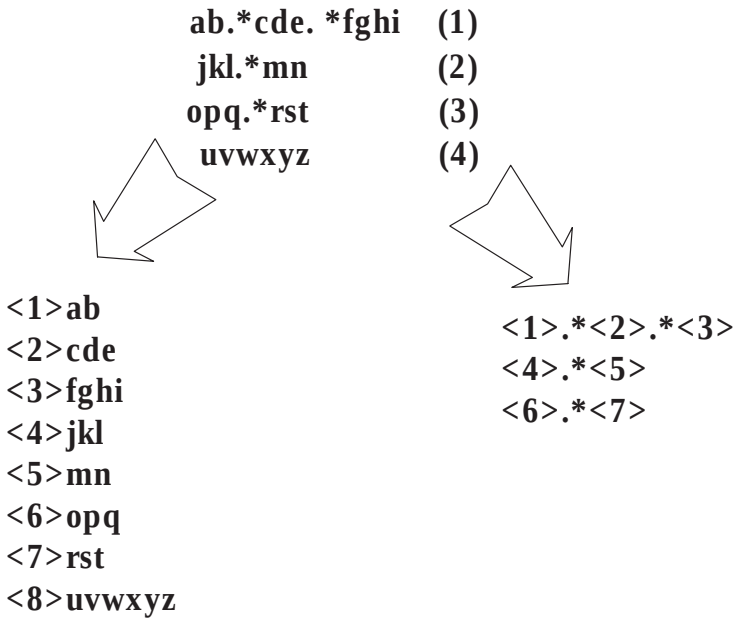


Figure 4.12: An example of the hybrid approach

4.4.3 Evaluation

To evaluate the proposed hybrid approach, we build a single NFA as the main processor and build a single DFA as the coprocessor using the 298 rules in the R_3. We compare a single-DFA approach, a single-NFA approach, and our hybrid approach in terms of memory consumption (state number) and processing speed (clock cycles/character).

We make the following assumptions for evaluation.

- Each core can only fetch a single state or transition per memory access.
- Each core's cache can store 2000 states or transitions.
- A cache hit and a cache miss need 2 and 30 clock cycles, respectively,

- Five percent of branches in the NFA are active all the time and these active branches may access memory at the same clock cycle.
- The average length of DFA-friendly substrings is about 10 times of the average length of NFA-friendly ones.
- The average number of repetitions is 30 for each NFA-friendly substring.

Note that the last two items conform to what we observe in Snort rule sets.

Based on the assumptions above, we choose the boundary between the DFA-friendly and NFA-friendly regular expressions for a rule set as follows. We first define the value V, the weighted number of states in DFA and NFA as in the following equation:

$$V = D + \alpha N \tag{4.1}$$

where D and N are the numbers of states in DFA and NFA, respectively, and α is a parameter that depends on the memory bandwidth or additional memory accesses needed by NFA. In our experiments, we set $\alpha = 5$. We classify regular expressions into 5 types as we presented in Section 4.4.2. We use Types "1–2" to represent the boundary between the first two categories, "2–3" between the second and the third, and so on. The results are shown in Figure 4.13. Since the values of V for 0–1 and 5–0 are much larger than others, so we chose 4–5 as the boundary between the DFA-friendly and NFA-friendly in our simulation.

The numbers of states are shown in Figure 4.14. The single-DFA has more than 200000 states, the single-NFA has 4176, and the hybrid approach has the fewest, 4093, which consists of 3110 DFA states and 983 NFA states.

The clock cycles per character is plotted in Figure 4.15. The single-DFA achieves 29.72

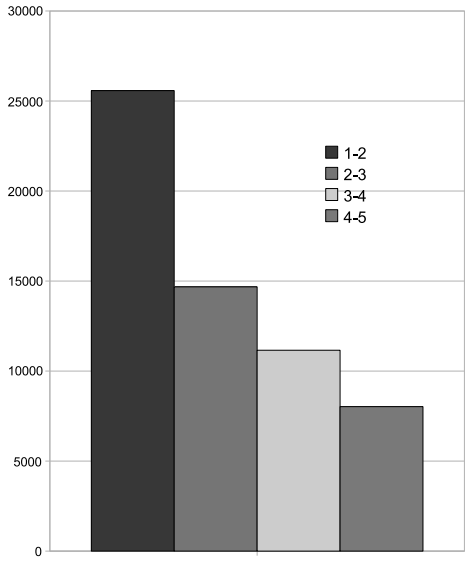


Figure 4.13: V for different boundaries

cycles/character with cache hit rate 1%, the single-NFA 80.3 with hit rate 47.89%, and the hybrid approach achieves 23.98 ($= 11.99 \times 2$) cycles/character with hit rate 64.31% for DFA and almost 100% for NFA. For fair comparison, we double the clock cycles used for each matching character.

From the evaluation results we can see that for the complex rule set, a DFA consumes too much memory while a NFA is slow due to multiple active states accessing memory in parallel. Our algorithm outperforms these approaches not only in memory consumption but also in processing speed.

4.4.4 Summary

In this section, we studied techniques used in deep packet inspection, in particular regular expression matching with growing complexity. We built efficient NFA/DFA generators and investigated the impact of the complexity of regular expressions using rule sets from actual NIDS. We demon-

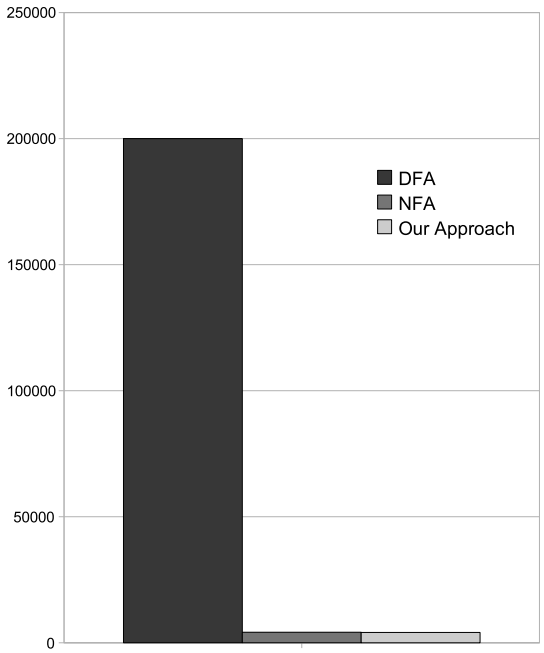


Figure 4.14: Memory consumption

strated that a singe NFA or a singe DFA performs poorly given a large set of complex regular expressions. Because multi-core processors are expected to dominate, there will be great demand on fast deep packet inspection algorithms that can utilize multi-core architecture for complex regular expressions. Thus, we proposed a hybrid algorithm that combines both NFA and DFA. It divides a complex regular expression and configures them into multiple cores to take advantage of available parallelism provided by multi-core processors. The evaluation results show that the hybrid approach outperforms the single DFA and single NFA approaches. With the advent of processors with more embedded cores, achieving maximum parallelism not only in deep packet inspection but in general packet processing will be crucial to success of network security applications. We plan to expand this work to general traffic monitoring systems with pipelining architecture.

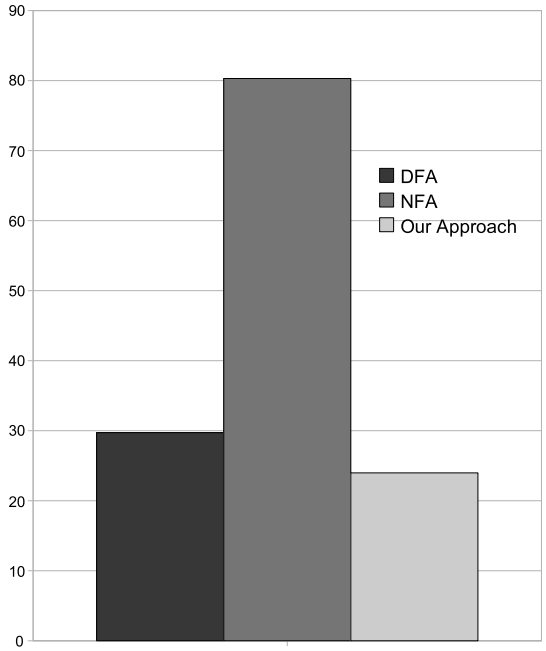


Figure 4.15: Evaluation on Processing Speed

4.5 DFA-based Regular Expression Matching on Compressed Traffic

Regular expressions are implemented using finite automata, which take the payload of a packet as an input string. However, existing approaches, including non-deterministic finite automata (NFA) and deterministic finite automata (DFA), do not deal with compressed traffic, which becomes more and more popular in HTTP applications. In this section, we propose an efficient algorithm for regular expression matching to implement deep packet inspection on compressed traffic. Based on the observations of DFA, we design a scheme to skip most of the matching process in the compressed parts of traffic. To the best of our knowledge, this is the first effort to design an efficient regular expression matching on compressed traffic. We evaluate our algorithm using

rule sets provided by Snort, a popular open-source intrusion detection system. The evaluation results show that our approach can reduce the number of state accesses in a DFA significantly.

4.5.1 Regular Expression Matching, Finite Automata, and Compressed Traffic

A network intrusion detection system (NIDS) classifies packets using a predefined rule set to determine whether packets are malicious or not by searching packet payloads for any signature in the rule set. Because of the increasing amount of network traffic and threats, intrusion detection systems become very resource-intensive. For instance, open-source NIDSs such as Bro [81] and Snort [89] expend all the resources, both CPU time and memory, and halt immediately when they are deployed under high-speed network environment [82]. Therefore, achieving high-throughput in regular expression matching and reducing memory access frequency are crucial for overall intrusion detection performance. In deep packet inspection, the regular expression matching becomes the bottleneck in the network applications and exhausts the CPU and memory resources, and the decompression procedure further degrades the overall performance. Therefore, to reduce the frequency of state accesses (memory accesses) is the major challenge, and our goal is to skip as many input characters as possible to avoid unnecessary memory accesses.

As we mentioned earlier, a finite automaton is either non-deterministic or deterministic; a non-deterministic finite automaton (NFA) may have multiple state transitions per character in the payload, which means there may be multiple active states at the same time. This possibility makes it very difficult to record previous state sequence, and thus an NFA not suitable for checking the compressed traffic. On the other hand, a DFA only has one active state all the time, which makes its state sequence easily to be recorded and reused. Therefore, we choose a DFA to represent regular expressions instead of NFA for compressed traffic.

The LZ77 [120] compression algorithm is commonly used in today's Internet traffic, and

the basic idea of the LZ77 compression algorithm is that when a series of continuous characters (a substring) has already appeared in the near past, a pair of numbers may be used to represent this repeated substring. The pair of numbers is also called a length-distance pair, which represents the distance in bytes of the two substrings and the length in bytes of the repeated substring. We also call such a pair of numbers a *pointer*, and the compressed substring its *pointer area*. For example, the plaintext `abcdefgabcde` will be compressed into `abcdefg(7,5)`. When performing regular expression matching on such compressed traffic, the naïve approach is to decompress the traffic first and then perform regular expression matching on the plain text as usual. However, this process is costly because the deep packet inspection itself consumes a lot of CPU time and memory resources. In this section, we propose an efficient algorithm to perform regular expression matching directly on compressed traffic, exploiting the properties of the compression algorithm and DFA.

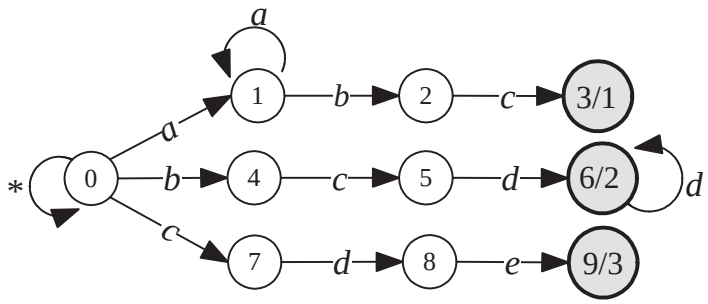
4.5.2 Our Approach

There have been many efforts to reduce memory consumption in DFA implementations. Kumar et al. developed the Delayed-input DFA (D^2FA) [87], which reduces space requirements by reducing the transitions based on the observation that many states have similar sets of outgoing transitions. In D^2FA, such transitions are grouped and represented by a single default one. In this way, their algorithm achieves a reduction of memory consumption by more than 95%. However, the drawback of this approach is that it may need to process multiple states for a single input character, which increases overall memory bandwidth. Later, Ficara et al. proposed Delta Finite Automata [101]. It is based on the same observation as D^2FA, but only requires a single transition per input character.

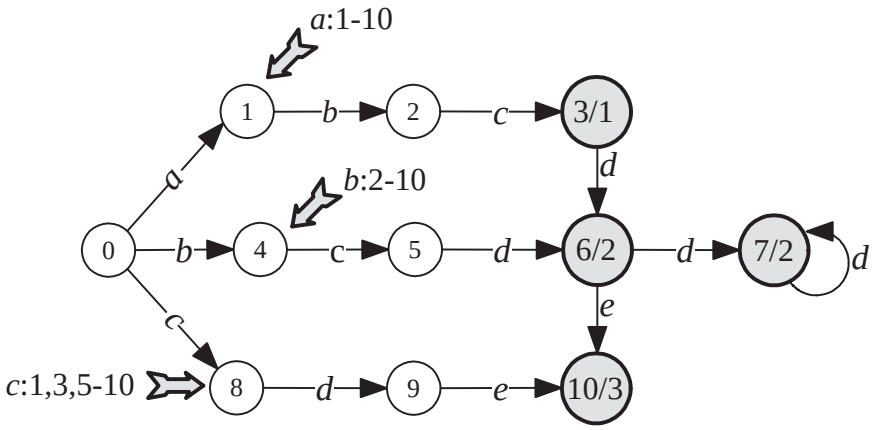
We revisit this from another viewpoint. The fact that redundant transitions usually occupy more than 95% of total transitions implies, given an input character, it is very likely that the next

state is the same state, regardless of the current state. The algorithm we describe below is based on this observation to to perform regular expression matching on compressed traffic.

To clarify this property of DFA, we analyze the same example brought by Becchi et al. [121]. In Figure 4.16, we show the NFA and DFA accepting regular expressions $a + bc$, $bcd+$, and cde on the alphabet $\Sigma = \{a, b, c, d, e\}$, constructed in the standard way [122]. Transitions leading to state 0 are omitted for brevity. The big arrows represent transitions leading to the current state.



(a)



(b)

Figure 4.16: (a) NFA and (b) DFA for regular expressions $a + bc$, $bcd+$, and cde

We show two examples of skipping the entire pointer area and skipping a part of the pointer area in Figure 4.17. The terminologies used in the figure are defined below.

- C-Trf: Compressed traffic including pointers
- D-Trf: Decompressed traffic without pointers
- State: State number in the DFA
- Status: The status of state in the DFA, unmatch (u) or match (m)

Example 1:

C-Trf : *e b c d a e c b* {6, 6} *c*
D-Trf : *e b c d a e c b c d a e c b c*
State : 0 4 5 6 1 0 8 4 - - - - - 4 5
Status : u u u m u u u u - m - - - - u

Example 2:

C-Trf : *e b c d e a b c* {6, 6} *c*
D-Trf : *e b c d e a b c c d e a b c c*
State : 0 4 5 6 10 1 2 3 8 9 10 - - 4 5
Status : u u u m m u u u - - m - - - u

Figure 4.17: Skipping the entire pointer area in Example 1, and skipping a part of the pointer area in Example 2

In the DFA in Figure 4.16(b), we can see that for a given input character, the possible next states are very limited, as shown in Table 4.6; for the same input character, it is likely to reach the same state, even from different current states.

Most of the time, the active state is the start state, which is usually state 0, or states reachable from the start state with a single input character. In Figure 4.16(b), states 0, 1, 4 and 8 are

Table 4.6: States comparison between NFA and DFA

Input character	Next state	ratio between these states
a	1	1
b	2, 4	1:10
c	3, 5, 8	1:1:9
d	6, 7, 9	2:2:1
e	10	1

such states. In our experiment, for more than 70% of time (1361 out of 1835) the active state is one of these states. That is because traffic seldom matches attack signatures in practice. There are 257 such states including state 0, because there are 256 symbols represented by 8 bits, and they create $2^8 = 256$ states as the next states from state 0. If the destination state is in this set of states, the same input character will lead all states to the same state. In other words, the next state is deterministic and it does not matter what the current state is. We call the part of DFA including these states the *Deterministic Area.* Note that 95% of transitions lead to this area as we discussed above. In the example shown in Figure 4.16(b), there are a total of 56 transitions, and 46 of them lead to states in the Deterministic Area, which means the probability of going into the Deterministic Area is about 82% on average in this example. With such a high probability for the current state to be in the Deterministic Area, the probability of entering the Deterministic Area in the next step is even higher in practice. We call the rest of DFA the *Non-deterministic Area.* Note that for the same input character, it is possible to reach the same state from the different current states, even in the Non-deterministic Area, e.g., $3 \rightarrow 6$ and $5 \rightarrow 6$. With more attack traffic, there will be more matches, and there will be more activities in the Non-deterministic Area, because it is likely to go deeper in the DFA. The Deterministic Area and Non-deterministic Area in a DFA are formally defined as follows:

- Deterministic Area: The starting state (state 0) and the states reachable from the starting

state through a single transition.

- Non-deterministic Area: States that are not in the Deterministic Area.

If we need to check the pointer area for a long substring and cannot skip these characters, there must be at least two paths in the DFA that accept the same input substring. We call these two paths a *path pair*. With the adoption of today's DFA compression techniques, however, the number of path pairs is decreasing significantly. In other words, each path is more likely to be unique, because the redundant parts have been combined or reduced. This fact helps us to skip more characters in compressed areas.

For example, in the DFA in Figure 4.16(b), the worst case is caused by such a path pair, $2 \Rightarrow 3 \Rightarrow 6 \Rightarrow 10$ and $0 \Rightarrow 8 \Rightarrow 9 \Rightarrow 10$, for the same input string `bcd`, which is shown in Figure 4.18. In this example, we need to check the entire pointer area again.

C-Trf :	*e*	*c*	*d*	*e*	*a*	*b*	{5, 3}	*c*		
D-Trf :	*e*	*c*	*d*	*e*	*a*	*b*	*c*	*d*	*e*	*c*
State :	0	8	9	10	1	2	3	6	10	8
Status :	u	u	u	m	u	u	m	m	m	u

Figure 4.18: The worst case in Figure 4.16 caused by a path pair

Since path pairs may adversely affect our algorithm, we propose a method to reduce the number of path pairs by combing them. In the same example shown in Figure 4.16(b), we combine the two sets of path pairs, and the results are shown in Figure 4.19.

We use tokens to represent different paths. A token is created when the label of the transition includes a dash, as shown in Figure 4.19, and it remains while the current transition label has an underscore. An existing token is removed if the transition does not have an underscore. There may be multiple tokens at the same time. The state 8 or state 9 can be a match state if at least

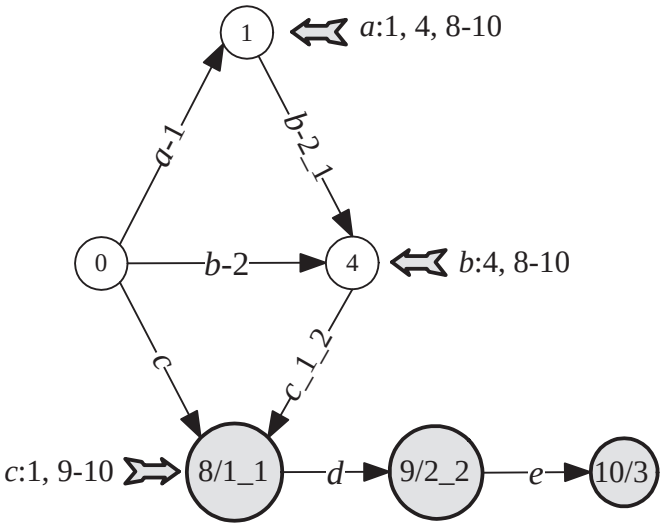


Figure 4.19: The compressed DFA for the example in Figure 4.16(b)

one of the current tokens matches the token in the state (token 1 and token 2 in state 8 and state 9, respectively). Therefore, after the compression, there remains no path pair in our example, and thus all characters excluding the first one in the pointer area can be skipped. The worst case shown in Figure 4.18 is shown in Figure 4.20 after the path pair compression.

C-Trf :	*e*	*c*	*d*	*e*	*a*	*b*	{5, 3}	*c*		
D-Trf :	*e*	*c*	*d*	*e*	*a*	*b*	*c*	*d*	*e*	*c*
State :	0	8	9	10	1	4	8	-	-	8
Token :	-	-	-	-	1	1,2	1,2	2	-	-
Status :	u	u	u	m	u	u	m	m	m	u

Figure 4.20: The worst case in Figure 4.18 after path pairs compression

4.5.3 Evaluation

In our experiment, the deeper we check the pointer area, the more chances we have to reach the same state as the previous substring, as indicated by the trend in Figure 4.21. 0 on the x-axis means

the whole pointer area is skipped. We can see that after checking seven characters in the pointer area, we may skip the rest of the pointer area with the probability greater than 90%.

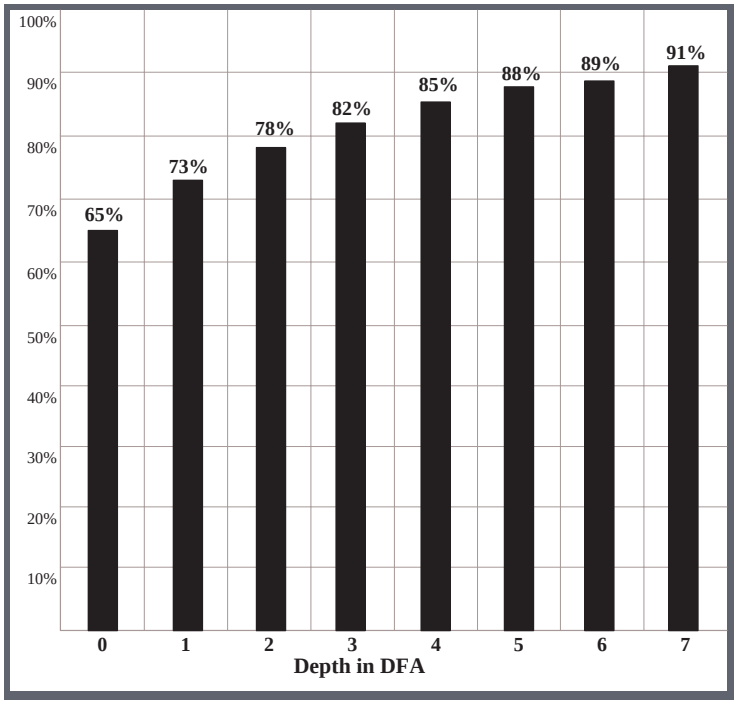


Figure 4.21: The probability to enter the same state as the previous substring

We select five sets of traffic with different compression ratios, between 10% and 90%, and build DFAs using three different regular expression sets, R_1, R_2, and R_3, with 100, 200, and 400 regular expressions, respectively, collected from the Snort rule set. We simulate the numbers of DFA state accesses in the original approach and the numbers of DFA state accesses in the proposed approach. The results with 100 regular expressions are shown in Table 4.7. The percentages of DFA state accesses saved by our approach are plotted in Figure 4.22.

From the evaluation results, we can see that our approach effectively reduces the number of state accesses, and the percentage of such savings depend mainly on the compression ratio, not on the number of regular expressions used. Besides, out approach does not introduce any false

Table 4.7: Percentages of DFA state accesses saved with different compression ratios

Compression ratio	10%	30%	50%	70%	90%
Number of original DFA state accesses	1364	1473	1753	1835	1583
Number of our DFA state accesses in R_1	1240	1083	1015	795	469
Number of our DFA state accesses in R_1	1241	1092	1029	804	505
Number of our DFA state accesses in R_1	1244	1093	1011	815	503

positive or false negative in matching.

4.5.4 Summary

In this section, we proposed an efficient regular expression matching in deep packet inspection for compressed traffic. We first analyzed the properties of NFA and DFA, and chose DFA because it maintains at most one active state. Based on the observation that for the same input character there is a high probability that the next state will be the same state regardless of the current state, we built an efficient DFA generator that combines some states and transitions to improve this property further. Our algorithm records the previous active state sequence and matching sequence. This information allows us to skip some compressed parts of the traffic when the active state is repeated. The simulation results show that our approach effectively skips most of the compressed parts in the traffic and reduces the frequency of DFA state accesses in proportion to the traffic's compression ratio. Although DFA is not suitable for a large number of complex regular expressions [106], our approach can be easily extended to be used in separate, multiple small DFAs, which will make our approach more scalable.

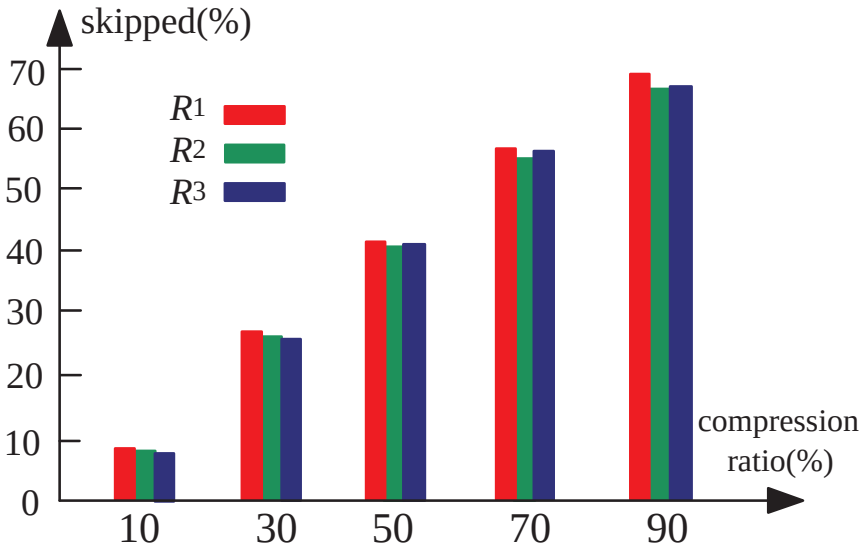


Figure 4.22: Percentages of DFA state accesses saved

CHAPTER five

PACKET PROCESSING ENGINE

5.1 Introduction

How to design efficient algorithms to promote these three components' performance is my one goal, and my another goal is to combine these components into a single integrated device, which can reduce processing time and resource usage by working in parallel or sharing some resources. In today's router, these three procedures are usually implemented as different components, which make the delay of total packet processing too slow to meet high-speed network's requirement, because all these procedures must be performed one by one. So if I can parallel them or overlap them efficiently in time or share some hardware resources in location, then I can improve the overall performance dramatically. Furthermore, these three tasks overlaps with each other sometimes, which means some redundant parts of these tasks can be removed. For example, some packets should be dropped by the packet classification do not need to be processed by IP route lookup and deep packet inspection, then how to eliminate these meaningless processing efficiently is also important. So I also propose a new integrated architecture which combines these three parts efficiently or prove that combining the three components is not good enough based on today's techniques.

A major challenge of integrating these components is the hardware limitation in limited space and power/cooling budget. There are two directions to eliminate these problems: first, share hardware resource between different components to meet the limitation, second, we only integrate important parts of different components as a fast path to process most of the tasks, then we can not only improve the overall throughput but also release the burden of remaining processes. So we

work towards both directions.

5.2 Proposed Combined Architecture

My combined approach are plotted in Figure 5.1.

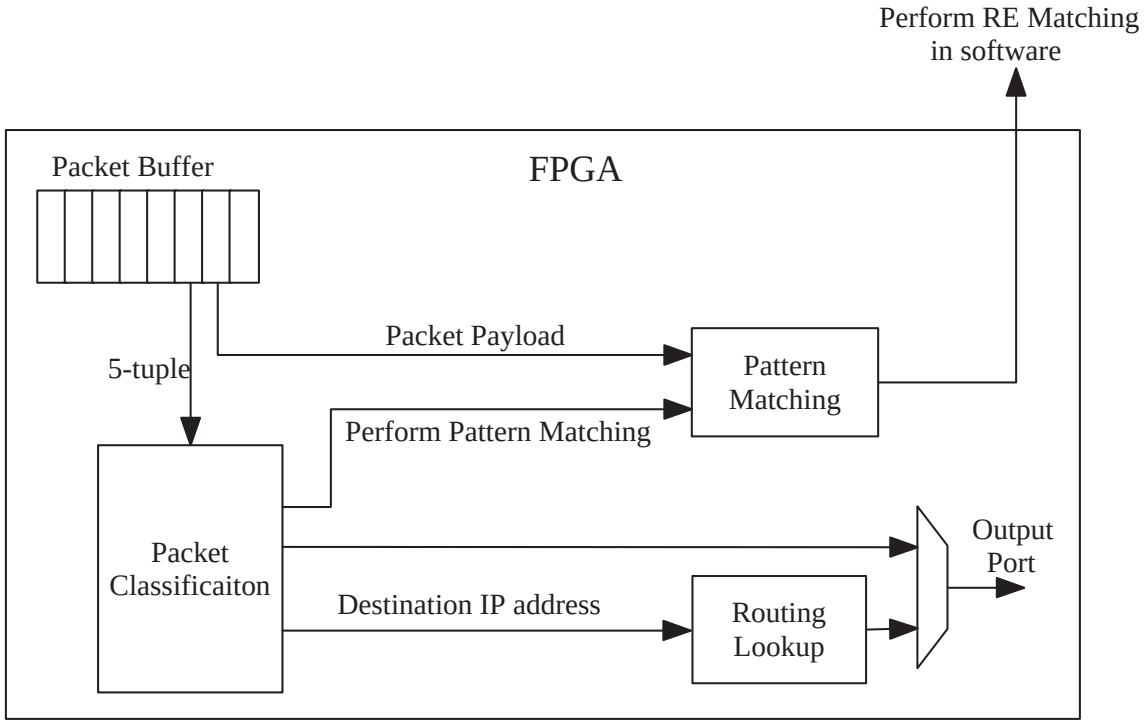


Figure 5.1: The top architecture of combined three components in a FPGA

Each packet is processed by packet classification unit first, as we discussed in previous chapters, we reduce the redundancy in the packet classifier and do not need to store rules with drop decision into TCAMs, so a packet does not need further process if it does not match any rule in the packet classifier. Otherwise, the packet will be processed by routing lookup unit and pattern matching unite simultaneously. If there is any match in the pattern matching, further regular expression matching will be performed in the software approach proposed. Rules stored in the

TCAM contain a destination IP address field, and each rule matches one or more IP prefixes in the routing table. If a rule only match a single output port in the routing table, then we can include the corresponding output port of the IP prefix in the rule decision, and we can get the corresponding output port directly from the packet classifier. If a rule matches more than one output ports, then the packet should be checked by the routing table lookup process. Because there is no overlap between any two rules in the packet classifier, we can store rules with the same decision together and the combined architecture is shown in Figure 5.2.

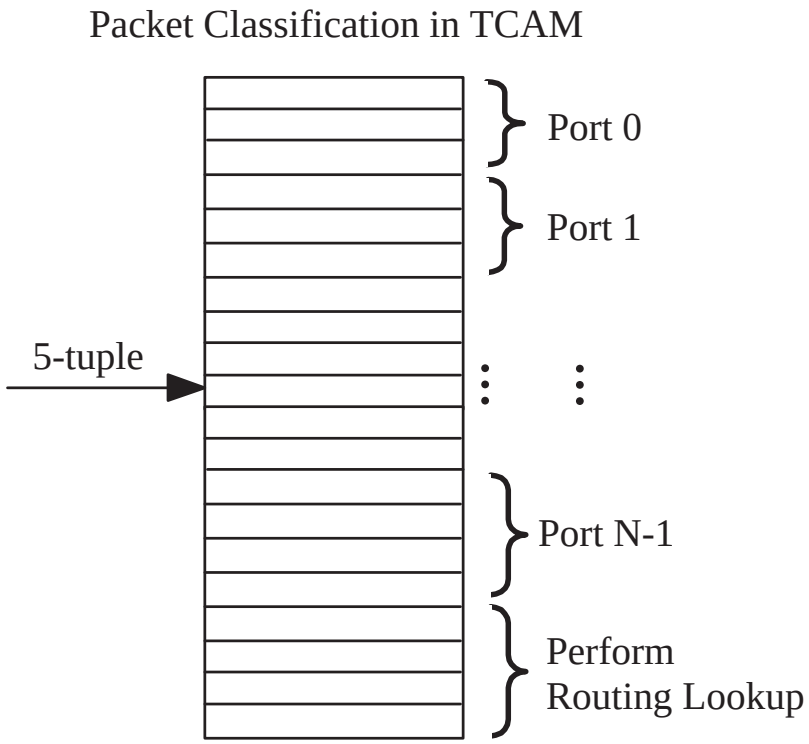


Figure 5.2: The architecture of packet classification including output port numbers

After combining rules in the packet classifier with IP prefixes in the routing table, we can remove some useless IP prefixes from the routing table. If a rule in the packet classifier matches more than one output ports in the routing table, these IP prefixes in the routing table must be

stored in TCAM, otherwise, the IP prefixes do not need to be stored in the TCAM. Fortunately, there are usually a small number of unique destination IP addresses, for example, there are only 15 unique destination IP addresses in Table 3.3. Note that we can not remove some rules in the packet classifier by our combining approach because rules in the packet classifier have more tuples than IP prefixes in the routing table.

In the worst case, the packet processing process will not be simplified much, and the latency will be the sum of packet classification, pattern matching and regular expression matching, which only skip the routing table lookup process. However, benign packets are more likely to go through the fast path only including the packet classification and pattern matching. So this will release the congestion in the nodes.

As a case study, Table 3.3 in Chapter 3 presents the analysis of the complete set of the 8214 Snort rules [70], and there are only 15 distinguish destination IP addresses. We also analyze the destination IP addresses in the latest Snort rules published in August 2011, and the results are shown in Table 5.1. We can see that there are only 13 unique destination IP addresses. Note that in the 174 "any", 8 of them only check the packet payload, 163 of them are used to check DNS service in TCP port 53 such as "BLACKLIST DNS request" with "discard" decision, and the remainder three are used to check overflow attempt in port 68, 69 and 5190 with "discard" decision.

We use the rules generated from [72]. In [72], 12 real packet classification rule sets are analyzed, Table 5.2 shows the unique address prefix lengths and most of the destination IP addresses belong to class C address prefixes (24-bit network address).

The problem is that if there is a rule with destination IP address field "any" and with the decision "accept", then no rules in the routing table can be removed from the TCAM. Fortunately, this seldom happens in the rule sets because of three reasons. First, some rule sets do not contain IP address with value "any" at all. Second, most rules with destination IP address field "any" also

Table 5.1: Analysis of destination IP address in Snort rules

Number	IP Addresses	Number	Percentage (%)
1	$HOME_NET	3888	63.60
2	$EXTERNAL_NET	1871	30.61
3	any	174	2.85
4	$SMTP_SERVERS	70	1.15
5	$SQL_SERVERS	55	0.90
6	$HTTP_SERVERS	40	0.65
7	$DNS_SERVERS	4	0.07
8	255.255.255.255	4	0.07
9	224.0.0.0/4	3	0.05
10	224.0.0.1	1	0.44
11	224.0.0.251	1	0.02
12	85.17.3.250	1	0.02
13	212.26.42.47	1	0.02
-	total	6113	100

Table 5.2: Number of unique address prefix lengths for source address (SA), destination address (DA), and source/destination address pairs (SA/DA).

Set	Size	SA	DA	SA/DA
acl1	733	6	20	31
acl2	623	13	13	50
acl3	2400	22	12	89
acl4	3061	22	15	98
acl5	4557	11	3	31
fw1	283	12	6	22
fw2	68	4	3	8
fw3	184	9	3	13
fw4	264	5	6	12
fw5	160	10	4	17
ipc1	1702	15	13	93
ipc2	192	4	2	5

Table 5.3: IP prefix reduction results with 4,906 original IP prefixes

Rule set	Number of rules	IP prefix	IP reduction	Save (%)
RuleSet1	500	4,906	4,815	98.14
RuleSet2	1000	4,906	4,801	97.86
RuleSet3	1500	4,906	4,534	92.42
RuleSet4	2000	4,906	4,465	91.01

Table 5.4: IP prefix reduction results with 9,812 original IP prefixes

Set	Number of rules	IP prefix	IP reduction	Save (%)
RuleSet1	500	9,812	9,729	99.15
RuleSet2	1000	9,812	9,708	98.94
RuleSet3	1500	9,812	9,179	93.55
RuleSet4	2000	9,812	9,035	92.08

combined with the decision “discard”. Last, after redundancy removal, most of the “any” will be replaced. In [72], we can see that some real rule sets do not contain any destination IP address with value “any”, and others contain a small percent of value “any” in the destination IP address field.

Another important factor that affects the number of rules can be removed from the routing table is the length of destination IP addresses. The IP addresses belong to class C address prefixes are more likely to cover less IP prefixes in the routing table than the IP addresses belong to class A address prefixes. In [72], we can see that most of the destination IP address prefixes are 32-bit. We generate four rule sets with different number of rules, when combing with different number of IP prefixes, the IP prefix reduction results are shown in Table 5.3, Table 5.4 and Table 5.5 respectively. And the comparison of IP prefix reduction is shown in Figure 5.3. We can see that we can save most of the IP prefixes in the routing lookup tables.

We select IP prefixes do not need to be stored in the TCAM in the routing table using Algorithm 9. We compare the IP prefixes with rules’ destination IP prefixes. If an IP prefix overlaps with one or multiple rules and all these rules have the same decision “discard”, then this

Table 5.5: IP prefix reduction results with 19,624 original IP prefixes

Set	Number of rules	IP prefix	IP reduction	Save (%)
RuleSet1	500	19,624	19,465	99.19
RuleSet2	1000	19,624	19,432	99.02
RuleSet3	1500	19,624	18,617	94.87
RuleSet4	2000	19,624	18,297	93.24

IP prefix does not need to be stored in the TCAM. If an IP prefix overlaps with one or multiple rules and all these rules have the same decision other than "discard" and the same output port, then this IP prefix does not need to be stored in the TCAM and the output port can be stored with these rules together. If an IP prefix does not match any previous condition, the the IP prefix needs to be stored in the TCAM.

We can not guarantee the maximum number of IP prefixes do not need to be stored in the TCAM with Algorithm 9. If two IP prefixes with different output ports overlap with a single rule's destination IP address and the rule's decision is not "discard", then these two IP prefixes compete for this rule. When multiple IP prefixes compete for some rules, we have multiple choices and they might result in different number of IP prefixes reduced. For example, there are three IP addresses and two rules in Figure 5.4, and the line between the IP prefixes and rules show the overlapping relationship between them and we can see the competition between the first IP prefix and the third IP prefix and the competition between the second IP prefix and the third IP prefix. So we have two choices, first, we remove the first and the second IP prefixes, second, we remove the third IP prefix. Obviously, we prefer the first choice. Usually shorter prefix has more chance to overlap with other prefixes because of its larger range, so we give longer IP prefix higher priority to be reduced, but this can not guarantee the maximum number of IP prefixes can be reduced.

If a rule set contains one or multiple rules cover all or many IP prefixes in the routing table, such as a rule with destination IP address field "any" and with the decision "accept", then no rules

Algorithm 10 IP Prefixes do not need to be stored in TCAM

Input : A non-overlapping rule set R: $r_1, r_2, \ldots, r_n$ and a set of non-redundant IP prefix P: $p_1, p_2, \ldots, p_n$ in order of descending prefix length
Output: A set of prefixes Q: $q_1, q_2, \ldots, q_n$
for Each IP prefix p_j in the IP prefix set P **do**
  for Each rule r_i in the rule set R **do**
    if p_j is a prefix of Destination IP in r_i or Destination IP in r_i is a prefix of p_j **then**
      Record r_i
    end if
  end for
  if All the recorded rules with the same decision "discard" **then**
    Clear all records
    Mark p_j
  else
    if All the recorded rules with the same decision and they are either appended with the same output port as p_j's or has not been appended with any output port **then**
      Appended recorded rules with p_j's output port
      Clear all records
      Mark p_j
    end if
  end if
end for
return all the marked prefixes in P

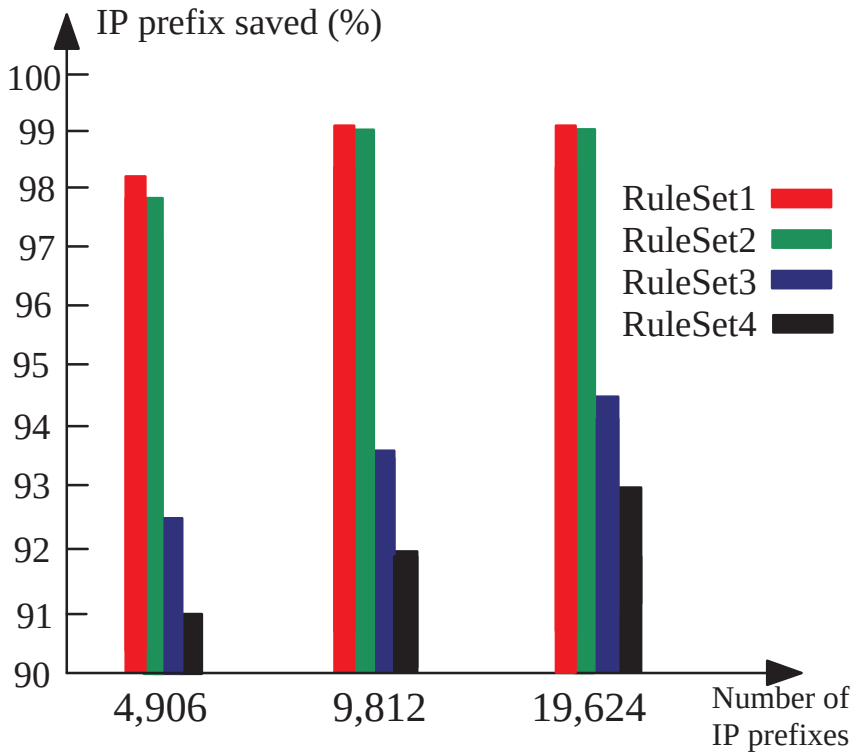


Figure 5.3: IP prefix reduction results with different numbers of original IP prefixes

in the routing table can be removed from the TCAM. In this situation, we can store all the covered IP prefixes in the TCAMs, and we can also use other two approaches. First, we perform routing table lookup in the software because we have showed that there are usually very limited number of such rules. Second, we store IP prefixes covered by such rules in another TCAM block. In both approaches, the routing table lookup process works as a hierarchical architecture: most of the packets do not need to go through the IP routing lookup TCAMs because they are encoded in the rules, some packets need to go through the fast IP routing lookup TCAM because only a small number of IP prefixes are stored in the fast IP routing lookup TCAM, and the remainder packets need to go through the slow IP routing lookup pathes, such as software or large TCAM.

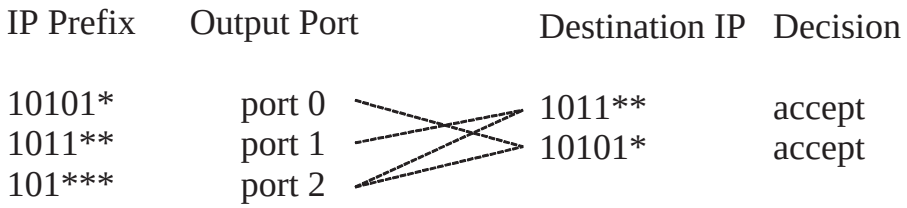


Figure 5.4: An example of multiple choices of reducing IP prefixes when combing IP prefixes and rules

Table 5.6: Hardware resource critical path reduction by using our approaches with 4,906 IP prefixes

Rule Set	Number of rules	Slices reduction	4-input LUTs reduction	BRAM reduction	Flip Flops reduction	Critical path reduction
RuleSet1	500	76.4%	51.2%	89.9%	41.3%	18.5%
RuleSet2	1000	74.7%	51.6%	88.2%	46.9%	18.9%
RuleSet3	1500	73.0%	53.2%	87.9%	44.0%	19.7%
RuleSet4	2000	78.3%	55.1%	88.4%	46.2%	20.8%

5.3 Evaluation

We implement our approach on NetFPGA [34], in order to compare with simple integration approach which consumes more hardware resource, we do simulation on Xilinx Virtex-5 TX240T FPGA. We pipeline our design to increase throughput by dividing it into three stages: TCAM match for packet classification, retrieving result in packet classification, IP routing lookup and pattern matching. We use 6423 patterns in Snort. With different number of packet classification rules and IP prefixes, the hardware resources and throughput are shown in Table 5.6, Table 5.7 and Table 5.8. We can see we can reduce hardware resource significantly and also reduce the critical path by our approaches.

Table 5.7: Hardware resource critical path reduction by using our approaches with 9,812 IP prefixes

Rule Set	Number of rules	Slices reduction	4-input LUTs reduction	BRAM reduction	Flip Flops reduction	Critical path reduction
RuleSet1	500	78.5%	52.8%	92.8%	42.8%	18.5%
RuleSet2	1000	76.2%	53.4%	91.4%	48.6%	18.9%
RuleSet3	1500	74.9%	55.9%	90.3%	45.2%	19.7%
RuleSet4	2000	81.5%	57.0%	93.1%	49.8%	20.8%

Table 5.8: Hardware resource critical path reduction by using our approaches with 19,624 IP prefixes

Rule Set	Number of rules	Slices reduction	4-input LUTs reduction	BRAM reduction	Flip Flops reduction	Critical path reduction
RuleSet1	500	79.5%	53.9%	93.0%	42.8%	18.5%
RuleSet2	1000	77.8%	54.0%	91.7%	48.6%	18.9%
RuleSet3	1500	75.7%	56.9%	90.9%	45.2%	19.7%
RuleSet4	2000	83.0%	58.2%	93.2%	49.8%	20.8%

5.4 Discussion

We did not compare the throughput our integrated architecture with other approaches because of the following reasons: first, no such existing integrated architecture we can compare with. Second, there are too many factors can affect the throughput of the integrated architecture such as the number of IP prefixes, the number of rules, the number of regular expression and the properties of them as we discussed, and the properties of incoming traffic also affect the throughput significantly. Third, it is not comparable between naive integrated architectures. Even though for naive integrated architectures, there are very different strategies such as software-based approaches and hardware-based approaches, for example, the hardware-based integrated architecture can achieve very high throughput but consumes much more hardware resources. In our integrated architecture, the bottleneck is the deep packet inspection component for regular traffic and the throughput is the throughput of the deep packet inspection component and this is also the throughput for the worst

case. In some special cases, the packet classification component could become the bottleneck of the system, such as most packets in the traffic is blocked by the packet classification component.

We showed the integration of three major packet processing components and there are still other tasks in the packet processing [123,124], such as CRC calculation [125,126], we can process these components using similar approaches.

5.5 Summary

In this section, we present the integration approach of three major components: IP routing lookup, packet classification and deep packet inspection. Our goal is to reduce hardware consumption by sharing some hardware resources, and our main approach focuses on how to share hardware resource between IP routing lookup and packet classification because both of them check packet header. We first remove redundancy in the packet classification based on the approach we have presented, and then remove IP prefixes only covered by rules with decision "discard", and also remove a IP prefix if such prefix is covered by rules with the same decision and not appended by a different output port, then append these rules with corresponding output port. Our simulation showed that this approach can reduce more than 90% IP prefixes needed to be stored in the TCAMs. And the simulation on the integration architecture showed that our approaches can reduce significant hardware resources especially TCAM entries, and also increase the overall throughput.

CHAPTER six

CONCLUSION

In this thesis, an important issue is discussed: how to design efficient packet processing devices. There are three important issues needed to be designed efficiently to support today's high speed network: IP route lookup packet classification and deep packet inspection. And these three procedures are usually implemented as different components, which make packet processing too slow to meet high-speed network's requirements. We focus on these issues and design efficient hardware-based solutions. We first design individual components and then propose a new integrated architecture which combines these three parts efficiently. Our approaches first remove the redundancy in individual components and then remove the redundancy between them.

In the IP routing lookup chapter, we first present a hybrid approach to IP route lookup using Binary CAMs to save memory usage and improve the throughput of longest prefix matching process. We treat prefixes with different lengths separately in parallel, and use different types of CAMs to take advantage of their characteristics. The simulation results show that our approach saves 57.6% of transistors, reducing the area and power consumption significantly. Then we present a novel approach to IP route lookup using TCAMs to save memory usage, increase update speed and improve the throughput of longest prefix matching process. We discussed three important issues still have not been well studied in TCAM-based IP route lookup including large RAM usage and long memory accesses, slow update process of routing table and unscalable problem. Based on the observation that all of them result from the sorted storage in TCAM entries, we store IP prefixes can be stored out of order in the first group and the remaining ones in the second group, and deal with them separately. The simulation results show that our approach solve these problems effi-

ciently: the usage of RAM has been reduced, the throughput has been increased and the update process speed also has been increased significantly compared with the traditional ways. Furthermore, the clock speed can also be increased because the memory access latency is reduced and the priority encoders are simplified. Our main contribution of this section is to present an approach to solve multiple major problems in TCAM-based applications, not only try to solve one problem but usually worsen the others.

In the packet classification design chapter, we first propose a tree-based overlapping removal algorithm to remove redundant rules and combine overlaying rules to build new rule sets in packet classifiers, and then remove a set of rules with the same decision based on the non-overlapping rule set. Based on the non-overlapping new rules, we can further reduce the TCAM consumption by range extension approach. Our experiments show a reduction of 85.4% in the number of TCAM entries after performing the two steps. We also propose a fast TCAM update scheme which enables out of order storage in the TCAM and reduces the TCAM entries usage. We also propose an approach to modify the TCAM entry itself. We propose energy efficient Comparator Content Addressable Memories (CCAMs) to solve the range expansion problem and reduce the rule set update delay. Our approach can store ranges in CCAM entries besides ternary state bits. In our algorithm, the hardware can be configured to store all kinds of port ranges. Simulations show that CCAMs consume about 27.6% of what the original TCAMs do in terms of the number of entries. As for transistor consumption, our approach saves about 66.4% of transistors compared with TCAM-based approaches.

In the deep packet inspection chapter, we first present an effective technique for pattern matching in deep packet inspection. We build a hierarchical pattern matching architecture which serves to exclude most packets from full pattern matching leaving only a small percentage to be fully checked in the pattern matching process. Our approach here may offer a key lever for im-

proving the throughput of regular expression matching through extracting exact-match strings from the regular expressions and preprocessing the matching dependent on those strings. We also analyze techniques used in deep packet inspection, in particular regular expression matching with growing complexity. We build efficient NFA/DFA generators and investigated the impact of the complexity of regular expressions using rule sets from actual NIDS. We demonstrate that a singe NFA or a singe DFA performs poorly given a large set of complex regular expressions. Because multi-core processors are expected to dominate, there will be great demand on fast deep packet inspection algorithms that can utilize multi-core architecture for complex regular expressions. Thus, we propose a hybrid algorithm that combines both NFA and DFA. It divides a complex regular expression and configures them into multiple cores to take advantage of available parallelism provided by multi-core processors. The evaluation results show that the hybrid approach outperforms the single DFA and single NFA approaches. We finally propose an efficient regular expression matching in deep packet inspection for compressed traffic. The simulation results show that our approach effectively skips most of the compressed parts in the traffic and reduces the frequency of DFA state accesses in proportion to the traffic's compression ratio. Although DFA is not suitable for a large number of complex regular expressions [106], our approach can be easily extended to be used in separate, multiple small DFAs, which will make our approach more scalable.

In the integrated architecture chapter, we present the integration approach of three major components: IP routing lookup, packet classification and deep packet inspection. Our goal is to reduce hardware consumption by sharing some hardware resources, and our main approach focuses on how to share hardware resource between IP routing lookup and packet classification because both of them check packet header. We first remove redundancy in the packet classification based on the approach we have presented, and then remove IP prefixes only covered by rules with decision "discard", and also remove a IP prefix if such prefix is covered by rules with the same decision

and not appended by a different output port, then append these rules with corresponding output port. Our simulation showed that this approach can reduce more than 90% IP prefixes needed to be stored in the TCAMs. And the simulation on the integration architecture showed that our approaches can reduce significant hardware resources especially TCAM entries, and also increase the overall throughput.

Our approaches are mainly based on TCAMs and focus on how to reduce the TCAM entry consumption based on given data. Some may argue that the price of TCAM is dropping and we do not need to compress data before storing it into TCAM in the future, but we believe the approaches we presented in the thesis is still important in that situation for the following reasons. First, the usage of TCAMs is limited mainly by its cost, so the TCAMs will be much more widely used if its price is competitive, and how to reduce the TCAM entry is also important because of the huge amount of TCAM usage. Second, the number of data and the length of data is also increase, such as the number of IP prefixes we showed and the shift from IPv4 to IPv6, so the amount of TCAM required by a single device is also increasing fast. Third, the power consumption is another important disadvantage of TCAMs and this problem will not be mitigated and how to reduce the TCAM usage is also important. Last but not least, some disadvantages of TCAM we discussed will not be mitigated with the dropping price, such as the slow update and slow and large priority encoder, and our approaches can solve these problems efficiently.

BIBLIOGRAPHY

[1] Yan Sun and Min Sik Kim. Ip prefix matching with binary and ternary CAMs. In *Proceedings of the 7th IEEE Consumer Communications and Networking Conference*, January 2010.

[2] Yan Sun, Haiqin Liu, and Min Sik Kim. Using TCAM efficiently for IP route lookup. In *roceedings of the 8th IEEE Consumer Communications and Networking Conference, CCNC'11*, pages 816–817, January 2011.

[3] Yan Sun and Min Sik Kim. Tree-based minimization of TCAM entries for packet classification. In *Proceedings of IEEE Consumer Communications & Networking Conference*, January 2010.

[4] Yan Sun and Min Sik Kim. Bidirectional range extension for TCAM-based packet classification. In *Proceedings of the IFIP/TC6 NETWORKING 2010*, May 2010.

[5] Yan Sun, Haiqin Liu, and Min Sik Kim. Hierarchical NFA-based pattern matching for deep packet inspection. In *Proceedings of the 20th International Conference on Computer Communications and Networks, ICCCN'11*, July 2011.

[6] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *Proceedings of IEEE INFOCOM*, pages 1454–1463, April 2001.

[7] Zhuo Huang, David Lin, Shigang Chen, Jih-Kwon Peir, and S. M. Iftekharul Alam. Fast routing table lookup based on deterministic multi-hashing. In *Proceedings of the 18th IEEE International Conference on Network Protocols*, October 2010.

[8] S. Kumar J. Turner P. Crowley. Peacock Hashing: Deterministic and updatable hashing for high performance networking. In *Proceedings of IEEE INFOCOM*, pages 101 – 105, April 2008.

[9] Socrates Demetriades, Michel Hanna, Sangyeun Cho, and Rami Melhem. An efficient hardware-based multi-hash scheme for high speed IP lookup. In *Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*, pages 103–110, August 2008.

[10] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor. Longest prefix matching using bloom filters. *IEEE/ACM Transactions on Networking*, 14:397–409, April 2006.

[11] Haoyu Song, Fang Hao, M. Kodialam, and T.V. Lakshman. IPv6 lookups using distributed and load balanced bloom filters for 100Gbps core router line cards. In *Proceedings of IEEE INFOCOM*, pages 2518 – 2526, April 2009.

[12] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *SIGCOMM Computer and Communication Review*, 35:181–192, August 2005.

[13] Heeyeol Yu, R. Mahapatra, and L. Bhuyan. A hash-based scalable IP lookup using bloom and fingerprint filters. In *Proceedings of the 17th IEEE International Conference on Network Protocols*, Oct 2009.

[14] N. F. Tzeng. Routing table partitioning for speedy packet lookups in scalable routers. *IEEE Transactions on Parallel and Distributed Systems*, 17(5):481–494, May 2006.

[15] Y. K. Chang and Y. C. Lin. A fast and memory efficient dynamic IP lookup algorithm based on B-Tree. In *Proceedings of International Conference on Advanced Information Networking and Applications*, volume 0, pages 278–284, May 2009.

[16] L. C. Wuu, T. J. Liu, and K. M. Chen. A longest prefix first search tree for IP lookup. *Computer Networks*, 51(12):3354–3367, August 2007.

[17] X. H. Sun and Y. Q. Zhao. An on-chip IP address lookup algorithm. *IEEE Transaction on Computers*, 54(7):873–885, July 2005.

[18] R. Rojas-Cessa, L. Ramesh, Dong Ziqian, Cai Lin, and N. Ansari. Parallel search trie-based scheme for fast IP lookup. In *Proceedings of IEEE Global Telecommunications Conference*, pages 210–214, November 2007.

[19] O. Erdem and C. F. Bazlamacci. Array design for trie-based IP lookup. *IEEE Communications Letters*, 14:773–775, 2010.

[20] Fong Pong and Nian-Feng Tzeng. SUSE: Superior storage-efficiency for routing tables through prefix transformation and aggregation. *IEEE/ACM Transactions on Networking*, 18:81–94, 2010.

[21] H. Liu. Reducing routing table size using Ternary-CAM. In *Proceedings of the The Ninth Symposium on High Performance Interconnects*, page 69, 2001.

[22] S. Stergiou and J. Jain. Optimizing routing tables on systems-on-chip with contentcaddressable memories. In *Proceedings of International Symposium on System-on-Chip*, pages 1–6, November 2008.

[23] Y. Tang, W. Lin, and B. Liu. A TCAM index scheme for IP address lookup. In *Proceedings of First International Conference on Communications and Networking in China*, volume 0, pages 1–5, October 2006.

[24] Kai Zheng, Zhen Liu, and Bin Liu. High performance embedded route lookup coprocessor for network processors. 3619:188–197, September 2005.

[25] H. Noda, K. Inoue, M. Kuroiwa, F. Igaue, K. Yamamoto, H.J. Mattausch, T. Koide, A. Amo, A. Hachisuka, S. Soeda, I. Hayashi, F. Morishita K. Dosaka, K. Arimoto, K. Fujishima, K. Anami, and T. Yoshihara. A cost-efficient high-performance dynamic TCAM with pipelined hierarchical searching and shift redundancy architecture. *IEEE Journal of Solid-State Circuits*, 40:245–253, January 2005.

[26] Yan Sun and Min Sik Kim. A hybrid approach to CAM-based longest prefix matching for IP route lookup. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM 2010)*, December 2010.

[27] H. Noda, K. Inoue, M. Kuroiwa, A. Amo, A. Hachisuka, H. J. Mattausch, T. Koide, S. Soeda, K. Dosaka, and K. Arimoto. A 143mhz 1.1w 4.5mb dynamic TCAM with hierarchical searching and shift redundancy architecture. In *Proceedings of International Conference on Solid-State Circuits*, pages 208–209, February 2004.

[28] S. Choi, K. Sohn, M. W. Lee, S. Kim, H. M. Choi, D. Kim, U. R. Cho, H. G. Byun, Y. S. Shin, and H.J. Yoo. A 0.7fj/bit/search, 2.2ns search time hybrid type TCAM architecture. *IEEE Journal of Solid-State Circuits*, 40(1):254–260, January 2005.

[29] J. S. Wang, H. Y. Li, C. C. Chen, and C. W. Yeh. An AND-type match-line scheme for energy-efficient content addressable memories. In *Proceedings of International Conference on Solid-State Circuits*, pages 464–610, February 2005.

[30] University of Oregon Route Views Project. `http://www.routeviews.org/`.

[31] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

[32] Deepak S Vijayasarathi, Mehrdad Nourani, Mohammad J. Akhbarizadeh, and Poras T. Balsara. Ripple-precharge TCAM a low-power solution for network search engines. iccd. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 243–248, Oct 2005.

[33] S. Abdel-hafeez and S. Harb. A VLSI high-performance priority encoder using standard CMOS library. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 52:597–601, 2006.

[34] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. NetFPGA: an open platform for teaching how to build gigabit-rate network switches and routers. *IEEE Transactions on Education*, 51(3):364–369, August 2008.

[35] QDR II SRAM interface for Virtex-5 devices. `http://www.xilinx.com/support/documentation/application_notes/xapp853.pdf`.

[36] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

[37] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proceedings of ACM SIGCOMM '05*, pages 193–204, August 2005.

[38] N. Mohan, W. Fung, and M. Sachdev. Low-power priority encoder and multiple match detection circuit for ternary content addressable memory. In *Proceedings of IEEE International SOC Conference*, pages 24–27, September 2006.

[39] Miad Faezipour and Mehrdad Nourani. Wire-speed TCAM-based architectures for multi-match packet classification. *IEEE Transactions on Computers*, 58(1):5–17, January 2009.

[40] S. Abdel-hafeez and S. Harb. A VLSI high-performance priority encoder using standard CMOS library. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 53(8):597–601, August 2006.

[41] Huan Liu. Efficient mapping of range classifier into ternary-CAM. In *Proceedings of the 10th Symposium on High Performance Interconnects*, pages 95–100, August 2002.

[42] Jan van Lunteren and Ton Engbersen. Fast and scalable packet classification. *IEEE Journal on Selected Areas in Communications*, 21(4):560–571, May 2003.

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

[44] M. Faezipour and M. Nourani. CAM01-1: A customized TCAM architecture for multi-match packet classification. In *IEEE Global Telecommunications Conference*, pages 1–5, November 2006.

[45] Hao Che, Zhijun Wang, Kai Zheng, and Bin Liu. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Transactions on Computers*, 57(7):902–915, July 2008.

[46] Young-Deok Kim, Hyun-Seok Ahn, Suhwan Kim, and Deog-Kyoon Jeong. A high-speed range-matching TCAM for storage-efficient packet classification. *IEEE Transactions on Circuits and Systems Part I Regular Papers*, 56(6):1221–1230, June 2009.

[47] Xia Deng, Zhiping Huang, Shaojing Su, Chunwu Liu, Guilin Tang, and Yimeng Zhang. High-speed packet classification with efficient parallel range match for IP network applications. In *Proceedings of the 2008 International Conference on MultiMedia and Information Technology*, pages 22–25, December 2008.

[48] A. Bremler-Barr and D. Hendler. Space-efficient TCAM-based classification using gray coding. In *Proceedings of the 26th IEEE International Conference on Computer Communications*, pages 6–12, May 2007.

[49] Hao Che, Zhijun Wang, Kai Zheng, and Bin Liu. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Transactions on Computers*, 57(7):902–915, July 2008.

[50] Kai Zheng, Hao Che, Zhijun Wang, Bin Liu, and Xin Zhang. Dppc-re: Tcam-based distributed parallel packet classification with range encoding. *IEEE Transaction on Computer*, 55(8):947–961, 2006.

[51] D. Pao, P. Zhou, B. Liu, and X. Zhang. Enhanced prefix inclusion coding filter-encoding algorithm for packet classification with ternary content addressable memory. *IET Computers and Digital Techniques*, 1:572–580, 2007.

[52] Chad R. Meiners, Alex X. Liu, and Eric Torng. Topological transformation approaches to optimizing TCAM-based packet classification systems. In *Proceedings of ACM SIGMETRICS*, pages 73–84, June 2009.

[53] Florin Baboescu, Sumeet Singh, and George Varghese. Packet classification for core routers: Is there an alternative to CAMs? In *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications*, pages 53–63, March 2003.

[54] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of ACM SIGCOMM '03*, pages 213–224, August 2003.

[55] A. X. Liu, C. R. Meiners, and Y. Zhou. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proceedings of the 27th IEEE INFOCOM*, pages 111–115, April 2008.

[56] David A. Applegate, Gruia Calinescu, David S. Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. Compressing rectilinear pictures and minimizing access control lists. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1066–1075, January 2007.

[57] Qunfeng Dong, Suman Banerjee, Jia Wang, Dheeraj Agrawal, and Ashutosh Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proceedings of ACM SIGMETRICS/Performance 2006*, pages 311–322, June 2006.

[58] Kai Zheng, Hao Che, Zhijun Wang, and Bin Liu. TCAM-based distributed parallel packet classification algorithm with range-matching solution. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 293–303, March 2005.

[59] Derek Pao, Yiu Keung Li, and Peng Zhou. Efficient packet classification using TCAMs. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 50(18):3523–3535, December 2006.

[60] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *17th IEEE International Conference on Network Protocols*, October 2009.

[61] Yeim-Kuan Chang, Chun-I Lee, and Cheng-Chien Su. Multi-field range encoding for packet classification in TCAM. In *Proceedings of the IEEE INFOCOM 2011*, pages 196–200, April 2011.

[62] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. ACM.

[63] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. EffiCuts: optimizing packet classification for memory and throughput. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 207–218, New York, NY, USA, 2010. ACM.

[64] Yan Luo, Ke Xiang, and Sanping Li. Acceleration of decision tree searching for IP traffic classification. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 40–49, New York, NY, USA, 2008. ACM.

[65] Derek Pao and Cutson Liu. Parallel tree search: An algorithmic approach for multi-field packet classification. *Computer Communications*, 30:302–314, January 2007.

[66] Fong Pong and Nian-Feng Tzeng. HaRP: Rapid packet classification via hashing round-down prefixes. *IEEE Transactions on Parallel and Distributed Systems*, 22:1105–1119, 2011.

[67] Fong Pong and Nian-Feng Tzeng. Hashing round-down prefixes for rapid packet classification. In *Proceedings of the USENIX Annual Technical Conference*, pages 71–85, 2009.

[68] Lynn Choi, Hyogon Kim, Sunil Kim, and Moon Hae Kim. Scalable packet classification through rulebase partitioning using the maximum entropy hashing. *IEEE/ACM Transaction on Networking*, 17:1926–1935, December 2009.

[69] Pankaj Gupta and Nick Mckeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, March 2001.

[70] Atsushi Yoshioka, Shariful Hasan Shaikot, and Min Sik Kim. Rule hashing for efficient packet classification in network intrusion detection. In *Proceedings of the 17th IEEE International Conference on Computer Communications and Networks*, August 2008.

[71] Yan Sun, Xin Zhang, and Xi Jin. High-performance carry select adder using fast all-one finding logic. In *Proceedings of the 2nd Asia International Conference on Modeling & Simulation*, May 2008.

[72] David E. Taylor and Jonathan S. Turner. ClassBench: a packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15:499–511, June 2007.

[73] Alex X. Liu and Mohamed G. Gouda. Complete redundancy removal for packet classifiers in TCAMs. *IEEE Transactions on Parallel and Distributed Systems*, 21:424–437, April 2010.

[74] Xilinx Corporation. *Xilinx DS253 Content-Addressable Memory v6.1, Data Sheet*, September 2008. Available as `http://www.xilinx.com/support/documentation/ip_documentation/cam_ds253.pdf`.

[75] Devavrat Shah and Pankaj Gupta. Fast updating algorithms for TCAMs. *IEEE Micro*, 21:36–4, January 2001.

[76] Yi Mao Hsiao, Ming Jen Chen, Yu Jen Hsiao, Hui Kai Su, and Yuan Sun Chu. A fast update scheme for TCAM-based IPv6 routing lookup architecture. In *Proceedings of the 15th Asian-Pacific Conference on Communications*, pages 1–5, October 2009.

[77] Weidong Wu and Ruixuan Wang. A TCAM management scheme for ip lookups. In *Proceedings of the 14th IEEE International Conference on Networks*, pages 1–4, September 2006.

[78] David E. Taylor and Jonathan S. Turner. ClassBench: A packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, June 2007.

[79] The Snort Project. *Snort User Manual 2.8.6*, April 2010. http://www.snort.org/assets/140/snort_manual_2_8_6.pdf.

[80] A. Bremler-Barr, D. Hay, and Y. Koral. CompactDFA: Generic state machine compression for scalable pattern matching. In *Proceedings of the 29th conference on Information Communications, INFOCOM'10*, pages 659–667, 2010.

[81] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, December 1999.

[82] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proceedings of the 11th ACM Conference on Computer and Comm. Security*, October 2004.

[83] Cisco IOS IPS signature deployment guide. http://www.cisco.com/.

[84] Marco Paolieri, Ivano Bonesana, and Marco Domenico Santambrogio. ReCPU: a parallel and pipelined architecture for regular expression matching. In *Proceedings of 15th Annual IFIP International Conference on Very Large Scale Integration*, pages 19–24, October 2007.

[85] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of ACM CoNEXT*, December 2007.

[86] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 155–164, December 2007.

[87] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, December 2006.

[88] Tippingpoint x505. http://www.tippingpoint.com/products_ips.html.

[89] The Snort Project. *Snort User Manual 2.8.5*, September 2009. Available as http://www.snort.org/assets/120/snort_manual.pdf.

[90] Wei Lin and Bin Liu. Pipelined parallel AC-based approach for multi-string matching. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems, ICPADS'08*, pages 665–672, December 2008.

[91] C. P. Jiang C. H. Lin, C. T. Huang and S. C. Chang. Optimization of pattern matching circuits for regular expression on fpga. *IEEE Transactions on Very Large Scale Integration Systems*, 15(12):1303–1310, December 2007.

[92] I. Bonesana, M. Paolieri, and M.D. Santambrogio. An adaptable FPGA-based system for regular expression matching. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 1262–1267, March 2008.

[93] N. Yamagaki, R. Sidhu, and S. Kamiya. High-speed regular expression matching engine using multi-character NFA. In *International Conference on Field Programmable Logic and Applications*, pages 131–136, September 2008.

[94] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 31–38, April 2003.

[95] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns. In *Conference on field-programmable logic and applications*, pages 956–959, September 2003.

[96] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. In *in IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, 2001.

[97] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *Processing of IEEE Symposium On Field-Programming Custom Computing Machines*, pages 258–267, April 2004.

[98] S. Yusuf and W. Luk. Bitwise optimised CAM for network intrusion detection systems. In *Processing of IEEE Symposium On Field-Programming Custom Computing Machines*, pages 444–449, August 2005.

[99] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 127–136, 2007.

[100] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *In IEEE Symposium on Security and Privacy*, pages 187–201, May 2008.

[101] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved DFA for fast regular expression matching. *Computer Communication Review*, 38(5):29–40, 2008.

[102] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154, 2007.

[103] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 81–92, 2006.

[104] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proceedings of the 26th IEEE International Conference on Computer Communications*, pages 29–40, May 2007.

[105] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, June 2006.

[106] Yan Sun, Haiqin Liu, V.C. Valgenti, and Min Sik Kim. Hybrid regular expression matching for deep packet inspection on multi-core architecture. In *Proceedings of the 19th International Conference on Computer Communications and Networks, ICCCN'10*, August 2010.

[107] Yan Sun and Min Sik Kim. DFA-based regular expression matching on compressed traffic. In *Proceedings of the 2011 IEEE International Conference on Communications, ICC'11*, May 2011.

[108] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 2009 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, October 2009.

[109] M. Alicherry, M. Muthuprasanna, and V. Kumar. High speed pattern matching for network IDS/IPS. In *Proceedings of the 14th IEEE International Conference on Network Protocols, ICNP'06*, pages 187–196, October 2006.

[110] F. Yu, H. R. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols, ICNP'04*, pages 174–183, October 2004.

[111] Anat Bremler-Barr and Yaron Koral. Accelerating multi-patterns matching on compressed HTTP traffic. In *Proceedings of the IEEE INFOCOM 2009*, pages 397–405, April 2009.

[112] Clam AntiVirus. *ClamAV*. `http://www.clamav.net/lang/en/`.

[113] Sourcefire Vulnerability Research Team. *Sourcefire Vulnerability Research Team (VRT) Snort Rule-set*, 2.8.6 edition, September 2010. Available at `http://www.snort.org/vrt`.

[114] Intel multi-core technology. `http://www.intel.com/multi-core/`.

[115] Multi-core processors: The next evolution in computing. `http://multicore.amd.com/Resources/33211A_Multi-Core_WP_en.pdf`.

[116] Coherent Processing System (CPS): Multi-threaded multiprocessor IP cores. `http://www.mips.com/`.

[117] The ARM11 MPCore is synthesizable. `http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html`.

[118] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[119] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[120] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, pages 337–343, May 1977.

[121] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 11th ACM Conference on Computer and Communication Security*, October 2004.

[122] J. E. Hopcroft and J. D. Ullman. Introduction to automata theory, languages, and computation. *Addison Wesley*, 1979.

[123] Yan Sun and Min Sik Kim. Energy-efficient routing protocol in event-driven wireless sensor networks. In *Proceedings of the 2011 IEEE ICC Workshop on Energy Efficiency in Wireless Networks & Wireless Networks for Energy Efficiency*, May 2011.

[124] Yan Sun and Min Sik Kim. A high-performance 8-tap FIR filter using logarithmic number system. In *Proceedings of the 2011 IEEE International Conference on Communications, ICC'11*, June 2011.

[125] Yan Sun and Min Sik Kim. A table-based algorithm for pipelined CRC calculation. In *Proceedings of IEEE International Conference on Communications*, 2010.

[126] Yan Sun and Min Sik Kim. A pipelined CRC calculation using lookup tables. In *Proceedings of the 7th IEEE Consumer Communications and Networking Conference*, January 2010.