Επέκταση Μεταβλητού Μεγέθους στην Αρχιτεκτονική Αναγνώρισης Προτύπων HashMem

Διπλωματική Εργασία

Άγγελος Αρελάκης

Πολυτεχνείο Κρήτης
Τμήμα Ηλεκτρονικών Μηχανικών & Μηχανικών Υπολογιστών
Εργαστήριο Μικροεπεξεργαστών και Υλικού (MHL)

Επιτροπή:

Αναπληρωτής Καθηγητής Διονύσιος Πνευματικάτος (Επιβλέπων)
Καθηγητής Απόστολος Δόλλας
Επίκουρος Καθηγητής Ιωάννης Παπαευσταθίου

ΧΑΝΙΑ, ΙΟΥΛΙΟΣ 2006

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέπων καθηγητή κ. Διονύσιο Πνευματικάτο για την πολύτιμη βοήθεια του, τις συμβουλές του και την καθοδήγηση του. Επίσης θα ήθελα να τον ευχαριστήσω για την άριστη συνεργασία που είχαμε.

Θα ήθελα επίσης να ευχαριστήσω τους καθηγητές κ. Διονύσιο Πνευματικάτο, κ. Απόστολο Δόλλα, κ. Ιωάννη Παπαευσταθίου για το σημαντικό ρόλο που έπαιξαν στη ζωή μου έτσι ώστε να ασχοληθώ με το Hardware.

Επιπλέον, είμαι ευγνώμων στο Διονύση Ευσταθίου για την πολύτιμη βοήθεια του όταν έκανα τα πρώτα βήματα στο Hardware κατά τη διάρκεια των σπουδών μου. Επίσης θα ήθελα να ευχαριστήσω τον κ. Μάρκο Κιμιωνή για την βοήθεια του. Πολλά ευχαριστώ σε όλα τη μέλη του εργαστηρίου για τη βοήθεια και τη συμπαράσταση τους.

Τέλος θα ήθελα να ευχαριστήσω θερμά τους γονείς μου και την αδερφή μου για την αμέριστη συμπαράσταση που μου έδειξαν και το συγκάτοικο μου για την υπομονή του. Πολλά ευχαριστώ επίσης στους φίλους μου Σωτηρία και Γιώργο για την υποστήριξη και υπομονή τους.

στους γονείς μου και την αδερφή μου, Στέλλα…

Σύνοψη

Ως ανίχνευση εισβολέων ορίζεται η διαδικασία ανίχνευσης ενός εχθρικού πακέτου (σε ένα δίκτυο) το οποίο προσπαθεί να αποκτήσει μη επιτρεπτή πρόσβαση σε ένα ξένο υπολογιστή. Υπάρχει μια πληθώρα μεθόδων για την ανίχνευση εισβολέων, όπως για παράδειγμα συστήματα επιθεώρησης (inspecting systems), firewalls, router logs και άλλα. Τα συστήματα ανίχνευσης εισβολέων σε ένα Δίκτυο (Network Intrusion Detection Systems - NIDS) έχουν σχεδιαστεί να επιθεωρούν την κίνηση του Δικτύου και να ψάχνουν για γνωστά patterns.

Σε αυτήν την εργασία επεκτείνεται η αρχιτεκτονική αναγνώρισης προτύπων HashMem ώστε να επιτρέπεται η αποθήκευση patterns μεταβλητού μεγέθους σε μία μόνο μνήμη μειώνοντας τον απαιτούμενο αριθμό μνήμων και συγκριτών. Η HashMem είναι μία αρχιτεκτονική αναγνώρισης προτύπων για ανίχνευση εισβολέων τύπου SNORT η οποία βασίζεται σε μνήμες. Χρησιμοποιεί συναρτήσεις τύπου κυκλικής αναφοράς (Cyclic Redundancy Check - CRC) για να καθορίσει μια μοναδική διεύθυνση στην οποία είναι αποθηκευμένο ένα pattern. Στη συνέχεια το pattern αυτό στέλνεται σε ένα συγκριτή και συγκρίνεται με την είσοδο σε περίπτωση που μοιάζουν. Με αυτήν την επέκταση βελτιώνουμε την πυκνότητα των μνημών (αυξάνεται ο αριθμός των αποθηκευμένων patterns στο συνολικό αριθμό καταχωρήσεων) και μειώνουμε την απαραίτητη λογική για συναρτήσεις CRC και συγκριτές.

Αυτές οι βελτιώσεις επιτρέπουν στην V-HashMem να αποθηκεύσει το νεότερο σύνολο κανόνων του SNORT χρησιμοποιώντας περίπου την ίδια μνήμη με τη HashMem και με πολύ χαμηλό κόστος (~0.06 LCs/ character). Το κόστος λογικής για κάθε χαρακτήρα, είναι σχεδόν μια τάξη μεγέθους μικρότερο από το κόστος λογικής άλλων μελετών και το Throughput περίπου 2.5 Gbps. Η αρχιτεκτονική μεταβλητού μεγέθους χρησιμοποιεί μονο-θύρες μνήμες και συνεπώς επιτρέπεται και η ταυτόχρονη επεξεργασία 2 χαρακτήρων ανά κύκλο χρησιμοποιώντας τις δίθυρες μνήμες της τεχνολογίας FPGA και επιπλέον λογική. Με αυτήν την βελτιστοποίηση η αρχιτεκτονική V-HashMem επιτυγχάνει σχεδόν διπλάσιο Throughput (περίπου 5 Gbps). Τέλος, επεκτείνουμε την αρχιτεκτονική V-HashMem ώστε να συμπεριλάβει το πεδίο *Header-ID* έτσι ώστε να υποστηρίζεται η δυνατότητα για header matching, χαρακτηριστικό το οποίο λείπει από τη HashMem και πολλές άλλες έρευνες.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The widespread use of Internet is obvious in the last decades. Millions of people are using it every day, since it has improved their life. It is considered as the biggest library; the communication is rapid through it and since the middle 90's it has been used for shopping. However, many threats lie in the background and are ready to attack Internet users taking advantage of the vulnerabilities of operating systems. Furthermore, very often more than one computer are connected on the same network as there are many advantages, such as the use of one peripheral device by many computers (printer), sharing of the same Internet Connection, easy transfer of data between different computers, etc, making the protection from these threats more complex. The solution came with the construction of Firewalls.

The Firewall is defined as a piece of hardware or a software program which functions in a networked environment to prevent some communications forbidden by the security policy, analogous to the function of firewalls in building construction. The firewall has the basic task of controlling traffic between different zones of trust. Typical zones of trust include the Internet (a zone with no trust) and an internal network (a zone with high trust). The ultimate goal is to provide controlled connectivity between zones of differing trust levels through the enforcement of a security policy and connectivity model based on the least privilege principle.

However, total reliance on the firewall tool, may provide a false sense of security. Very often we hear people say: "We have a firewall in place and therefore our network must be secure". This is a myth which has been generated especially by firewall marketing companies to more efficiently promote their goods. The firewall does not work alone as it is not a panacea. Human intervention is also required to decide how to screen traffic and "instruct" the firewall to accept or deny incoming packets. It is de facto a complex and sensitive task. Just a single security policy rule established for the wrong reasons can lead to a system being vulnerable to outside attackers. Once must also remember, that a poorly configured firewall may worsen the system's effective immunity to attacks. For these reasons Intrusion Detection Systems are used. They are the first line of defence (behind firewalls).

For purposes of simplicity we can say that Intrusion Detection System is a system that detects burglary attempts. If one wishes to compare to a home anti-burglary system, firewalls perform the role of door and window locks. These types of locks will stop the majority of burglars but sophisticated intruders may circumvent security devices that protect an intended target i.e. a home. Therefore, most people use a combination of sophisticated locks with alarm systems. An IDS performs the role of such an alarm system.

Furthermore, networks evolved all the time as a result their speed is increased and consequently it is easier for malicious packets to come in. NIDS (Network IDS) prevent these malicious packets from entering the computer system, since they perform deep packet inspection in order to verify that it is not a known threat. That is another reason why NIDS are very important for the protection. Many NIDS have been constructed either on Software either on Hardware.

A Hardware – based NIDS and more specifically an FPGA – based NIDS is the Variable – Length HashMem architecture. The idea was to extend the HashMem architecture to allow storing of *variable*-length patterns in a *single* memory structure, reducing the number of required memory structures and comparators. HashMem is a memory based, exact pattern matching architecture for SNORT-like intrusion detection. It uses CRC-style functions to determine a unique location for a possible match and then matches the input against the pattern stored in the specified memory location. By this extension, we improve the density of the memories and reduce the necessary logic for CRC functions and comparators. V-HashMem architecture was implemented in XILINX FPGAs like VirtexIIpro, Virtex4 and Spartan3. Also, it fits comfortably in small or medium FPGAs since a few tens of memory blocks and about 0.06 Logic Cells per character are finally used.

This dissertation is organized as follows: In chapter 2 we are discussing about Network Intrusion Detection Systems in more detail and about SNORT intrusion detection system. In chapter 3, we describe HashMem Architecture in short and then the procedure we follow to determine the final structure of V-HashMem Architecture. In chapter 4, we describe the V-HashMem Architecture (Design and Implementation) and make further improvements. In addition, in chapter 5 we evaluate our system and give to the readers results about the performance, the cost and the utilization of our design. In the same chapter, we compare our work with HashMem as well as with other related works. Finally in chapter 6 we give our conclusions and suggest some issues for future work.

Chapter 2

Network Intrusion Detection Systems (NIDS)

Intrusion Detection is one of the hottest areas in the information security landscape. Intrusion Detection is defined as the act of detecting a hostile network packet which is attempting to gain unauthorized access. A number of popular methods are used to detect intruders, such as inspecting systems, firewalls and router logs for hostile or unusual activities. Although, these methods are helpful, they become difficult, if not possible, to perform on a daily basis.

The roots of modern-day intrusion detection systems lie in the Intrusion Detection Expert System (IDES) and Distributed Intrusion Detection System (DIDS) models that were developed by the U.S. Department of Defense in the late 80s and early 90s [22]. These were some of the first automated systems to be deployed. Today, most intrusion detection systems are designed with the same goal: to help automate the process of looking for intruders. This can be as simple as the real-time parsing of firewall logs looking for port scans, or as complex as applying inspection routines to raw network traffic looking for buffer overflow attempts.

There are basically two models of Intrusion Detection Systems: 1) network-based intrusion detection systems (NIDS) and 2) host-based intrusion detection systems (HIDS). Many other intrusion detection models do exist but the above two are the most important.

Network – based Intrusion Detection Systems (NIDS) are designed to inspect network traffic and look for known attack patterns or "signatures". They perform this task by examining each and every packet that traverses the monitored network segment. In short, NIDS are looking for a substring within a stream of data carried by network packets. If they find this substring, they identify those network packets as vehicles of an attack. On the other hand, HIDS require agents to be installed in all monitored systems. They monitor system logs for basic events (i.e. failed login attempts) and kernel messages for activities that might be interpreted as hostile.

One of the most appealing aspects of the NIDS model is that NIDS devices are passive. In most cases, the rest of the systems do not even know that they are there. Even better, deploying NIDS devices do not require the involvement of system administrators, a resource that becomes a stumbling point for large HIDS deployments.

In this chapter we will discuss about SNORT and Software and Hardware – based NIDS.

2.1 SNORT Network Intrusion Detection System

There are a great number of open source IDS solutions in the community that are worth investigating. The best and the most popular of them is SNORT NIDS, which was created by Marty Roesch, and is capable of performing real-time traffic analysis and packet logging on IP networks. It can perform protocol analysis, content searching/matching and can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, OS fingerprinting attempts, and much more. SNORT is often considered to be the Linux of the Intrusion Detection Field. It touts a very active development community, a wide set of signatures and a large base of deployed users. Recent advances in both the rules language and detection capabilities offer the most flexible and accurate threat detection available, making Snort the "heavyweight" champion of intrusion prevention.

Snort uses a flexible rules language to describe traffic that it should collect or pass. It has three primary uses. It can be used as a straight packet sniffer like tcpdump, a packet logger (useful for network traffic debugging, etc), or as a full blown network intrusion prevention system. Snort rules are powerful, flexible and relatively easy to write, so new rules to detect the latest malware may be written by everyone very easily. In these rules a keyword modifier can be added based on what kind of search the rule writer want to do. Some of these keywords are:

- Depth: the depth keyword allows the rule writer to specify how far into a packet Snort should search for the specified pattern.
- Offset: the offset keyword allows the rule writer to specify where to start searching for a pattern within a packet.

- Distance: the distance keyword allows the rule writer to specify how far into a packet Snort should ignore before starting to search for the specified pattern relative to the end of the previous pattern match. This can be thought of as exactly the same thing as depth except it is relative to the end of the last pattern match instead of the beginning of the packet.
- Within: the within keyword is a content modifier that makes sure that at most N bytes are between pattern matches It is designed to be used in conjunction with the distance rule option.

An example which combines the first 2 options (Depth and Offset) is shown in figure 2.1. The keyword content is the most important because it allows the user to set rules.

```
alert tcp any any -> any 80 (content: "cgi-bin/phf"; offset:4; depth:20;)
```

Figure 2.1**:** Combined Content, Offset and Depth Rule. The rule says: Skip the first 4 bytes, and look for cgi-bin/phf in the next 20 bytes.

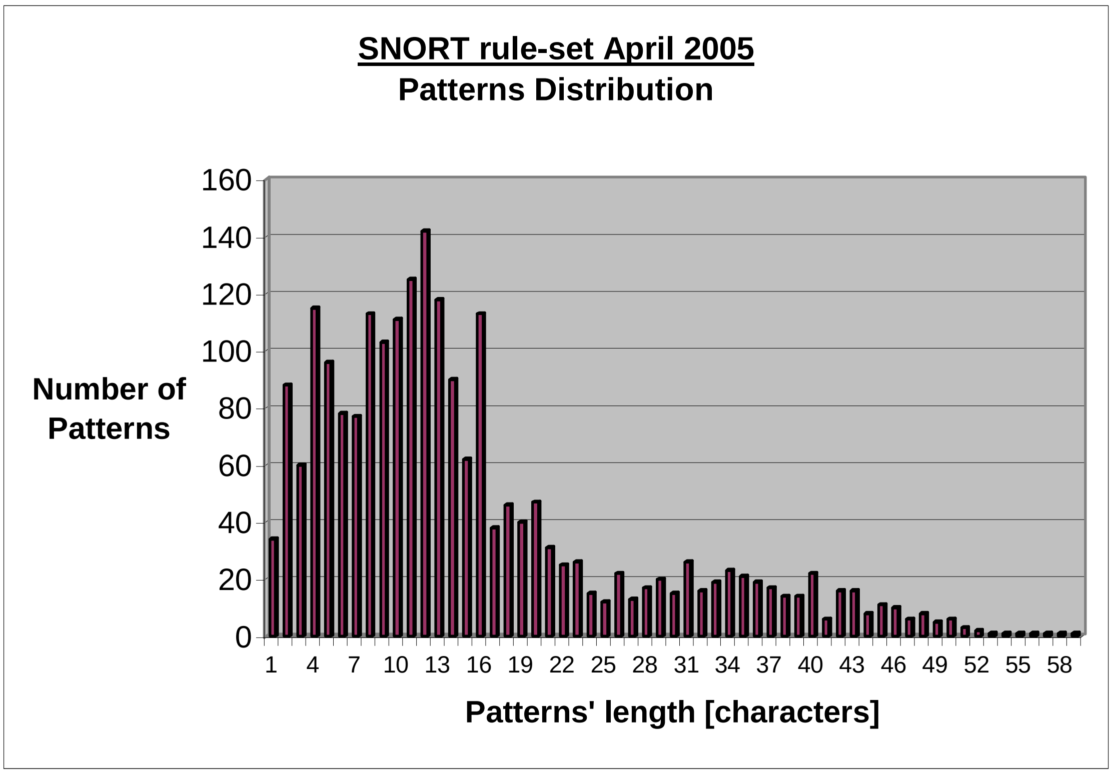

Figure 2.2: *Patterns Distribution of SNORT rule-set of April 2005. There are also some patterns longer than 58 characters which are not shown in this figure. The number of them is 5, one per each category of the following categories (based on character length): 61, 62, 80, 107, and 122. Finally, somebody can notice that most of patterns have length in the range from 1 to 16 characters. This notice is very important and it is used in chapter 3.*

Each Snort rule can contain header and content fields. The header part contained information about protocol, source and destination IP addresses and port. The content part contains substrings that may exist in packets' payload. In this work we used the SNORT rule-set [20] of April 2005 which contains 2187 rules or 33613 characters. Patterns' Distribution of these patterns is shown in Figure 2.2.

If we look at Figure 2.2 we notice that most of patterns are in the range 4 to 16 characters long. Notice that we selected patterns' length 17 as the "border" between the short and the long strings because the number of patterns after the border is very small. This notice is very important and we use it in chapter 3.

2.2 Software and Hardware – Based NIDS

Many software and hardware – based NIDS have been designed the last years. In software – based NIDS, software pattern matching algorithms are used based on SNORT rule-set. However, the most important problem of software – based NIDS is the slow throughput and slow performance. Through the years, many improvements have been done. One of these improvements was Boyer-Moore algorithm which improved the performance at 200% to 500% but generally, for a few hundred Mbps the software – based NIDS became a serious bottleneck in networks' speed as a result hardware accelerators are necessary to process packets in real time or near real time.

On the other hand, Hardware – based NIDS can be used to overcome the problems of low speed and low throughput. NIDS have been designed in FPGAs as well as in ASIC. In ASIC systems, patterns are stored in large memory blocks and determine whether there is pattern matching or not using integrated design machines. In addition, in these systems the update to support new rule-sets is a hard procedure since memories have to be reloaded with new data without having the capability of upgrading the search engine in case the kind of rules changed. This approach is very expensive and furthermore their performance is not impressive. However, they achieve much better throughput than Software – Based NIDS.

Hardware – based NIDS with the FPGA approach can be a significant alternative. FPGA are reconfigurable and moreover achieve very good performance and throughput comparing to Software – Based NIDS or Hardware – Based NIDS with the

ASIC approach. Since FPGAs are reconfigurable the entire machine can be changed as a result any update can be done. The only constraint is to keep the interface unchanged. However, this procedure can be very hard since the whole machine is changed but it is rather easy when a small change occur in an updating. There are many FPGA – based architectures, such as NFAs and DFAs based on regular expressions, regular CAM, Hashing, etc.

The most common FPGA approach is regular expression using *NFAs (Non Deterministic Finite Automata)* and *DFAs (Deterministic Finite Automata)*. Regular Expression is a string that describes a set of strings, according to certain syntax rules. Regular Expressions could be the following:

- ab (a followed by b)
- a* (one, zero, or more a)
- a^+ (one or more a)
- a/b (a or b)
- ε (empty)

DFA is a finite state machine in which for a state and for a specific input there is a deterministic next state. On the other hand NFA is, again a finite state machine (the node is a state and the edge is a character or the empty which is symbolized by ε), but there could be multiple next stages. For instance, if we are in state, say, 5 and the input is B the next state could be either 7 or 10. It is similar to a tree-like structure. Basically, the main difference between DFAs and NFAs is that in the first ones there is exactly one next state for a possible input while in the second there could be multiple next states for one given input. It is easy to design an NFA but the implementation is complex in contrast with DFA in which design is difficult but implementation is easy.

Generally, finite automata are usually restricted in their operating frequency by the amount of combinational logic for state transitions. Also, the use of parallelism (processing multiple bytes or characters per cycle) is in general difficult in finite-automata implementations that are built with the implicit assumption that the input is checked one byte at a time. One proposed solution to this problem is the usage of packet-level parallelism where multiple pattern matching subsystems operating in parallel can process more than one packet.

The first hardware implementation was introduced by Floyd and Ullman in 1982, implemented in PLA [15]. Then Sidhu and Prassanna [16] introduced regular expressions and Nondeterministic Finite Automata (NFAs) for finding matches to a given regular expression. Their primary goal was to minimize the time needed in order to construct an NFA. They achieved about 58 to 94 MHz frequency for a single regular expression using the Virtex Device Family.

Then, Franklin, Carver and Hutchings [17] expanded on Sidhu et al. work. Their primary goal was to cover the maximum number of expressions and for that reason they used regular expressions with more complex rules and meta-characters. They managed to include up to 16,000 characters requiring 2.5-3.4 logic cells per matching character. They achieved about 100 MHz for Virtex Device Families.

Finally, Clark and Schimmel [18] developed a pattern matching coprocessor that supports the entire SNORT rule-set using NFAs. Their primary objective was to have small area cost. In order to achieve it they used centralized decoders instead of character comparators for the NFA state transitions. They can match over 1,500 patterns (17,537 characters) and achieved 100 MHz frequency having total throughput 0.8 Gbps in a Virtex-1000 device having one character processing throughput. Later [9] they expanded their work and allowed more than one characters processing throughput. Their detailed results proved that NFAs and predecoding can produce low cost designs with higher performance, compared to DFAs and simple bruteforce approaches.

Another FPGA-based approach is *CAMs* and *Discrete Comparators*. One approach was introduced by Gokhale, et al. [6]. They used a CAM to implement SNORT rules. Their hardware can serve a total throughput of 2 Gbps in a VirtexE device. Cho, Navab and Mangione-Smith [19] presented an architecture of pattern matching using discrete comparators. They were the first to use 4 parallel comparators for every pattern in order to exploit parallelism and process 4 bytes of packet data every clock cycle. They used an Altera EP20K device and achieved a frequency of 90MHz, achieving 2.88 Gbps throughput.

A very good approach was also introduced by Sourdis et al [5]. They, firstly, reduced the area cost of character matching using (i) character pre-decoding before their comparison in the CAM line, (ii) efficient shift register implementation using the SRL16 Xilinx cell. Then they achieve high operating frequencies by (iii) using fine grain pipelining for faster circuits and (iv) decoupling the data distribution network from the processing components. Their results show that for matching more than 18,000 characters (the entire SNORT rule set) their implementation requires an area cost of less than 1.1 logic cells per matched character, achieving an operating frequency of about 375 MHz (3 Gbps) on a Virtex2 device. When using quad parallelism to increase the matching throughput, the area cost of a single matched character is reduced to less than one logic cell for a throughput of almost 10 Gbps.

Finally, another FPGA-based approach is *Hashing* in which the input stream enter a Hash function and the result is a pointer to a possible match. V-HashMem Architecture is based on this idea. Also, many other works have been based on this idea, such as HashMem by Papadopoulos et al.[2], Bloom Filters by Dharmapurikar et al [4, 8], PHmem by Sourdis [3] et al and others which are described in more detail in the paragraph Related Work of the chapter 5.

Chapter 3

The Variable-Length HashMem (V-HashMem) Structure

In this chapter, we describe in short the HashMem Architecture which was designed and implemented by George Papadopoulos and Dionisios Pnevmatikatos [2]. Then, we describe the idea on which V-HashMem Architecture is based and the special issues of V-HashMem approach. Finally, we describe the procedure we followed in order to design the basic V-HashMem structure.

3.1 The HashMem Architecture

Before starting the description of the V-HashMem architecture lets refresh our memories about the HashMem architecture [2]. The HashMem pattern matching Architecture is a SNORT accelerator architecture based on the idea that a simple hash function of the input can generate a set of sparse but distinct addresses for the search patterns. This address is used as a “pointer” to a possible matching string, and compares that to the input to determine the final match signal.

3.1.1 Basic Architecture

The HashMem Architecture consists of 14 subsystems, one for each character length from the range 3 to 16 characters. Each one subsystem looks like the system of figure 3.1.

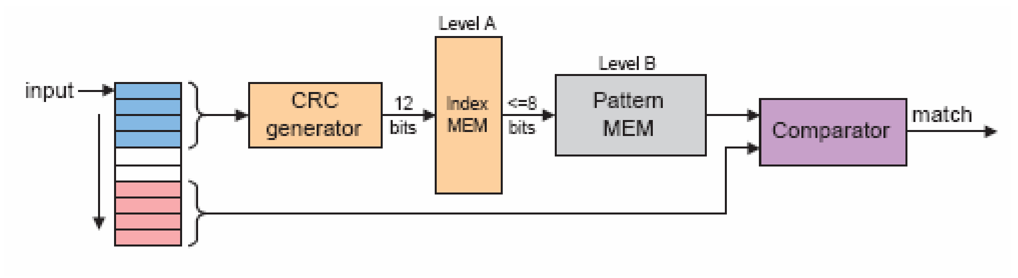

Figure 3.1: *One of the 14 subsystems the HashMem Architecture consists of.*

The number of stored patterns in each sub-system is smaller than 256 and the width of each of them is the number of its characters multiplied by 8 (each ASCII character is 8 bits). So patterns of each length can be stored in a Pattern Memory of 256 entries (the shortest memory block of Xilinx FPGAs has 512 entries). Furthermore it was found experimentally that a 12-bit polynomial is a simple polynomial for SNORT patterns. A 12-bit polynomial means that the pattern memory must consist of 4096 entries and consequently the utilization of the memory is less than 2.5%. That is the reason why an Indirection (Index) Memory is used. Indirection Memory is basically a pointer memory which uses the 12-bit address that CRC generator generates and feeds an 8-bit address into the Pattern Memory achieving very good utilization of the Pattern Memory.

Thus, the input stream is fed into the CRC generator which generates a 12-bit address. Indirection Memory "transforms" the 12-bit address into an 8-bit address and feeds the Pattern Memory. The output of Pattern Memory is compared to the properly delayed input stream for a match. However, since any given character of the input stream can be the last character of a pattern of arbitrary width, system of figure 3.1 is replicating once for each of the different pattern widths for the range 3 to 16 characters. The resulting architecture is shown in figure 3.2.

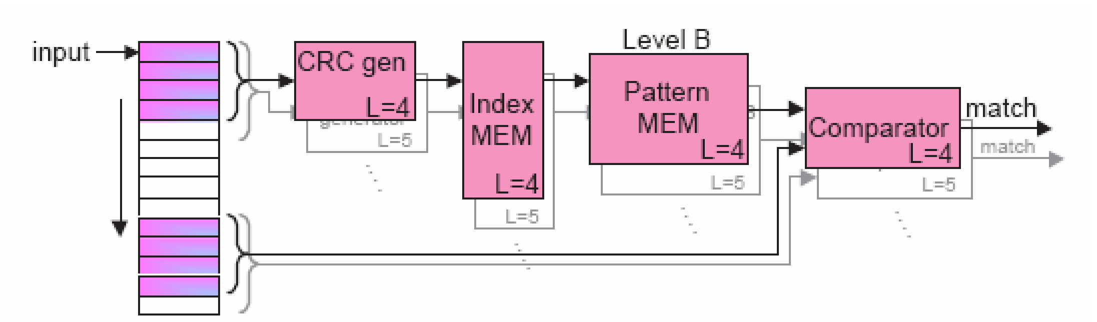

Figure 3.2: *The HashMem Architecture for pattern widths 3 to 16 characters*

3.1.2 Architecture for Very Short Patterns

SNORT rule-set also contains very few patterns of one or two characters. These patterns consist of very few bits making the CRC calculation overkill. The architecture which handles these patterns is shown in figure 3.3. For the patterns of a single character a LookUp Table (LUT) of 256 entries is used. The input byte is used directly as an address to access the LUT, which is initialized with ‘1’ in the addresses that represent the search patterns.

The two characters patterns are less than 64, thus allowing an encoding with 6 bits. Based on this observation, a recoding function in the unused bits of the single character lookup table is added, recoding the 8 bits input into a 7 bit code. The most significant bit represents a single character match and the rest 6 bits are the encoded value of the half part of dual-byte patterns. Then, each encoded part is stored in a register and after the appropriate delay of the first encoded part, the two input recoded characters, which amount to 12 bits, address a 4Kx1 lookup table to determine any two-character match. Each of these two lookup tables uses one memory block and a minimal amount of logic for the pipeline delay.

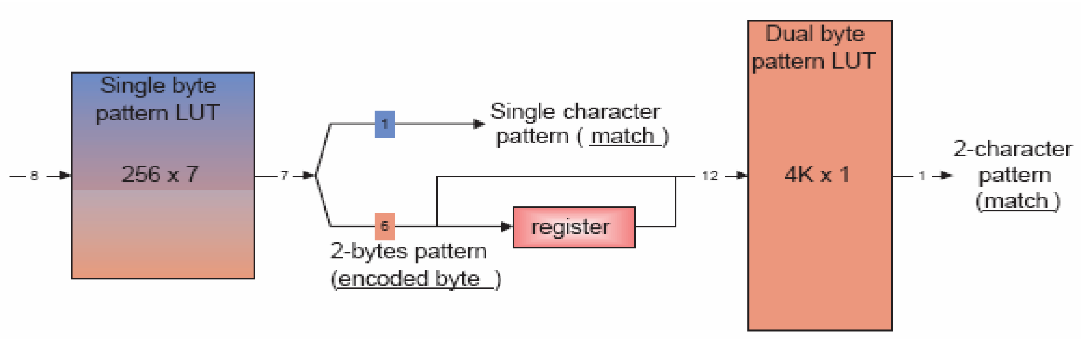
8
Single byte pattern LUT
256 x 7
7
1
Single character pattern (match)
6
register
2-bytes pattern (encoded byte)
12
Dual byte pattern LUT
4K x 1
1
2-character pattern (match)

Figure 3.3: *The HashMem Architecture for short patterns (one and two character)*

3.1.3 The HashMem Architecture handles very Long Patterns

SNORT rule-set contains, finally, few long patterns in a range from 17 to 122 characters. These patterns require small but very wide memories. For example, there is only one pattern of 122 characters and its width is 976 bits:

$$\boxed{\text{Pattern Width} = \text{Number of characters} * 8 \text{ bits/character}}$$

This pattern would require 28 “512x36” memory blocks. In addition, it would require high-cost CRC generators and comparators making their design inefficient for this case.

The idea was to split these wide patterns in smaller pieces, store them in the Pattern Memories of the basic HashMem Architecture updating the Indirection Memories, reuse logic and add extra glue-logic to combine the partial matches into the complete match. The final HashMem architecture is shown in Figure 3.4.

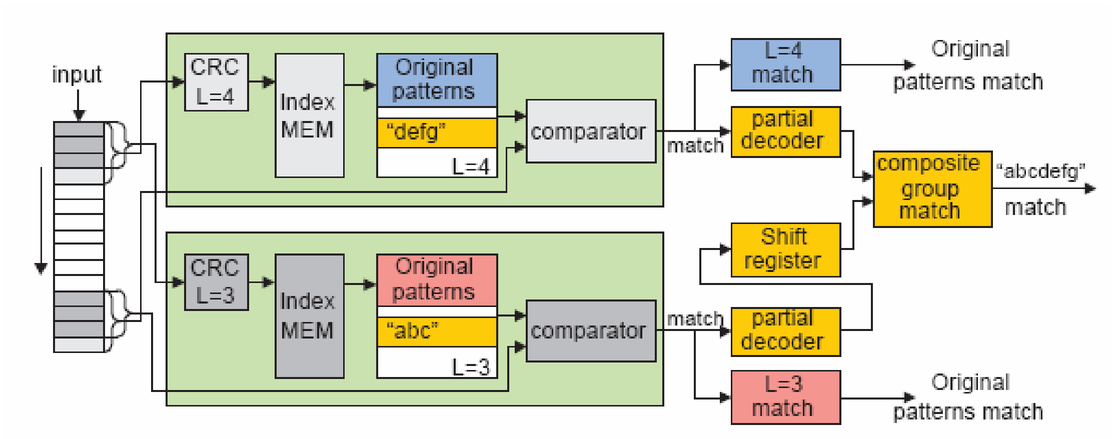

Figure 3.4: *Final HashMem Architecture. In this figure it is shown a short example of a wide pattern matching. The wide pattern has only 7 characters but it was chosen only for reasons of clarity and brevity. Wide patterns are in the range from 17 to 122.*

3.2 V-HashMem Architecture – The Idea

Variable-Length HashMem (V-HashMem) Architecture is an extension of HashMem Architecture and it is based on the idea that it is advantageous to store the search patterns of different lengths in a single memory structure. If this is possible, then we need fewer memory structures, i.e. less memory, as well as fewer CRC generators and comparators, i.e. less logic. The V-HashMem targets exactly these factors. However, this extension raises the following issues that have to be addressed.

- **Hash function generation**: HashMem uses all *L* characters of the input to hash the address of the possible match. If we allow for several lengths, how do we know how many characters to use for the hash calculation?

- **Dealing with Prefix Conflicts**: In HashMem, all patterns of *L* characters are stored in one Memory. Is it possible to store all patterns of different lengths in the same memory or there are prefix conflicts and how do we deal with?

- **Comparators**: Similar to the previous issue, how do we know how many characters to match between the input and the stored pattern?

3.2.1 Hash Function Generation

The first issue we have to address is how many characters we must use to feed the CRC generator. The hashed value is just a pointer to a possible match since the comparator determines if there is a match or not. Thus, we can use the minimum length of the stored patterns in the structure to feed the CRC generator as long as the CRC generator produces a distinct value for every single input. For example, if we have a structure that holds patterns of 5, 6 and 7 characters long, we can feed the CRC generator with 5 characters as long as the result of this function is unique for every single pattern which is stored in this structure.

If this is successful, not only do we not care about how many characters have each stored pattern but also we use less CRC generators and especially the narrower.

3.2.2 Dealing with Prefix Conflicts

Although storing variable-length patterns in a memory structure is clearly beneficial, it is not always possible. In 3.2.1, we said that we use the minimum length of the stored patterns in a structure to feed the CRC generator as long as the hash value is unique for every single pattern. The minimum length of the stored patterns in a structure is basically named as prefix.

A conflict rises when stored patterns in one structure have the same prefix because different patterns are mapped in the same position in the memory. Assume that we have a structure that holds patterns of 3, 4 and 5 characters long. We feed the CRC generator with a prefix of 3 characters of all stored patterns to find the address in the memory. Patterns like “abc” and “abcd” as well as patterns like “abcd” and abcf” or “abcdr” and “abceh” cannot be in the same structure because we use the prefix “abc” to map the pattern to a memory address but in every case both strings are mapped into the same position. If we observe the first case again we can say that it is not a real problem because any input matching ”abcd” also matches ”abc” so there will be a reported match but we have to find a way to solve the other cases.

3.2.3 Comparators

Another problem is how many characters to match between the input and the stored pattern. We do not know in advance which input characters may match which pattern length, but we do know in advance the length of each of the search patterns we are looking for. So, it is a good idea to encode this information in the memory and use it during the comparison. The best way encoding pattern’s length is to add Don’t Care bits between the characters of the stored pattern. Of course, we do not add Don’t Care bits in the characters of the prefix as it is matched always and does not need marking. For example, if we have a structure where patterns of 3 to 5 characters long are stored we add Don’t Care bit before the 4th and the 5th character as they are optional.

By adding these bits, stored pattern is a little wider than before as a result we have a small increase in the memory as well as in the cost of the comparator.

3.3 V-HashMem Structure

The HashMem Architecture consists of 14 similar subsystems, supporting patterns of a length range 3 to 16 and the other patterns by reusing logic and memory. As one begins the design of V-HashMem many questions are raised. For instance: Are we going to construct subsystems supporting all the patterns without reusing logic and memory? If not, after which length are we going to reuse logic and memory and how do we group the patterns in each variable length subsystem? Is there any other conflict except the one that is mentioned in paragraph 3.2.2?

3.3.1 Basic V-HashMem Structure

As it is mentioned before, Variable-Length HashMem is based on the idea that multiple patterns of different lengths are stored in a single memory structure. It is easy to decide that the patterns which are stored in the same memory structure must have close lengths. Then, we must find a way to decide which patterns we are going to group in the same memory structure.

SNORT rule-set contains many patterns in a range from 1 to 122 characters. We saw the Patterns Distribution in Figure 2.2. Since our goal is not only to use less memory structures than HashMem but also with a good utilization, we will try to group the patterns of different lengths in the same memory so that to achieve the best utilization.

Firstly, we group the patterns as it is shown in Table 3.1 in order to achieve the goal which is mentioned before.

Structures	Number of Patterns
3-5	271
6-8	268
9-11	339
12-14	350
15-17	213

Table 3.1: *Number of Patterns in each structure. We put patterns of length 18-20 or 18-21 together but the number of them was 133 and 164 respectively which is not a good memory utilization.*

We select patterns' length 17 as the "border" between the short and the long strings because the number of patterns after the border is very small and we have to put patterns of no close lengths together in order to achieve a satisfactory utilization. Furthermore, the width of the memory will be very wide and many memory blocks will be required for its implementation. For example, if we put the patterns, which have lengths 18, 19 and 20, together they are only 133 (Figure 3.5) and this is not a good utilization of the memory. Also, the memory's width will be 20 * 8 = 160 and this requires 5 memory blocks.

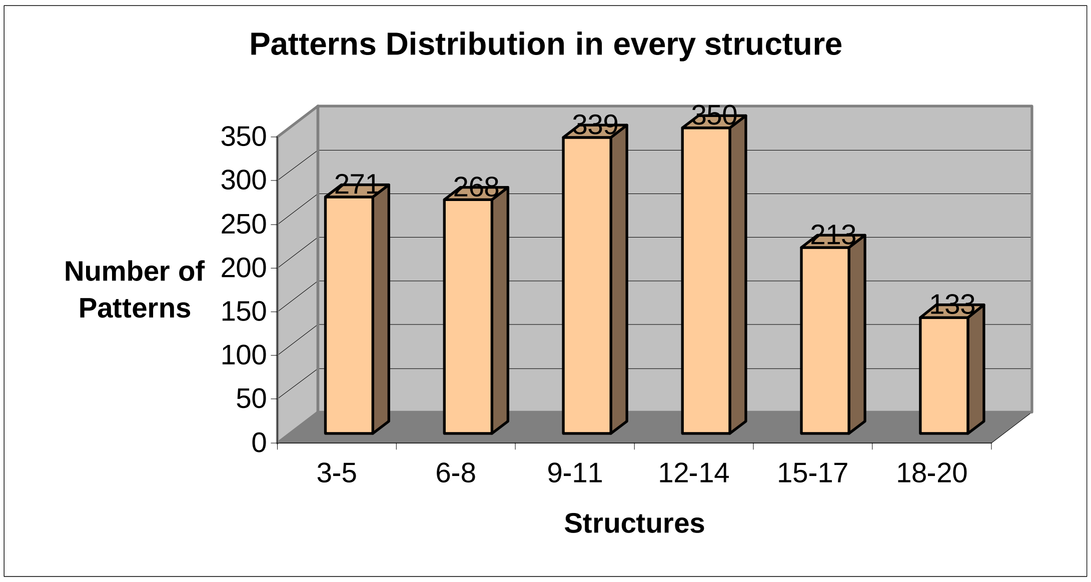

Figure 3.5: *Patterns Distribution in every structure. This proposal was rejected because using as a border the number of 20 characters and not the number of 17 characters we have an additional memory structure, the 18-20 memory structure, in which the number of patterns is very small and we do not achieve satisfactory memory utilization. Furthermore the memory's width is 160 bits; as a result 5 memory blocks are required for this memory structure.*

Then, we have to check these groups for the conflict that is mentioned in 3.2.2. For that reason, a software programme designed in C was used. The number of conflicts was great and we have to find another grouping. The new grouping is shown in Figure 3.6 and there are no conflicts. Also the memory utilization is satisfactory.

3.3.2 V-HashMem Structure for Long Patterns

We managed to group the patterns which have lengths between 3 to 17 characters. Now we have to find a way to break the long patterns and put them in the structures which are shown in Figure 3.6.

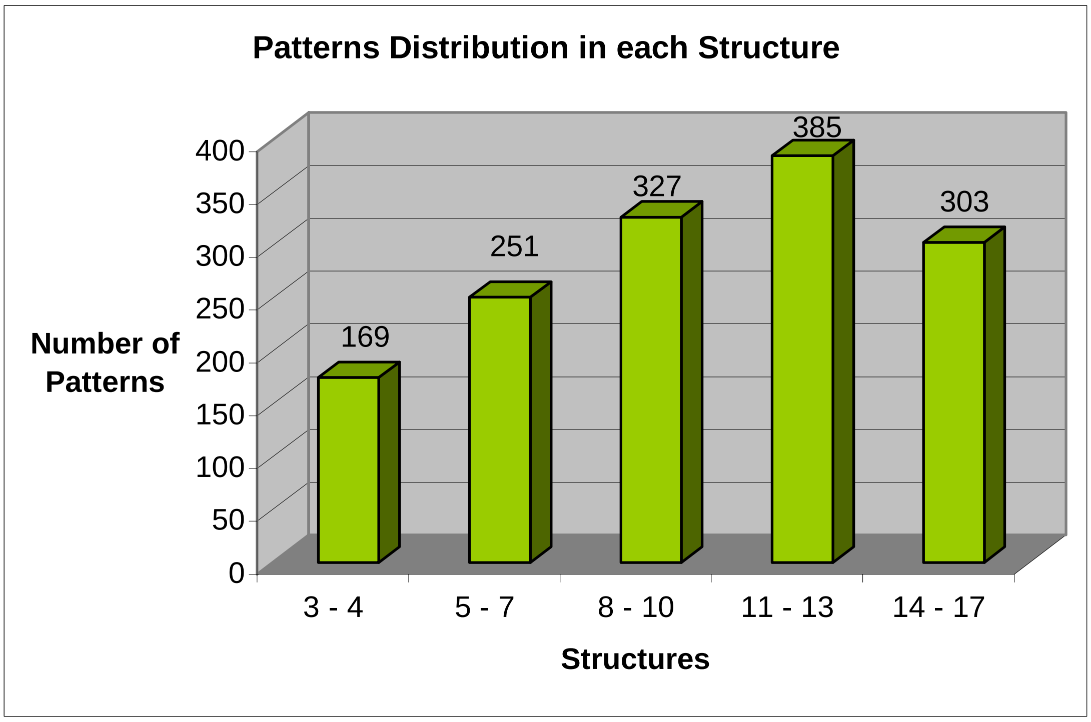

Figure 3.6: *New grouping of patterns to avoid conflicts. No conflicts were found.*

First of all, we have to decide in how many parts we will break each pattern. We don't care if the partial patterns, the fragments, have the same length but we do care that each pattern will be broken into the minimum possible parts in order to reduce the possible time to glue them and report the final match (small overhead). A simple algorithm was designed and used to break the long patterns into shortest ones fulfilling the above requirements. The steps of the algorithm are:

1. Initialization of Number of Parts to 2.
2. Number of characters is divided with Number of Parts. The result is the number of characters of the current part.
3. The number of characters of the current part is subtracted from the number of characters.
4. Reduce the Number of Parts by a factor of 1.
5. Follow steps 2 – 4 for all the parts until Number of Parts become zero.
6. Check if there is any part that consists of more than 17 characters (the longest fragment must have at most 17 characters) and report.
7. If the 6th step is true add one to the Number of Parts, initialize again Number of Characters with number of characters of the long pattern and follow again the steps 2 – 7. Otherwise, report the results.

Then we use the results of the above algorithm and divide every pattern of the current pattern group using a simple script written in Python. An example is always useful to better comprehend the concept. Assume that we have a pattern that consists of 48 characters. Using the above algorithm we have:

The number of parts is initialized to 2. The number of characters is 48. We follow step 2 and the result is 24. This is the number of characters of the first part. Following step 3 we subtract this number from 48 and the result is 24. We also subtract 1 from the number of parts, following step 4. We follow again steps 2 – 4 and we have 2 parts of 24 characters. We check the number of characters of the first part and we observe that the number of characters is greater than 17. We add one to the number of parts (3 is now), we initialize number of characters with 48 again and we follow again steps 2 to 7.
Following step 2, the number of characters of the first part is 16. We make the subtraction 48 – 16 and 32 is the number of characters now. We also reduce the number of parts by 1 and the new number of parts is 2. Following again step 2, we divide the number of characters (32) with the number of parts (2). The number of characters of the second part is 16, and the number of parts is 1. Following again steps 2 – 4, the third part has 16 characters and there are no other parts left. Following step 6, none of the parts is longer than 17 characters as a result (step 7) the algorithm reports the results which are: 1st part: 16 characters, 2nd part: 16 characters, 3rd part: 16 characters.

Following the procedure we described before, we broke all patterns longer than 17 characters. Then, we select the fragments and we put them in the structures which are shown in Figure 3.6. The number of patterns before and after the partitioning is shown in Figure 3.7.

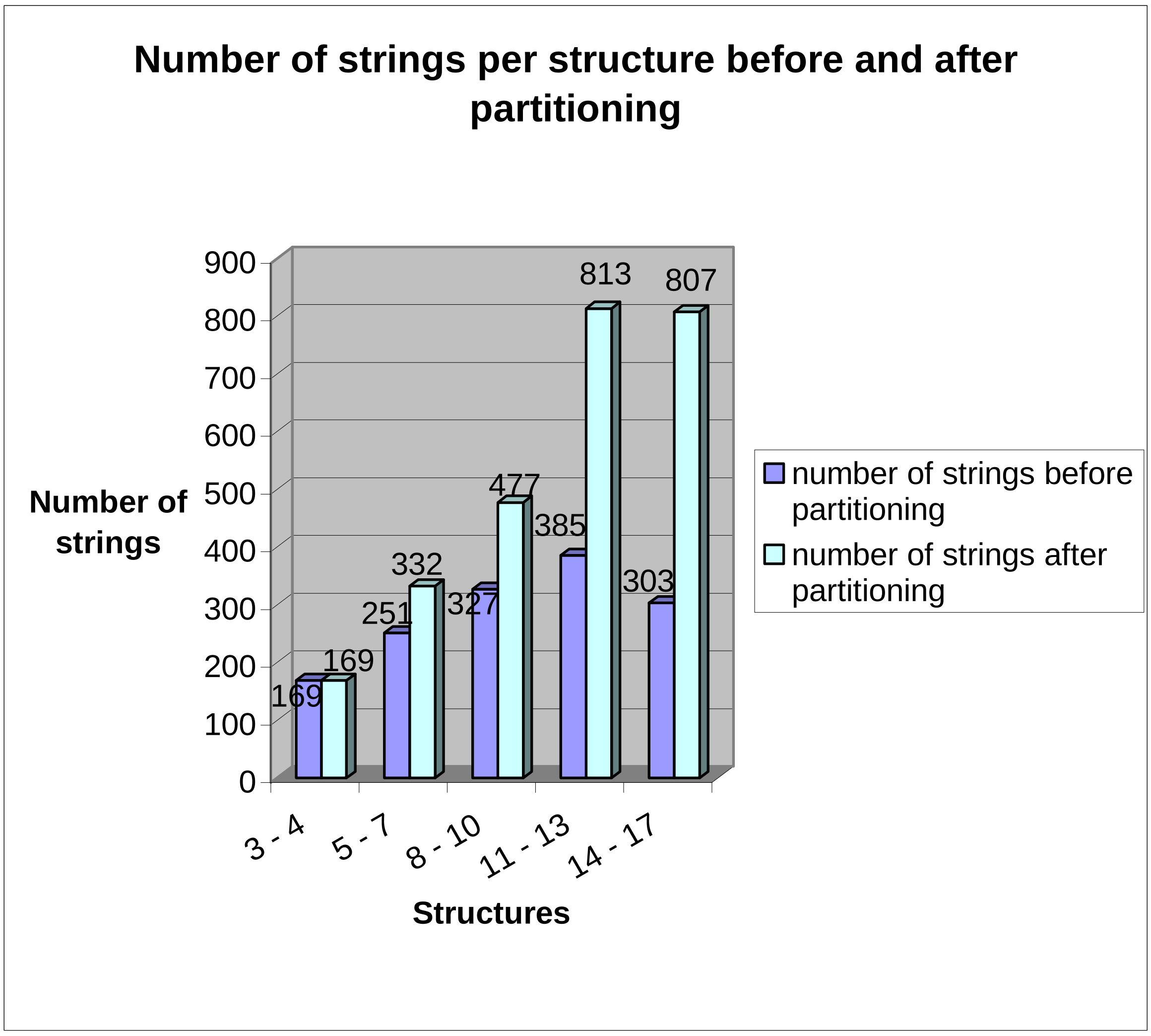

Figure 3.7: *Number of patterns per structure before and after partitioning. In structures 3-4, 5-7, 8-10 the new number of patterns is still smaller than 512 while in structures 11-13 and 14-17 we have about 130% increase in the number of patterns.*

After putting all fragments in the structures we had to check them again for conflicts following the procedure we followed in paragraph 3.3.1. It resulted to a great amount of conflicts in each structure. We have to find a new and better structure which removes the conflicts regardless of the SNORT rule-set we use.

The idea is to have overlapping structures or to have more than one same length structures in the V-HashMem. For example, assume that we have two overlapping structures: 5-10 and 8-13. The conflicts for patterns which have length 8, 9 or 10 characters can be removed by moving one of the patterns who create the conflict from structure e.g. 5-10 to 8-13. By this technique we solve many conflicts in which a couple of patterns have the same prefix. There are also few cases in which the patterns that create a conflict are more than two, usually three. In these cases, we just remove the conflict by letting one pattern in the current structure, moving the second to another structure, deleting the third one and re-breaking the long string, in which the third one was a part, again randomly. The distribution of patterns in each structure in the final V-HashMem is shown in Figure 3.8.

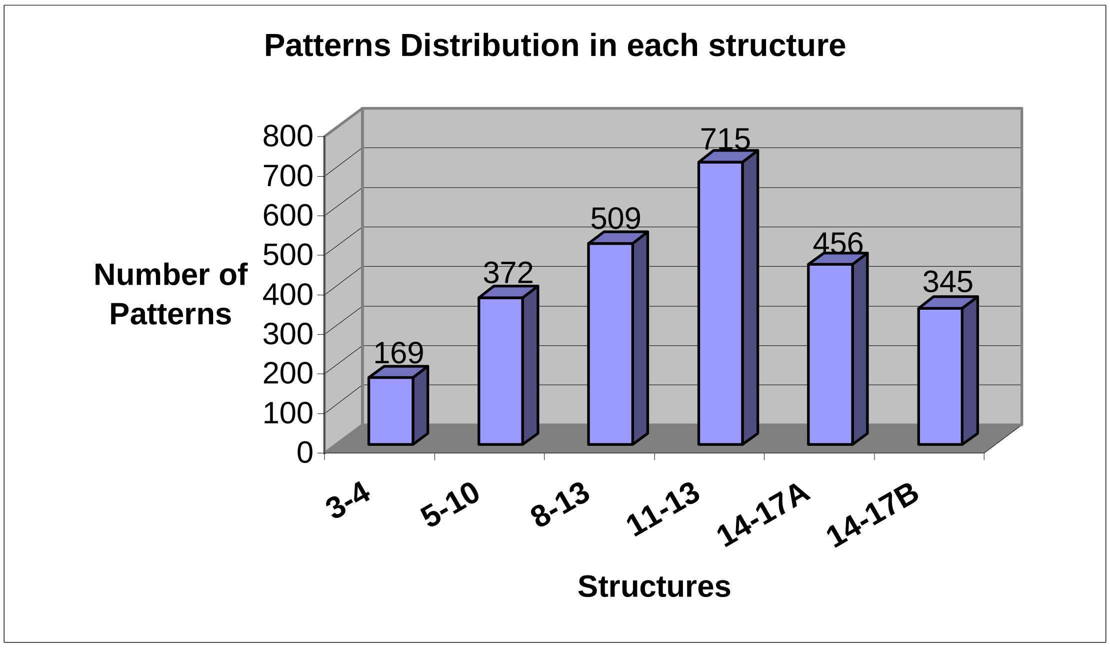

Figure 3.8: *Patterns' Distribution in each structure in the final V-HashMem. The Patterns' Distribution which is shown in this figure is the final and it is after solving not only the conflicts we mentioned before but also all some other conflicts we found and we are talking about them in the next chapter.*

3.4 Summary

In this chapter, we described in short the HashMem Architecture. Then we presented the idea on which V-HashMem was based and we tackled the issues concerning the CRC generators, the comparators and prefix conflicts. Finally, we presented to the reader the steps we followed to design the V-HashMem structure.

Chapter 4

Variable – Length HashMem Architecture (V-HashMem)

In this chapter we describe the design of the basic V-HashMem Datapath. Then, we describe Glue Logic and after these issues we describe the design and implementation of the final V-HashMem Architecture. At the end of the chapter we introduce 2 improvements in order to make our design more efficient.

4.1 Basic V-HashMem Datapath

In this paragraph we describe how to design the Basic V-HashMem Datapath based on what we have already mentioned in chapter 3. At first, we describe the design of V-HashMem for the very short patterns, then the design of each component (CRCs, memories, etc) and finally we present the basic V-HashMem Architecture.

4.1.1 How V-HashMem handles Very Short Patterns

SNORT rule-set contains very few patterns of one or two characters as we mentioned in paragraph 3.1.2. V-HashMem handles 1-character long patterns in exactly the same way as HashMem. However, there is a slight difference about how V-HashMem handles 2-character long patterns comparing to the HashMem.

Two-character patterns in the SNORT rule-set of April 2005 are more than 64 but less than 128. We use the same idea as HashMem but we recode the 8 input bits into an 8-bit code from which the most significant bit is a single-character match and the other 7 is the encoded value of the half part of dual-byte patterns. Again after the appropriate delay the two input recoded characters amount now to 14 bits, address to a 16Kx1 lookup table. This approach does not increase the memory block requirements comparing to HashMem since the shortest (smallest width) primitive memory block is

16Kx1 (For Dual byte Pattern LUT) and the widest 512x36 (For Single Byte Pattern LUT). Figure 4.1 shows how V-HashMem handles one and two characters patterns.

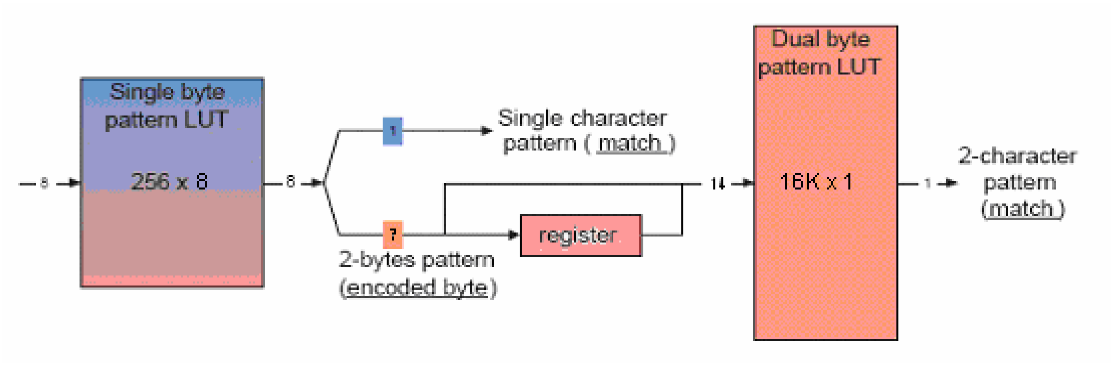
Single byte pattern LUT
256 x 8
8
8
1
Single character pattern (match)
?
2-bytes pattern (encoded byte)
register
14
Dual byte pattern LUT
16K x 1
1
2-character pattern (match)

Figure 4.1: *How V-HashMem handles very short patterns.*

4.1.2 CRC generation

A CRC generator transforms the input data into other data using a specific polynomial. In paragraph 3.2.1, we said that we use the minimum length of the stored patterns in the structure to feed the CRC generator. In table 4.1, it is shown how many characters we use to feed the CRC generator in each structure.

Structure	CRC's input width [bytes]
3-4	3
5-10	5
8-13	8
11-13	11
14-17	14

Table 4.1: *How many character we use to feed the CRC generator in each structure. A character is one byte.*

Since wide CRC generators have a high cost, we implement full CRC generators for inputs' lengths 3, 4, 5 and 6 characters (bytes). For inputs longer than 7 characters we break the input stream into shorter ones and feed the existing CRC generators. Then, we glue their result and feed another CRC generator. For example, if we have an input of 8 characters we break it into 2 pieces (e.g. 4 characters each one) and we feed each piece into a 4-byte CRC generator. All the 12-bit results of every CRC are concatenated and the result is fed into a 3-byte CRC generator. This is shown better in Figure 4.2.

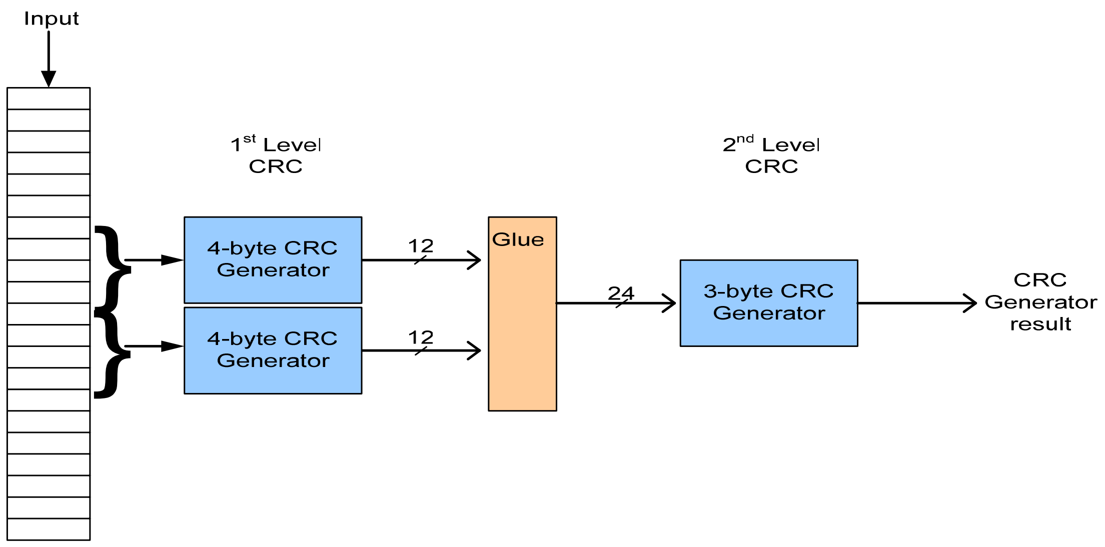

Figure 4.2: *CRC generation for an 8-character input.*

The width of the output of the CRC generator affects the length of the memory of each structure and consequently its cost, since the output of the CRC generator is used as an address to access the memory. For that reason, we want CRC generator to have the minimum output width. Minimum output width means that we use polynomial of small degree. On the other hand, the maximum the degree the better mapping of patterns in the memory. Taking into account these parameters we resulted that the best polynomial's degree is 12 for the all the patterns, except the patterns which are in the structure 3 – 4 where we use degree 11 and for the patterns which are in the structure 11 – 13 where degree 13 is used. In the table 4.2 it is shown clearly how many bytes have the CRC generators in each structure.

Structure	CRC's input width [bytes]	1st level CRC's input width [bytes]	2nd level CRC's input width [bytes]
3-4	3	3	no
5-10	5	5	no
8-13	8	5, 3	3
11-13	11	6, 5	3
14-17	14	3, 3, 4, 4	6

Table 4.2: *In this table, it is shown the Input width for the CRC generator of each structure in each level. In structure 11 – 13 we use polynomial with degree 13 only for the 2nd level CRC generator. For the CRC of the 1st level we use 12th degree polynomial.*

Using a programme which was constructed by George Papadopoulos [2] we produce efficient CRC polynomials for the CRC generators we mentioned in Table

4.2. This programme specifies the best CRC polynomial for the specific patterns. If there is no polynomial which produces distinct addresses for every single input pattern this programme will specify the CRC polynomial which creates the least conflicts. We removed those conflicts by just moving one of the patterns which participate in the conflict from one structure to another. The number of patterns which created every conflict does not overcome 2.

Using these polynomials we generate the CRC generators (XOR-based) with Easics tools [21]. Then, we pipelined each CRC generator to achieve better 1) throughput and 2) cycle time.

4.1.3 Basic V-HashMem Datapath – Design and Implementation

V-HashMem Architecture, just as HashMem, has Index Memories in order to achieve better utilization in the pattern memory (this is described better in paragraph 3.1.1). Usage of Index Memories has another very important advantage: when we are storing the patterns in the Pattern Memory, we can put each pattern wherever we want. Then we initialize the Index Memory based on the calculated CRC values and the locations of the patterns in the pattern memory. This is shown in Figure 4.3. We described in the paragraph 4.2.2 how we use this property.

Based on what we have said until now, we design and implement V-HashMem datapath. The datapath which is shown in Figure 4.4 does not include all the structures but only the 3-4 and 5-10 structures because all structures have been designed in the same way, except structure 14 – 17. Structure 14 – 17 consists of two structures: 14 – 17A and 14 – 17B but we use one CRC generator for both of them because firstly it is the same for both structures and secondly only one of them will report a match for a specific input.

The input string is inserted into a long shift register, which has one byte (character) width, once per cycle. The smallest length of the input (based on the structure) is fed into the CRC generator. Then, the output of the CRC generator is used to access the Index Memory and take the Pattern Memory's Address which we use to find the pattern. Then we send the pattern and the appropriately delayed packet's string into the variable comparator which determines whether there is a match or not. This implementation has a single-byte processing throughput per cycle.

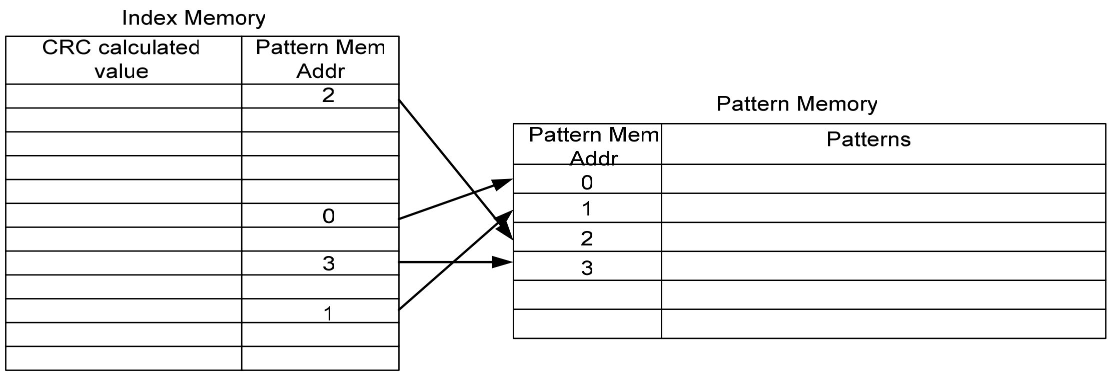

Figure 4.3: *How Index Memory is used. CRC address is used to access the Index Memory. Then data of Index Memory (Pattern Address) is used to access the Pattern Memory.*

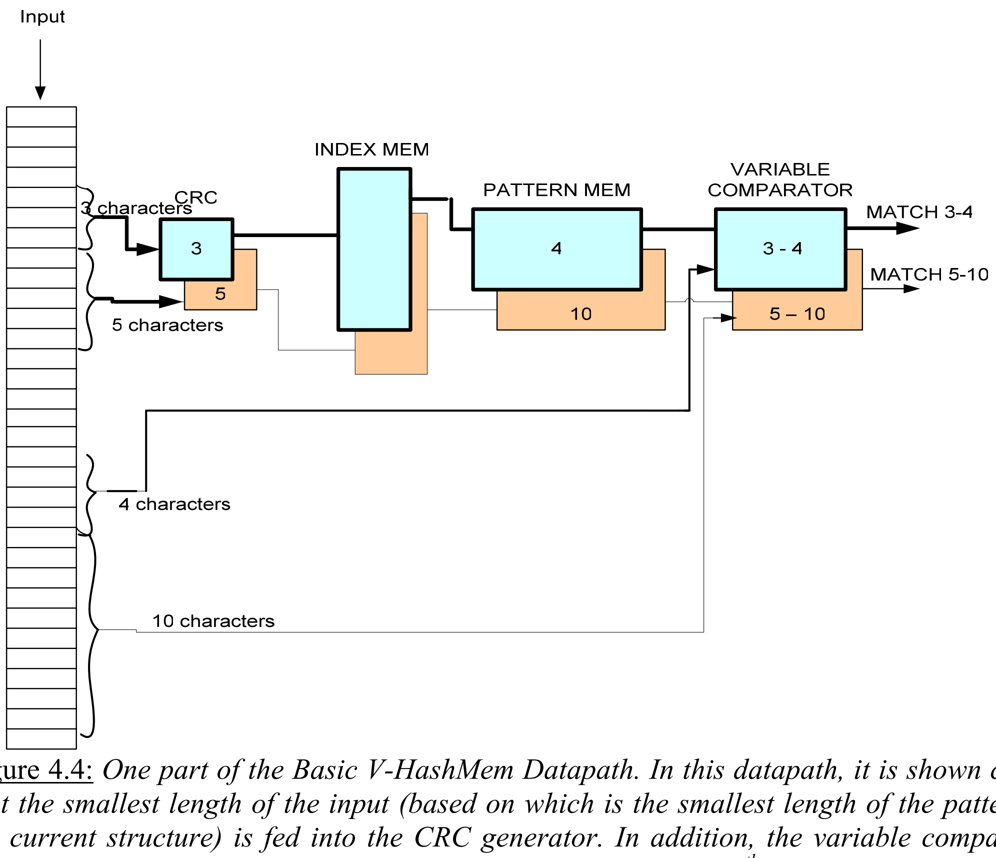

Figure 4.4: *One part of the Basic V-HashMem Datapath. In this datapath, it is shown clearly that the smallest length of the input (based on which is the smallest length of the patterns of the current structure) is fed into the CRC generator. In addition, the variable comparators take into account the Don't Care bits for the variable portion (4th character of the first one and 6-10 characters for the second) for the comparison.*

The area and memory cost (Index and Pattern Memory Cost) for the implementation of the Basic V-HashMem Datapath is shown in Table 4.3, Table 4.4 and Table 4.5 respectively. In these tables the cost is shown for each and every structure of the basic V-HashMem Datapath. In this implementation we use the SNORT rule-set of April 2005 which contains 2187 rules or 33613 characters.

Structure	Area Cost [Logic Cells (LC)]
3-4	88
5-10	126
8-13	164
11-13	120
14-17	308
Total	**806**

Table 4.3: *Area cost per structure and Total Area Cost of the basic V-HashMem.*

Pattern Memory Cost			
Structure	**Number of Entries**	**Width[bits]**	**Number of memory blocks**
3-4	512	33	1
5-10	512	85	3
8-13	512	109	4
11-13	1024	106	6
14-17A	512	139	4
14-17B	512	139	4
Total			**22**

Table 4.4: *In this table it is shown how many memory blocks are used for the implementation of the Pattern Memory, for each structure and for the total implementation of basic V-HashMem.*

Index Memory Cost			
Structure	**Number of Entries**	**Width[bits]**	**Number of memory blocks**
3-4	2048	9	1
5-10	4096	9	2
8-13	4096	9	2
11-13	8192	10	5
14-17A	4096	9	2
14-17B	4096	9	2
Total			**14**

Table 4.5: *In this table it is shown how many memory blocks are used for the implementation of the Index Memory, for each structure and for the total implementation of basic V-HashMem.*

The pattern memories of all the structures except structure 11 – 13 are implemented using the shortest and simultaneously widest memory block of Xilinx 512x36. The number of memory blocks in these cases depends clearly on the width of the pattern memory of each structure. On the other hand, in structure 11 – 13 we use the 1024x18 memory block because we need 1024 entries for the implementation of the pattern memory and we use 6 blocks because pattern memory's width is 106.

For the index memories we use longer memory blocks but with a small width. For example, the index memories of the structures 5 – 10 or 8 – 13 are implemented using the 2048x9 memory block 2 times for each structure putting them vertically. We could also use the 4096x4 memory block but you can understand that this would require 3 memory blocks (we would place them horizontally) of it, since these index memories have 9 bits width. We can easily understand that Xilinx offer many choices about how to use the memory blocks of the FPGA.

Finally, in the Table 4.6 we present the total Area and Memory Cost for the basic V-HashMem, and calculate the logic cost per character as well.

Basic V-HashMem Architecture	
Area Cost [LC]	806
Memory Cost [Memory blocks]	36
Logic Cells/ Character	0.02

Table 4.6: *Total Area and Memory Cost for the Basic V-HashMem Architecture.*

4.2 Glue Logic

In the previous paragraphs we have described the basic V-HashMem Architecture. We have described how the patterns are stored in the memories of V-HashMem and how a pattern matching, if it exists, is reported. Until now, V-HashMem Architecture reports a match, if it exists, but it does not mention what kind of match it is. We have to extend the current V-HashMem Architecture so that after the fragments are matched in the regular memory structures, custom glue logic will combine the partial match signals to determine whether the entire wide pattern was actually found in the input. Determining the actual match involves delaying the partial match signals appropriately to indicate the actual position of the partial pattern in the overall input. This technique has been also used in HashMem Architecture.

In the following paragraphs, we describe the idea on which Glue Logic is based. Then we describe some special cases on which Glue Logic is not working properly and what we have done in order to face them. Finally, we implement Glue Logic.

4.2.1 The Idea

Given the number of wide patterns (624 patterns wider than 18 characters), using a *per fragment* solution as in HashMem is inefficient. Instead we used an addressing convention that places fragments in consecutive locations, so that when the first fragment of a wide string is located say at location *x*, the expected location of the subsequent fragment is already known to be at location *x*+1. Then we compare the delayed expected address to the address of the second fragment and determine the match. Since this rule holds for all fragments of all patterns, the cost of the approach is to add one to the address and broadcast it (appropriately delayed) to all memory structures for future inspection.

In addition, it is useful to know what kind of pattern we have (fragment or not and what kind of fragment: first, medium or last), since we report only the non fragment and the last fragment. It is also useful to know in which memory structure the previous fragment is stored provided that the fragment exists. We encode this piece of information into 2 and 3 bits respectively and store it into every memory structure. This encoding is shown in True Tables 4.7 and 4.8. This piece of information is useful when the address is broadcasted (see paragraph 4.2.3).

Kind of Pattern	
Non Fragment	00
First Fragment	01
Medium Fragment	10
Last Fragment	11

Table 4.7: *Kind of Pattern.*

Previous Structure	
5-10	000
8-13	001
11-13	010
14-17	011
8-13 or 14-17	100
11-13 or 14-17	101

Table 4.8: *Previous Struct.*

4.2.2 Special Cases

With the previous idea we implement a great amount of fragments. However, there are patterns that when broken down to fragments create “strange” structures. These structures are:

- Tree – like structures
- Reverse Tree – like structures

- Special List structures

First of all, we will discuss about Tree – like structures like the ones that are shown in Figure 4.5a and Figure 4.5b. Consider the case of two patterns “abcdef” and “abcxyz”. A match of “abc” followed by a match of either “def” or “xyz” is a full match. This fan-out structure deviates from the +1 rule. This situation is relatively rare (178 patterns that form 41 trees in our rule set), and we solved it by adding a small exception memory in parallel to each pattern memory. In our example we would place the “abc” and “def” fragments in locations *x* and *x* + 1. Then we would place the “xyz” fragment at another location *y* and place the entry *x* + 1 in location *y* in the exception memory, indicating that a match at location *y* after matching location *x* + 1 is a full match.

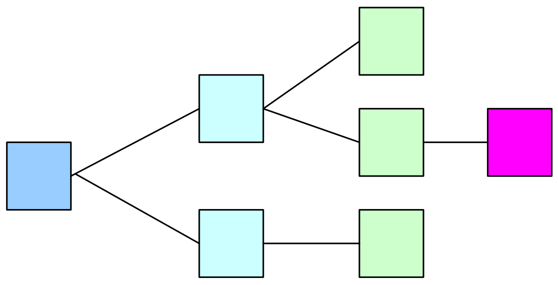
Figure 4.5a: *Multi – level Tree – like structure.*

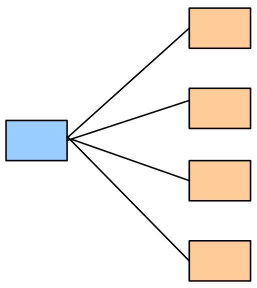
Figure 4.5b: *A 2-level Tree – like structure. Most of tree cases are like this one.*

The other case is the Reverse Tree – like structure and it is shown in Figure 4.6. In this case we put the common fragment in a position say x and all the other different fragments in a position x-1, using +1 rule. However, in some cases two or more different fragments may be in the same memory structure as a result they cannot be stored in the same position x-1. In these cases, we just move the fragments into another memory structure, removing this conflict.

There are also some cases which are a combination of tree and reverse tree like structures like the one that is shown in Figure 4.7. Since these cases are a combination of the cases we mentioned before we face them by dividing them into known cases.

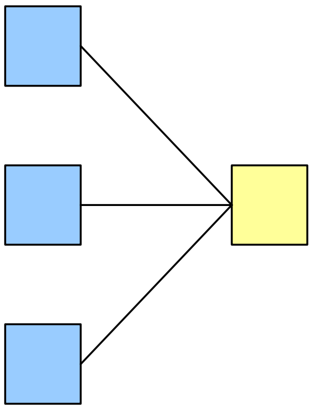

Figure 4.6: *Reverse Tree – like structure.*

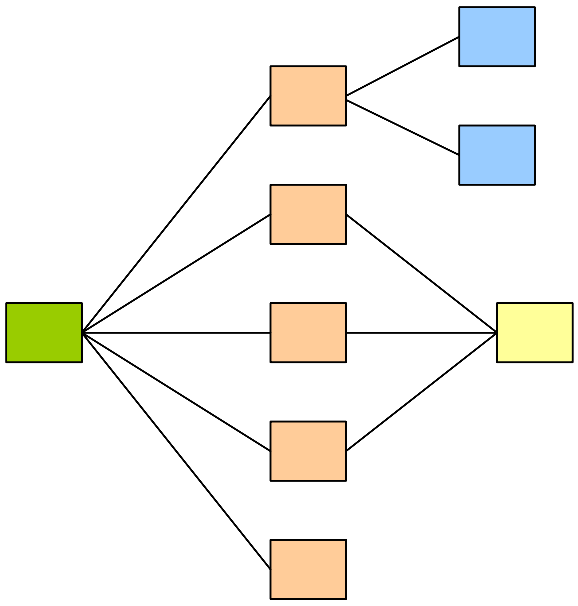

Figure 4.7: *Combination of Tree and Reverse Tree – like structures. We separate these cases into known cases and manage them using the techniques we showed.*

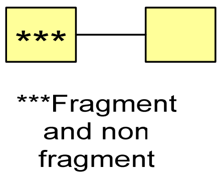

Figure 4.8: *Special List Structure.*

Finally, we have some cases which create special list structures. These structures are like the one is shown in Figure 4.8. These structures are simple lists in which the first or an intermediate node is both fragment and non-fragment and we have to report the match if it exists. Since these cases are very few (6 cases) we put these patterns into known positions and if we have a match we report it independently of being a first or an intermediate part of a long pattern.

4.2.3 Glue Logic – Design and Implementation

In the previous paragraphs, we described how we handle the fragments. We proposed our ideas and now we are ready to design the Glue Logic. The block diagram of Glue Logic is shown in Figure 4.9.

We use the output data from the Index Memory to access the Pattern Memory as we described in paragraph 4.1.3. We also use some bits of this data to determine (tree decoder) whether there is a tree-like case (Mux output is the Exception Memory's Output) or not (Mux output is the Data from Index Memory). The patterns of one structure which create Tree-like cases are placed into a certain zone (consecutive places) of the pattern memory. The id of this zone is compared with the input of the Tree Decoder. The number of bits of this id is not standard as it depends on the Structure (5-10, 8-13, etc) because the "size" of each zone depends on the number of the patterns which create Tree-like cases in each structure. The remaining bits of the output data of Index Memory are used in order to access the Exception Memory.

The Mask Decoder determines which broadcasted address to use based on the saved information in Memory (Bits of Structure of Previous Fragment). The Control Signal of the Mask Decoder depends on the kind of the pattern. If we have a non fragment the output of the Mask Decoder must be zeros since there is nothing to be glued. In any other case the control signal has a value 1.

We also use a 2-input OR gate after the first comparator. If we did not put this gate, we would never report a match when we have a non partial pattern because the output of the 4-input OR gate would be zero. For that reason, we put this 2-input OR gate so that when we have a non partial pattern the output of the 4-input OR must be one independently of the comparators. On the other hand, in any other case it will be zero and the report will depend on the comparators. In addition, we report a match when there is a match and it is either a non partial pattern or the last fragment and that is the reason why we use this XNOR gate. True Table 4.7 shows exactly why we use an XNOR gate. Moreover, we broadcast the address only when we have a first or an intermediate fragment and there is also a match. In any other case we broadcast zeros and that is the reason why these registers are cleared not only by reset but also by other signals, like match and the kind of pattern. The last comparator is used to face the Special List – like cases. Only in these cases and when there is a match we report it independently of the kind of pattern and the output of the other logic.

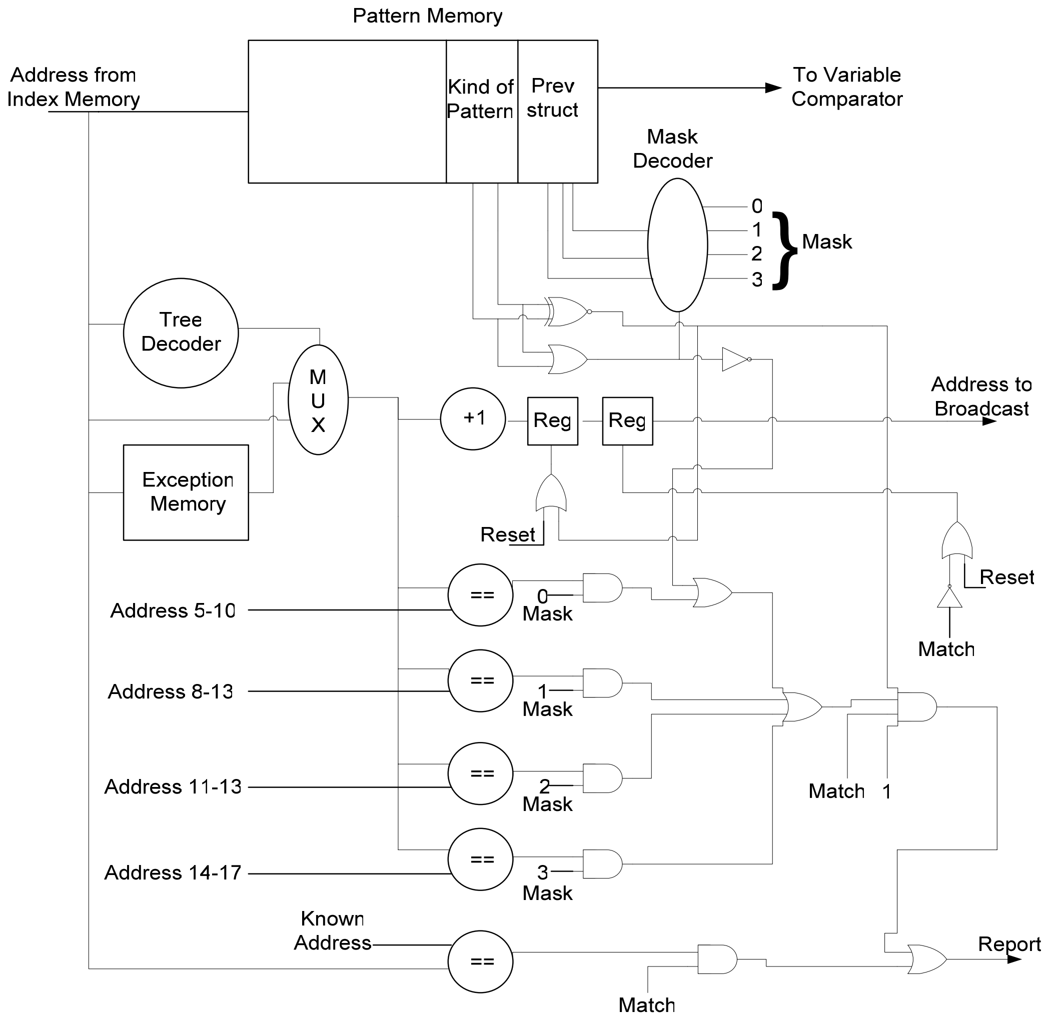

Figure 4.9: *Glue Logic Datapath. We use the data from Index Memory to access the Pattern Memory as we described before. We also use it to access the Exception Memory. The Mask Decoder determines which broadcasted address to use based on the saved information in Memory (Bits of Structure of Previous Fragment). The Control Signal of the Mask Decoder depends on the kind of the pattern. If we have a non fragment the output of the Mask Decoder must be zeros since there is nothing to be glued. As it is mentioned before we report a match when there is a match and it is a non partial pattern or the last fragment and that is the reason why we use this XNOR gate. Finally, we broadcast the address only when we have a first or intermediate fragment and there is a match. In any other case we broadcast zeros.*

After implementing the Glue Logic we pipelined it in order to achieve better cycle time. The duration of Glue Logic is about 10 cycles. Of course some signals are used few cycles after their creation and for that reason they are delayed appropriately using shift registers. Also, the broadcasting addresses are delayed appropriately before they are fed into the comparators. The area cost is 948 slices taking into account all the appropriate delays. The exception memories are implemented using the distributed memories of Xilinx which do not use memory blocks but logic and this logic cost is included in the area cost we mentioned before. However, we need to use slightly more memory blocks than these of table 4.4 since we put the extra bits in the pattern memories. We present the final memory and area cost in the next chapter (Evaluation and Results).

4.3 Final V-HashMem Architecture – Implementation and Evaluation

After completing the implementation of Glue Logic we have to replicate it and put it in every single structure except the 3-4 structure as it does not contain any fragment. Also, the V-HashMem Architecture consists of 5 structures and every single structure can report a match. For that reason we construct a priority encoder and determine the final match from all the matches. In the Figure 4.10, we show which structure has the biggest priority to determine the final match.

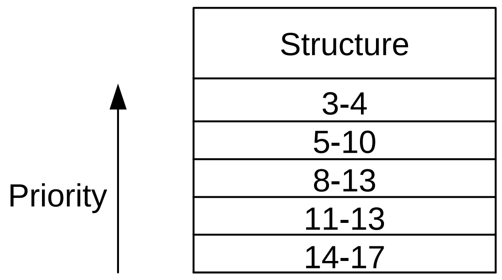

Figure 4.10: *Match Priority of the structures*

In Figure 4.11, we show one part of the Final V-HashMem Architecture. We do not show the whole V-HashMem Architecture since every single structure except the structure 3-4 are similar. Structure 3-4 does not contain any fragment as a result Glue is not needed. In this figure we also show how we report a long pattern match after gluing its fragments' matching signals. Also, the structure 14-17 is slightly different than the others as we mentioned in previous paragraphs as it consists of the 2 sub-structures: 14-17a and 14-17b. We put the Glue Logic in both sub-structures and since only one of them can report a match we choose the right one using a multiplexer.

The final area and memory cost as well as the performance of the V-HashMem Architecture are shown in the next chapter (Evaluation and Results). Finally, we have to mention that the above architecture has a single – byte (character) processing throughput per cycle. We can increase it and have a two – byte (character) processing throughput per cycle but this requires duplicating of the area since we have to use dual port memories, two CRC generators, two variable comparators and two pieces of Glue Logic per structure. The V-HashMem Architecture (only the 3-4 structure) with two input byte processing throughput per cycle is shown in Figure 4.12.

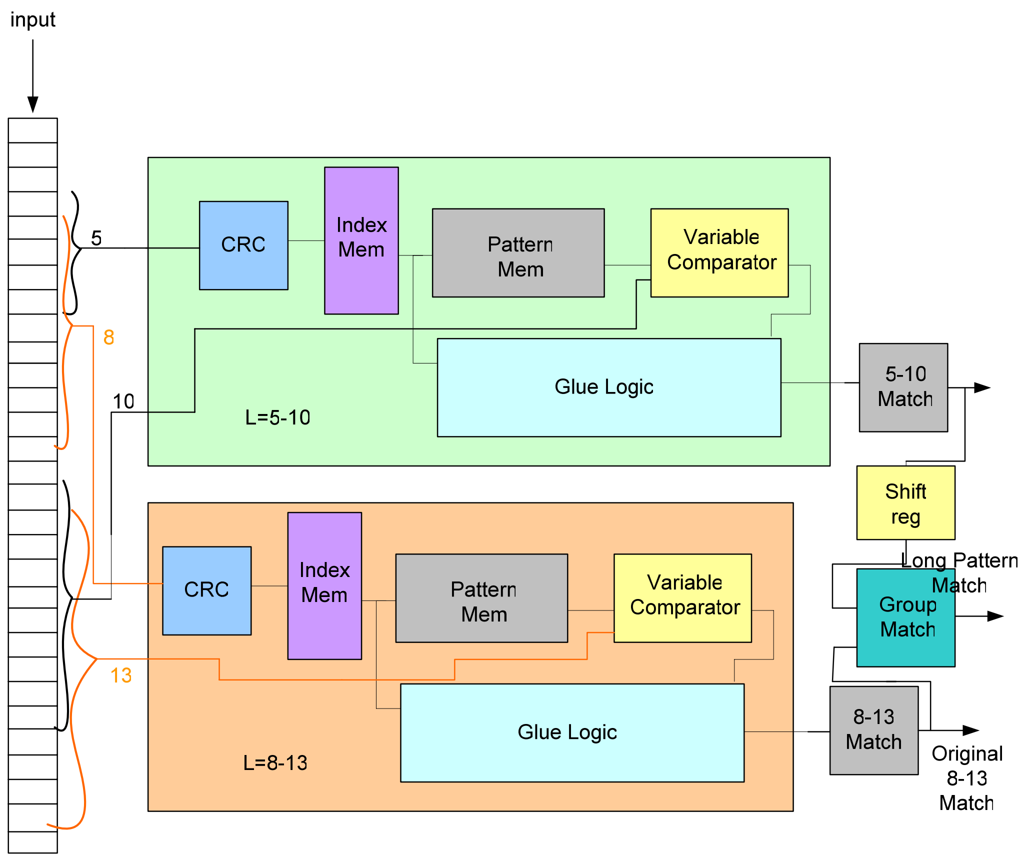

Figure 4.11: *Final V-HashMem Architecture. In this figure, we show only one part of the Final V-HashMem because the other structures are like these of Figure 3.18. Only structure 3-4 is different because it does not contain any fragment as a result Glue Logic is not needed.*

In this figure, we notice that we feed the CRC generators with 2 input streams at offsets 0 and 1 per cycle. Then, each output of the Index Memory is used to access the

Pattern Memory and each output of Pattern Memory is sent to a Variable Comparator which makes the comparison between the input stream, for offset 0 and 1, and the output of the Pattern Memory. Then each comparator's output is sent to a Glue Logic component to report the match or not. It is clear that by this improvement we almost double the throughput but we need to duplicate the area as a result the area cost is doubled.

Finally, we investigated the support for header matching information. A complete SNORT-like NIDS system combines two pieces of information: header matching with payload scan. It is easy to see that payload scan is by far the most difficult and complex task, since header matching generally involves merely equality or range matching on fixed numeric fields.

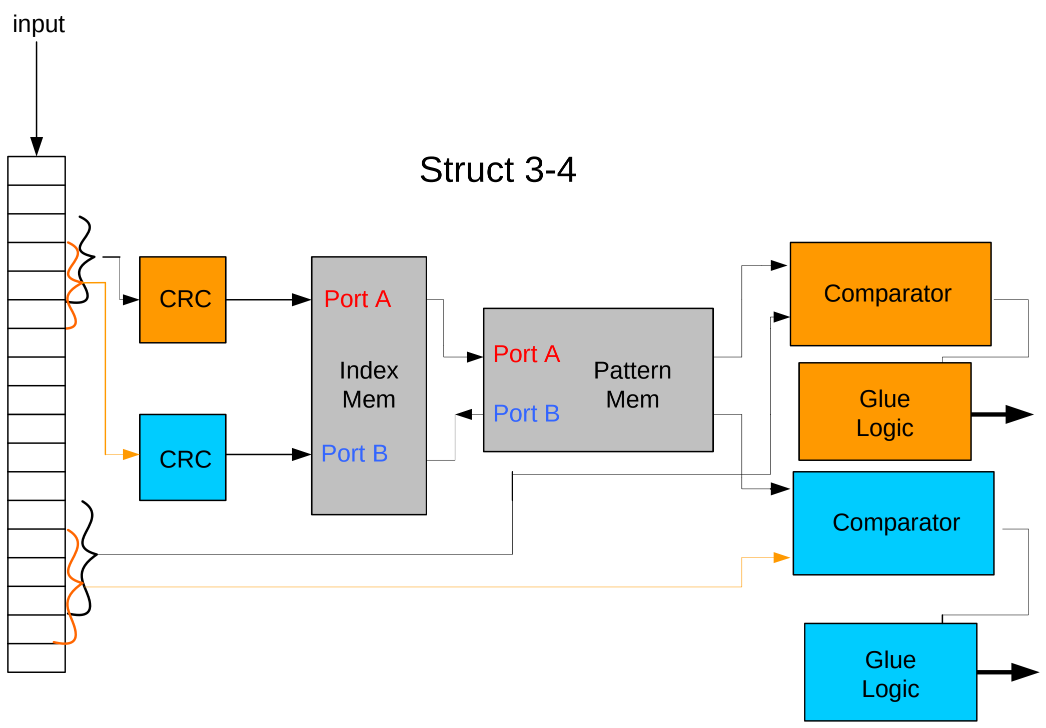

Figure 4.12: *In this figure, it is shown a part of the V-HashMem Architecture for doubled processing throughput (2 input characters processing throughput per cycle). However, we have to duplicate all the logic components (CRCs, comparators and Glue Logic) and use dual port memories. That means, that we have to double the area cost in order to achieve this doubled processing throughput.*

However, payload scanning should consider the header matching information and report matches only when the combined check of header and pattern are found. To include this functionality, we can add a *Header Group ID* field along with the search pattern in the pattern memories. This ID will determine the set of search patterns that are compatible with the current packet header. Earlier research of Dimopoulos *et al* [1] indicates that the snort rules can be classified into ~300 groups, which can be encoded with 9 bits. Upon arrival of a new input packet, the header matching circuitry performs its tests and provides the Header-ID to the V-HashMem sub-system. The pattern checking occurs as described earlier, but to report a match the comparators also test the Header-ID field for equality with that of the incoming packet. This test suppresses false positive answers when the search pattern is not compatible with the header of the packet. The additional cost of this feature is the memory bits to store the Header-ID and the additional comparator logic.

Of course, a complete system would also add the cost of header classification into the header group identifiers. This system could be like the one which is shown in Figure 4.13. The header data of a packet is inserted into the Header field extractor which performs header delineation, and field separation. In header classification, only six of all the possible header fields are necessary: source and destination IP address and ports, the protocol type and the ICMP type.

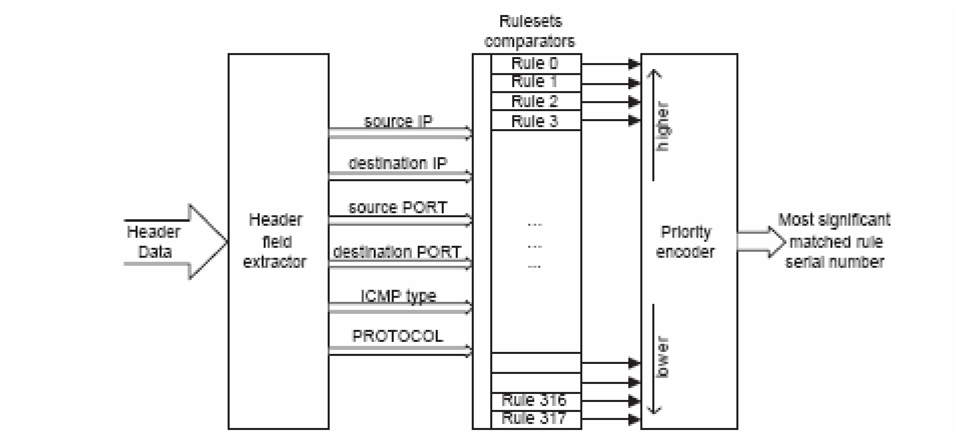

Figure 4.13: *Header Classification Circuitry which was proposed by B. Dimopoulos et al* [1].

The six header fields are forwarded to the rule set comparator module. The output of this module is a bitmask indicating all possible matching rules. The matches are strictly prioritized based on T-Gate (software which was proposed by Dimopoulos *et al* [1]) and SNORT. Therefore, the rule match indications are fed to a priority encoder to identify the most significant matched rule and to provide its encoding along with the packet data to the next processing level. We have to mention that the cost of the Header classification will not be included in our measurements in the next chapter.

Chapter 5

Evaluation and Comparison with Related Work

In chapters 3 and 4 we presented the V-HashMem Architecture. We presented the idea on which V-HashMem Architecture is based and then we described the design process and the implementation of it. Also, we proposed an idea in order to improve the throughput of it and another one to support Header matching. Now, we have to evaluate it and present the area cost as well as the performance and the throughput. These values will give us a clear view of the quality of our design. Furthermore we will compare our design with other related works.

First of all, we evaluate our system using the SNORT rule-set of April 2005 which contains 2187 rules or 33613 characters. We used Xilinx ISE 7.1i in order to implement our design and ModelSim in order to verify its correct functionality. The device families we used are VirtexIIpro, Spartan3 and Virtex4 and the device speeds are -7, -5 and -12 respectively. After completing the implementation we measure the area cost and the performance using the synthesis and place and route tools of ISE.

Before starting the evaluation of our design and present the results we will give some information to the readers about the metrics we use in order to evaluate our system. The metric we used to measure the area cost is the number of Logic Cells. The **number of Logic Cells** is the number of the Reported Slices multiplied by a factor of 2. Another metric which is based on the number of the measured Logic Cells is the number of **Logic Cells per character**. This number is the ratio of the number of Logic Cells and the total number of characters of the rule-set and it is a very useful metric since it shows how many logic cells we use in order to match a single character.

Furthermore we use some metrics in order to measure the speed of our design. Using Xilinx ISE synthesis tools we measure the **Performance (Operating Frequency)** of the system. Multiplying this Frequency with the input bits to the system per cycle we calculate the **Throughput**. Throughput is used widely by most researchers in order to evaluate their research. Moreover, we use another metric in order to combine the speed and area cost of our design. This metric is called **Performance Efficiency Metric (PEM)** and is also used by all researchers. PEM is the ratio of performance over the area cost or, in other words, the ratio of Throughput over the number of Logic Cells per Character. Finally, as our first goal was to reduce the number of the required Memory Blocks reducing the area cost and keeping the same the Performance we use another metric, **PEM/m** which shows clearly the efficiency of our design according to all the parameters we mentioned. In the following figure (Figure 5.1) we show all the metrics we use.

This chapter contains, at first, area and memory evaluation and utilization. Then we evaluate the Performance of our system. At paragraph 5.3 we evaluate V-HashMem Architecture again but combining Area, Memory and Performance results. Finally, we compare our implementation with related work.

$$\text{Number of LCs} = \text{Reported Slices} \times 2$$

$$\text{Throughput} = \text{Frequency} \times \text{Input bits}$$

$$\text{LCs/char} = \frac{\text{Number of Logic Cells}}{\text{Total Number of Characters}}$$

$$\text{PEM} = \frac{\text{Performance}}{\text{Area Cost}} = \frac{\text{Throughput}}{\text{LCs/char}}$$

$$\text{PEM/m} = \frac{\text{PEM}}{\frac{\text{Mem (kbits/characters)}}{100}}$$

Figure 5.1: *In this figure, the metrics we use for the evaluation of our system are shown. Also, the formulas which give these metrics are also shown.*

5.1 Area and Memory Evaluation

In chapter 4 we presented some first results on memory and area cost. Then, we described the design of the glue logic. In addition, we completed our design by adding the FIFO, from which the input stream is coming, and the priority encoder, which determines the final reported match and the broadcasted address. In the next tables we are going to show the area cost for every structure of the V-HashMem Architecture.

First of all, the area cost for every structure of the V-HashMem Architecture is shown in Table 5.1. Tree Logic contains the Exception Memory, the Tree Decoder and a MUX (Figure 4.9). Special List Logic contains a comparator, an AND gate, an OR gate and the appropriate delays (bottom of Figure 4.9) and +1 rule Logic contains all the remaining Logic of Figure 4.9.

Area Cost per Structure (Logic Cells)					
Structure	**CRC Generators**	**Comparators**	**Tree Logic**	**"+1 rule" Logic**	**Special List Logic**
3-4	62	26	0	0	0
5-10	64	62	30	80	10
8-13	82	82	44	90	20
11-13	46	74	90	90	10
14-17	108	200	160	160	20
Total	362	444	324	420	60

Table 5.1: *This table shows the Area cost in Logic Cells (LCs) for each and every structure of the V-HashMem Architecture. Tree Logic contains the Exception Memory, the Tree Decoder and a MUX (Figure 4.9). Special List Logic contains a comparator, an AND gate, an OR gate and the appropriate delays (bottom of Figure 4.9) and "+1 rule" Logic contains all the remaining Logic of Figure 4.9.The cost for the CRC generators and the Comparators is the same before and after the adding of Glue Logic. Notice that Glue Logic is the sum of Tree Logic, Special List Logic, +1 rule Logic and the appropriate delays for the broadcasting addresses. For structure 3-4 we observe that there is no Tree Logic, "+1 rule" logic or Special List Logic Cost because this structure does not contain fragments as a result there is no Glue Logic.*

The cost for the CRC generators and the Comparators is the same before and after the adding of Glue Logic. Notice that Glue Logic is the sum of Tree Logic, Special List Logic, "+1 rule" Logic and the appropriate delays for the broadcasting addresses. The last one is not shown in the Table 5.1. In the next table (Table 5.2) we are showing the total area cost for every component that is contained in the V-HashMem Architecture as well as the Total Area Cost of the final implementation of it. The medium column shows the Area Cost for processing Throughput 1 character per cycle. At the end of chapter 4 we proposed an improvement of V-HashMem Architecture in order to increase the Throughput by increasing the processing Throughput from one byte (character) to two bytes (2 characters) per cycle. This improvement almost doubles the required Area Cost as every single component must be duplicated except FIFO as it was shown also in Figure 4.12. Adding Header-ID matching, we have to add 80 and 160 LCs, which is the cost of the Header comparators, for 1 input character and 2 input characters per cycle respectively.

Area Cost		
Component	**Processing Throughput: 1 character per cycle.**	**Processing Throughput: 2 characters per cycle.**
CRC generator	362	724
Comparator	444	888
FIFO	250	250
Glue Logic	948	1896
Priority Mux	80	160
Total	**2084**	**3918**

Table 5.2: *Total Area Cost for every Component of the V-HashMem Architecture and the Total Area Cost of the final implementation of the V-HashMem Architecture for processing Throughput (TH) 1 and 2 characters per cycle.*

Regarding Memory, we showed in chapter 4 the memory cost for the basic V-HashMem Architecture but this is not the final memory cost since we put 5 extra bits in the Pattern Memories in order to encode significant information about the kind of the Pattern and the source structure of a previous fragment as it is mentioned in paragraph 4.2.1. In Table 5.3 and 5.4 we show how many memory blocks are used in the final implementation for the Pattern Memories and Index Memories respectively.

Pattern Memory Cost				
Structure	**Entries**	**Width(bits)**	**Number of memory blocks**	**Header ID**
3-4	512	33	1	+1
5-10	512	90	3	+0
8-13	512	114	4	+0
11-13	1024	111	7	+0
14-17A	512	144	4	+1
14-17B	512	144	4	+1
Sum			**23**	+3

Table 5.3: *In this table, it is shown how many memory blocks are used in order to implement the Pattern Memories of V-HashMem. Comparing this Table with Table 4.4 we notice that the number of memory blocks in the final implementation is only increased from 22 to 23 memory blocks.*

Comparing Table 5.3 with Table 4.4 we notice that the number of memory blocks in the final implementation is only increased from 22 to 23 memory blocks. We have only this small increase because the maximum width of the Memory Blocks of all the pattern memories except the one of structure 11-13 was enough to accept the adding of the 5 extra bits. In structure 11-13 we just add another memory block. Regarding the memory blocks which are used for the implementation of the Index Memories we mention that the cost is the same as it is shown also in Table 4.5. Furthermore, there is an amount of 2 additional memory blocks for the implementation of the very short strings (of one or two characters). So, the total number of Memory Blocks we used in order to implement the memories of our design is 39. Taking also into account the Header support, the stored Header IDs make the pattern memories wider. Some memories have spare bits and can accept this extension while other do not; overall, storing the Header-ID along with the patterns requires an additional 3 memory blocks (Table 5.3), a small price for the increased functionality. Finally, we will not describe why we used this number of memory blocks for the implementation of Index and Pattern Memories since it was described in much detail in paragraph 4.1.3.

Index Memory Cost			
Structure	**Number of Entries**	**Width[bits]**	**Number of memory blocks**
3-4	2048	9	1
5-10	4096	9	2
8-13	4096	9	2
11-13	8192	10	5
14-17A	4096	9	2
14-17B	4096	9	2
Total			**14**

Table 5.4: *In this table it is shown how many memory blocks are used for the implementation of the Pattern Memory, for each structure and for the total implementation.*

We evaluated our implementation and we exported the results for the area cost. Before evaluating our design about the Performance lets see which is the area and the memory utilization since it is also an important factor of the efficiency of our implementation.

Firstly, in Table 5.5 we show the memory utilization for the Pattern Memories of the V-HashMem Architecture. Observing this Table we can notice that we achieve a satisfactory utilization except for the memory structure 3-4 in which the utilization is modest.

Pattern Memory		
Structure	**Number of Patterns**	**Memory Utilization (%)**
3-4	169	33
5-10	372	72.6
8-13	509	99.4
11-13	715	69.8
14-17A	456	89
14-17B	345	67.3

Table 5.5: *In this table it is shown the Pattern Memory's Utilization. Looking at this Table we notice that we achieve very satisfactory memory utilization. Only for the memory structure 3-4 memory utilization is modest but we cannot do something better as the patterns which are stored are neither so short nor so long to handle them either on the way we handled patterns of 1 or 2 characters or on the way we handled long patterns.*

In Table 5.6 we show the memory utilization for the Exception Memory which was suggested in order to solve the Tree – Like structures. According to this table we also achieve very good utilization. Finally, Exception Memories are not used in the Structure 3-4 because this structure does not contain any fragment. The exception memory was constructed using the Distributed Memories of Xilinx, which do not use memory blocks but logic. We choice to use Distributed Memories and not block memories because the number of Data, as it is shown in Table 5.6, is very small. If we used block memories we would use 5 times the memory block 512x36 and achieve 12% utilization in the best case (14-17B).

Exception Memory		
Structure	**Number of Data**	**Memory Utilization (%)**
5-10	3	18.8
8-13	21	65.6
11-13	58	60.4
14-17A	33	51.2
14-17B	63	49.2

Table 5.6: *In this table it is shown the Utilization for the Exception Memories. Looking at this Table we notice that we achieve very satisfactory memory utilization.*

In addition, in Figure 5.2 we show the Area Utilization of V-HashMem Architecture and we notice that Glue Logic takes up about the half of the used Area of it while CRC generators and the Comparators take up 17% and 21% of the total used Area.

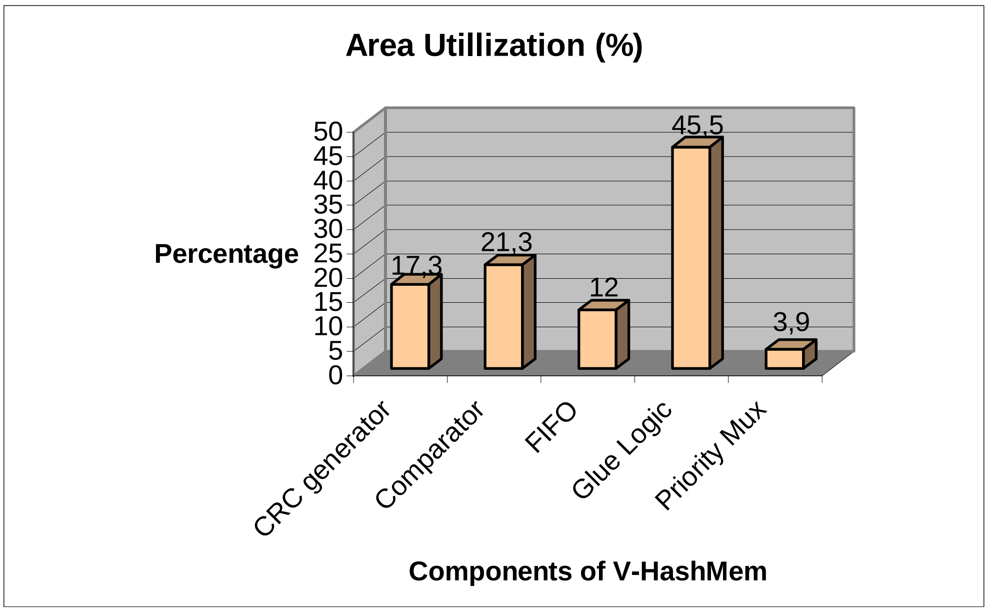

Figure 5.2: *In this Figure, it is shown how much area is used by every component of the total 2084 Logic Cells of V-HashMem Architecture. Figure 5.2 would be similar if we calculated the utilization of the Area for processing Throughput 2 characters per cycle.*

To analyze Device Area and Memory Utilization we used three different devices of the same Device family VirtexIIpro to implement the design, within and without the improvement we mentioned in the last paragraph of chapter 4. Table 5.7 shows this Device Area and Memory Utilization for the above cases. Observing Table 5.7, we can notice that V-HashMem Architecture within or without the improvement can fit in a device of medium size as well as a small device.

	Device Area Utilization (%)		Device Memory Utilization (%)	
Devices	Processing throughput 1 character/cycle	Processing throughput 2 characters/cycle	Processing throughput 1 character/cycle	Processing throughput 2 characters/cycle
VirtexIIpro 7	21.1	39.8	84.1	84.1
VirtexIIpro 20	11.2	21.1	42.1	42.1
VirtexIIpro 30	7.6	14.3	27.2	27.2

Table 5.7: *In this Table, we show the Area and Memory Utilization for 3 different devices for both implementations within or without improvements.*

We will give the area and memory cost, using the metrics we mentioned in the introduction of this chapter, after the performance evaluation on a summary table (Table 5.9).

5.2 Performance Evaluation

For our implementation we used 3 different Device families: VirtexIIpro, Spartan3 and Virtex4 and measured the Frequency of the system. Then we calculated Throughput and the results are shown in Table 5.8.

	Frequency (MHz)		Throughput (Gbps)	
Device Family	Input bits: 8	Input bits:16	Input bits: 8	Input bits:16
VirtexIIpro	320.146	~309	2.56	~4.95
Spartan3	190.387	~176	1.52	~2.82
Virtex4	346.021	~340	2.77	~5.44

Table 5.8: *In this table the Frequencies and the TH for V-HashMem Architecture with one and two input characters (8 and 16 input bits respectively) for 3 different device families are shown.*

These results are for specific devices of each Device Family. We used the devices: xc2vpx20, xc3s3000 and sc4vsx35 for the device families VirtexIIpro, Spartan3 and Virtex4 respectively. All these devices have medium size. In addition, we measured the Frequencies using the modes Advanced 1.90 2005-01-22, Advanced 1.35 2005-01-22 and Preview 1.52 2005-01-22 of Place and Route tool of Xilinx ISE for the device families VirtexIIpro, Spartan3, Virtex4 respectively.

According to table 5.8, using VirtexIIpro for the implementation with 1 input character processing throughput (without the improvement) we achieve a very good Throughput, near 2.5 Gbps. Also for Virtex4 Device family, which is faster and better than VirtexIIpro, we achieve Frequency 346.021MHz for the same implementation. On the other hand, Frequency and Throughput for both implementations on Spartan3 Device Families are not impressive. But we do have an explanation for these results: Spartan3 Device Families have generally low speed in comparison with other Device Families, such as VirtexIIpro and Virtex4 and are really cheap and have low energy consumption.

5.3 Total Evaluation and Results

In Table 5.9, we present the total results for V-HashMem Architecture with 1 and 2 input characters (meaning without or within the improvement) using also all the metrics we mentioned at the introduction of this Chapter.

V-HashMem Architecture: Area, Memory and Performance Evaluation									
	Num of Patterns	Patterns Chars	LCs	LCs/char	Mem Blocks (kbits)	Frequency (MHz)	TH (Gbps)	PEM	PEM/m
Input Char 1	2187	33613	2084	0.06	39 (648.32)	320	2.560	42.66	22.45
Input Chars 2	2187	33613	3918	0.11	39 (648.32)	309	4.948	44.98	23.67

Table 5.9: *Table with total results for the V-HashMem Architecture for Input 1 and 2 characters. This Table contains the results for all the metrics which were mentioned in the introduction of Chapter 5.*

Looking at Table 5.9, we notice that for the first approach we achieve Frequency 320 MHz and Throughput 2.560 Gbps using only 2084 Logic Cells or 0.06 LCs per character. On the other hand, for 2 input characters (doubled processing throughput) although we achieve slightly lower performance (309 MHz), we almost doubled the Throughput (4.948 Gbps). To achieve this we needed to use Dual port Memories and duplicated components for each memory structure as a result the Area Cost almost doubled. Furthermore, Performance Efficiency Metric is better in the second approach than the one of the first approach. Finally, if we consider the fact that we stored a really great number of patterns using only 39 memory blocks, using about 20% and 40% of the total area of a small FPGA and achieving Frequencies 320MHz and 309 MHz for both approaches we can say that V-HashMem Architecture is a very efficient design.

5.4 Comparing V-HashMem Architecture with HashMem Architecture

In the previous paragraphs we evaluated V-HashMem Architecture in order to see how efficient our design is. However, this is not enough. It is very crucial to compare our work with other related works. At first we are going to compare V-HashMem Architecture with HashMem Architecture which was designed and implemented by George Papadopoulos and Dionisios Pnevmatikatos [2].

As we described in chapter 3, HashMem Architecture contains 14 memory structures where patterns from 3 to 16 characters length are stored. Also there is one memory structure where very short patterns (of one or two characters) are stored there. Finally, patterns longer or equal 17 characters are broken into shorter strings and are stored in the existed memory structures. Extra glue logic is used to combine the matching signals of the partial patterns in order to report the final long match. The main difference of HashMem Architecture and V-HashMem Architecture is that the in the first one every single memory structure contains patterns of the same length while in our design patterns of different but close lengths are stored in the same memory.

In V-HashMem Architecture, the resulting structures use a total of 74 character comparators, compared to 150 in HashMem, a 50% improvement, and require 5 CRC generators with a total of 41 character input, compared to 150 character input for HashMem as it is shown in Table 5.10.

V-HashMem	**Width (characters)**	
Structures	**CRC generator**	**Comparators**
3-4	3	4
5-10	5	10
8-13	8	13
11-13	11	13
14-17	14	34
Total	**41**	**74**
HashMem	150	150

Table 5.10: *V-HashMem Architecture compared with HashMem Architecture on the width of CRC generators and Comparators.*

Given the observation that in HashMem CRC generation and comparators account for 32% and 42% of the logic respectively, there is the potential for considerable logic savings (in the order of 40%), despite using the newer SNORT rule-set with 70% more characters. On the other hand, the newer rule-set also includes many more wide strings that have glue logic overhead, reducing the potential improvement.

The memory usage of our V-HashMem configuration is also depicted in tables 5.4 and 5.5. As we said, 23 memory blocks are needed for pattern storage and 14 for index memories. In addition to these memories we need another 2 memory blocks to match the very narrow 1 and 2 character patterns. This leads to a total of 39 memory blocks. The best HashMem configuration used 31 memory blocks but to store significantly fewer patterns and using both memory read ports. This best HashMem approach, which is named as "HashMem + Reuse + Share + Small CRCs", is an improved version of the HashMem we described in Chapter 3. In this approach, the memory is partitioned in two independent portions. The "upper" portion is used for patterns of width X and the other is used for patterns of widths Y (usually X+1). This approach was possible because of 1) the low density of the indirection memory and pattern memory in the HashMem Architecture and 2) that Xilinx memories are dual ported. However, in this approach they could not increase the processing throughput from one to two characters per cycle since both ports of memory are used now.

Table 5.1 shows the area cost of each sub-system of our V-HashMem architecture. We break down the cost into (a) the character FIFOs that accumulate the characters for the CRC generators and the comparators, (b) CRC generators, (c) comparators, (d) glue logic for partial matches, and (e) final address and match reporting circuitry (priority encoder). The total calculated logic cost is 2086 Logic cells, or a cost of ~0.06 LCs/character (Table 5.9). This corresponds to a per character improvement of 50% over the unoptimized HashMem. In Figures 5.3 – 5.6 we compare the two approaches of V-HashMem Architecture with the best HashMem Architecture over 5 parameters.

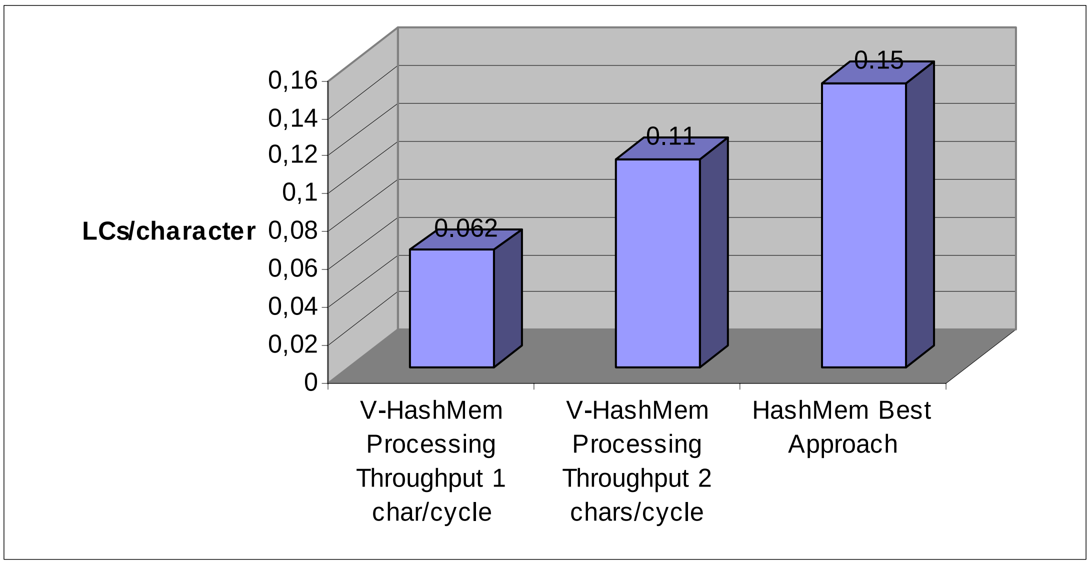

Figure 5.3a: *Comparing V-HashMem with the best HashMem over LCs/char.*

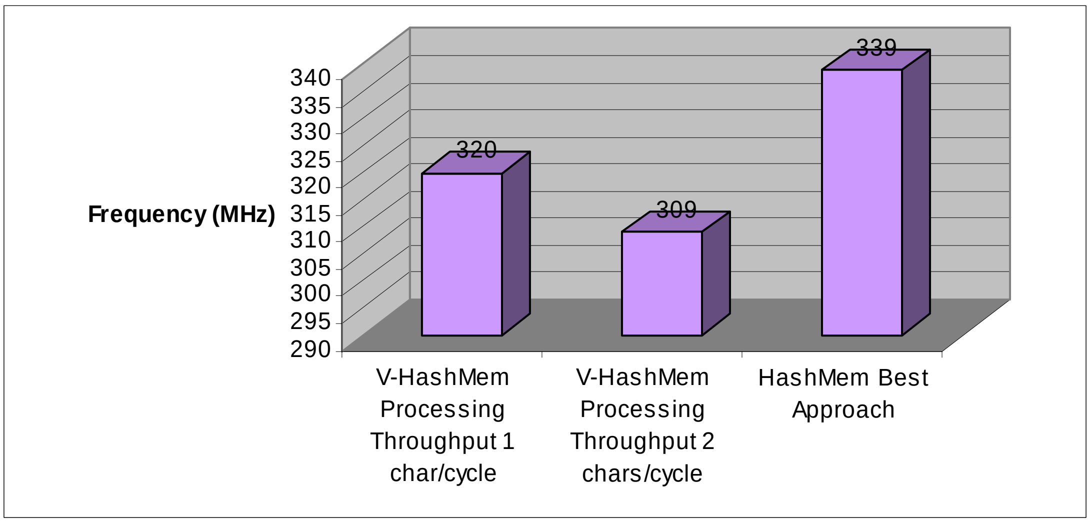

Figure 5.3b: *Comparing V-HashMem with the best HashMem over Frequency using VirtexIIpro Device Family.*

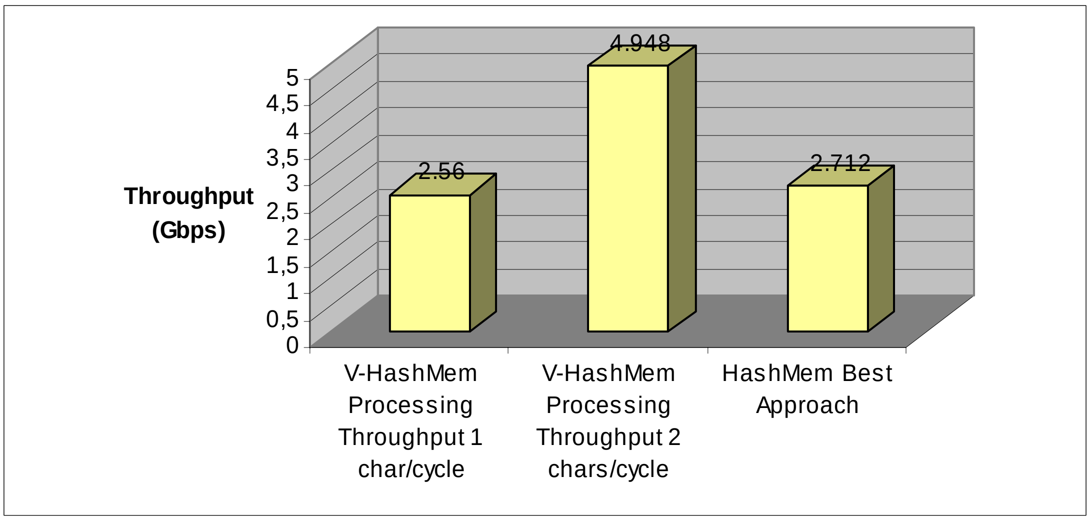

Figure 5.4: *Comparing V-HashMem with the best HashMem over Throughput.*

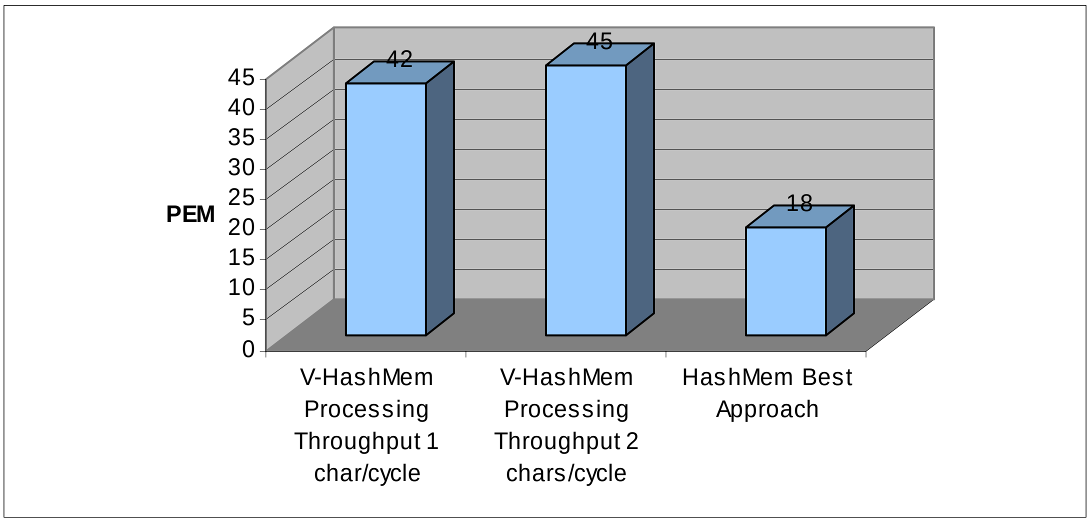

Figure 5.5: *Comparing V-HashMem with the best HashMem over PEM.*

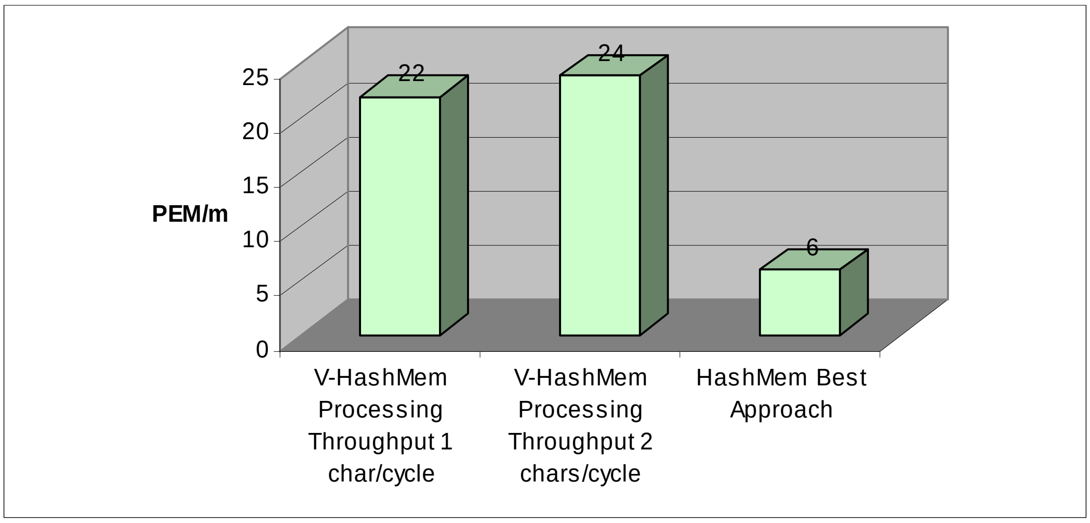

Figure 5.6: *Comparing V-HashMem with the best HashMem over PEM/m.*

Looking at figures 5.3b and 5.4 we observe that although HashMem Architecture achieves better Frequency than both approaches of V-HashMem Architecture, the second approach of V-HashMem Architecture achieves almost doubled Throughput. Looking at Figure 5.5, we notice that both approaches of V-HashMem Architecture achieve more than doubled PEM. Furthermore, looking at Figure 5.6, we notice that PEM/m in both approaches of our design is 3 or 4 times better than PEM/m of HashMem, even though we use 8 memory blocks more. The combination of these results and the low area cost per character in comparison with the one of the HashMem Architecture taking into account also that the rule-set we use contains 70% more characters than the rule-set which was used in HashMem approach shows clearly that V-HashMem Architecture is more efficient than the HashMem Architecture (best approach).

5.5 Comparing V-HashMem Architecture with Related Works

In the recent years many pattern matching architectures have been proposed specifically for accelerating a SNORT-like NIDS using FPGAs. The architectures differ in the approach (finite automata or CAM-like), in their internal organizations, and obviously in their cost-performance tradeoffs [6, 7, 8, 9, 10, 11, 12, 13, 3, 4, 5]. These works strive for lower cost, at the same or better performance. V-HashMem is based on two ideas: (i) the use of simple hashing to summarize the multiple input bits (also used in Bloom filters [8, 4]), and (ii) the use of memories to provide exact match with fewer gates (also used by Cho *et al*. [10, 11], Sourdis *et al.* [3]).

The use of Bloom filters for pattern matching has been proposed by Dharmapurikar for low cost pattern matching [4, 8]. Bloom filters are very elegant in representing set membership, but suffer two potential drawbacks: (i) they require multiple hash functions and memories, and (ii) they give an approximate match answer since they allow false positives. Solutions to these limitations exist but at additional cost. In this architecture, it may be possible that a total amount of 420000 characters can be stored. However, the stored patterns can only be between 2 and 26 characters long. On the other hand, V-HashMem Architecture uses fewer hash functions and eliminates the false positives by exactly defining the intrusion pattern using two-level memories. Furthermore, the number of stored patterns is only 33613 characters. This is the SNORT rule-set of April 2005 and the length's range of the stored patterns is not kept between 2 and 26 characters but between 1 and 122 characters. Also, V-HashMem Architecture can be easily updated to include even more patterns without significant overhead in area cost and with little increase in memory cost.

In figures 5.7 to 5.11, we compare V-HashMem Architecture with Bloom Filters over the same parameters used for the comparison between V-HashMem Architecture and HashMem Architecture. Looking at these figures we notice that both approaches of V-HashMem Architecture achieve about 5 times better Frequency, about 5 and 10 times for first and second approach respectively better throughput and about 7 times better PEM than the ones in Bloom Filters. Of course, the performance of VirtexE 2000 which was used for the implementation of Bloom Filters is worse than

VirtexIIpro but not worse than 35%. So the slowest V-HashMem Architecture is 210% faster than Bloom Filters. On the other hand, we achieve better area cost using the first approach but worse using the second approach in comparison to with Bloom Filters.

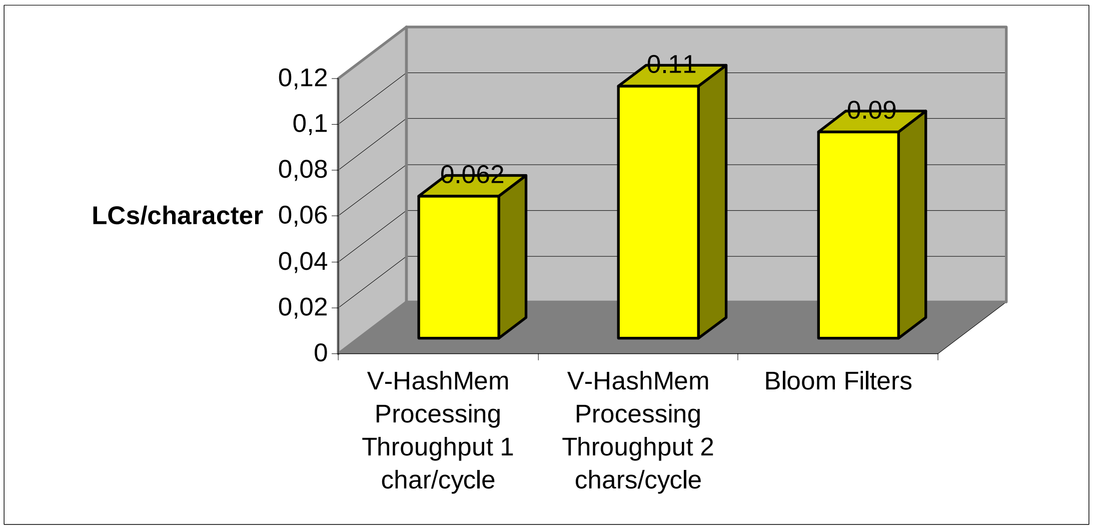

Figure 5.7: *Comparing V-HashMem with the Bloom Filters over LCs/char.*

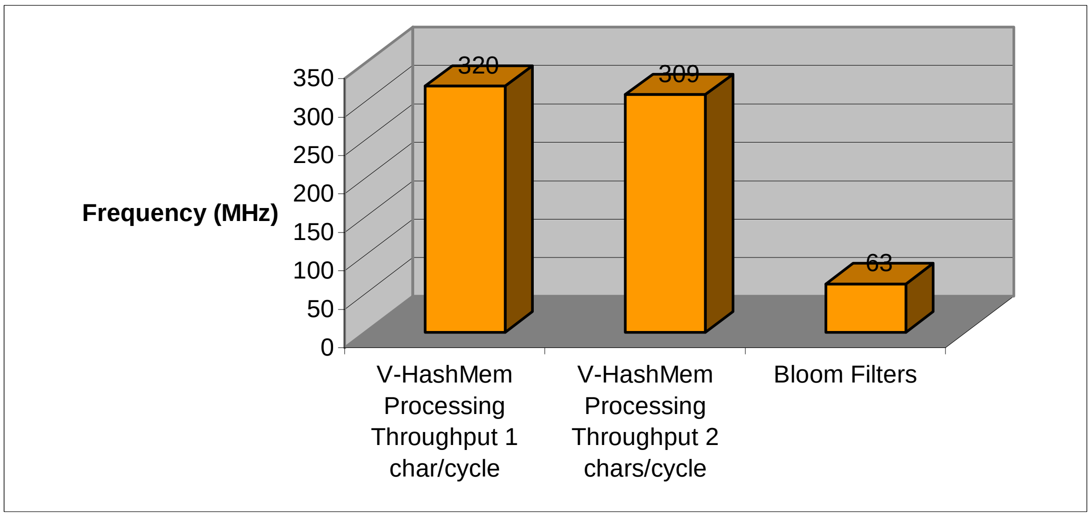

Figure 5.8: *Comparing V-HashMem with the Bloom Filters over Frequency.*

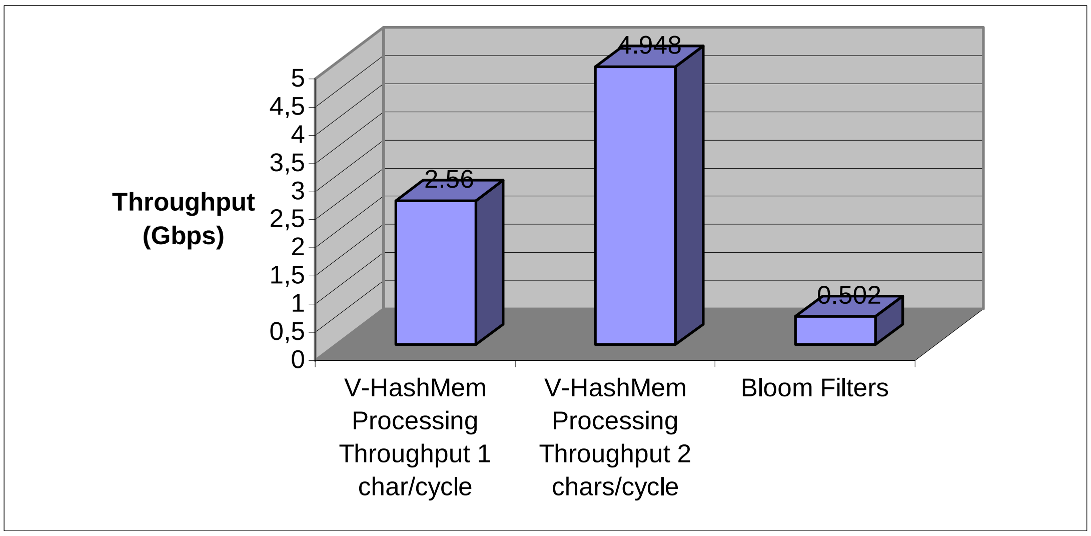

Figure 5.9: *Comparing V-HashMem with the Bloom Filters over Throughput.*

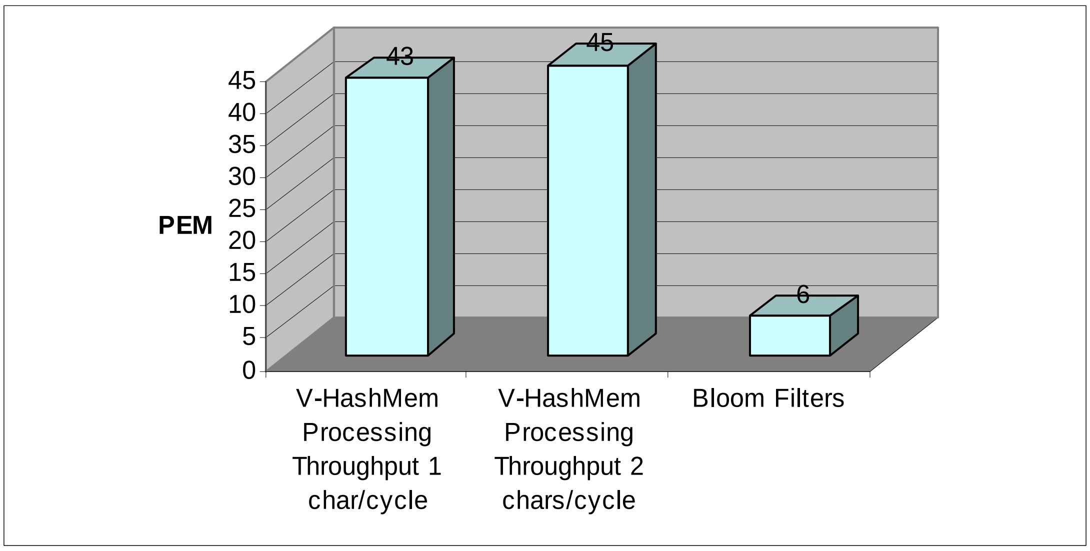

Figure 5.10: *Comparing V-HashMem with the Bloom Filters over PEM.*

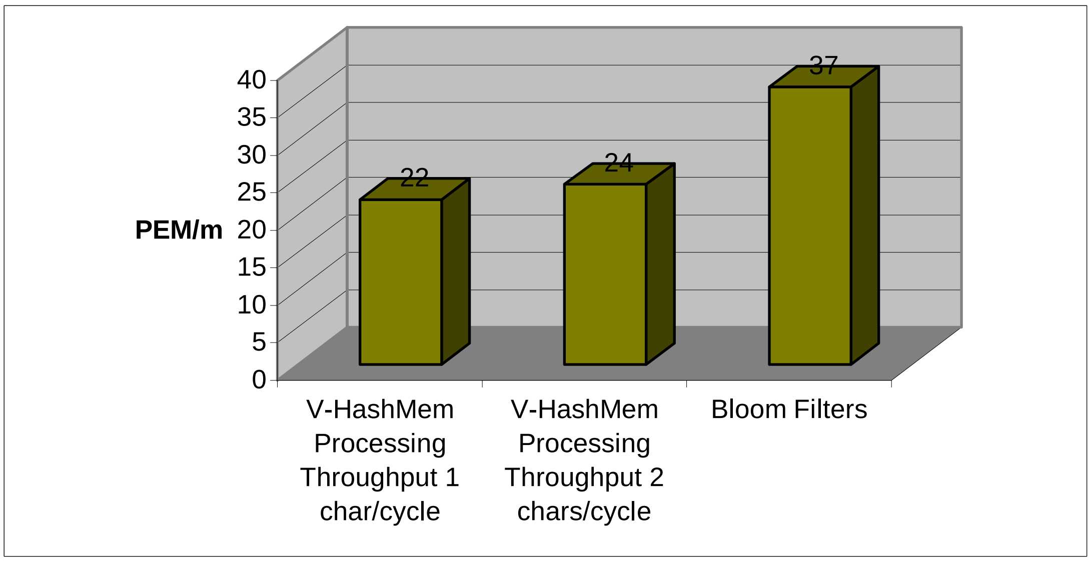

Figure 5.11: *Comparing V-HashMem with the Bloom Filters over PEM/m.*

Finally, PEM/m in Bloom Filter is 1.5 times better than ours. However, if they used the implemented number of characters and not the number of characters their system was able to store the PEM/m would become much lower and possibly worse than ours.

Another approach (RDL+ROM) was proposed by Cho and Mangione-Smith [10]. They used a CAM to match short patterns and to match unique prefixes of longer search patterns. They choose the CAM width so as to provide unique prefix signals for each possible match. The match signals for all prefixes are then encoded to provide a memory address where the candidate suffixes are stored. The remaining input is compared against the expected suffix, and the result is the overall match for the pattern. Their approach offers very good memory density and low gate count. The cost of this approach however increases if the patterns have many and long common prefixes.

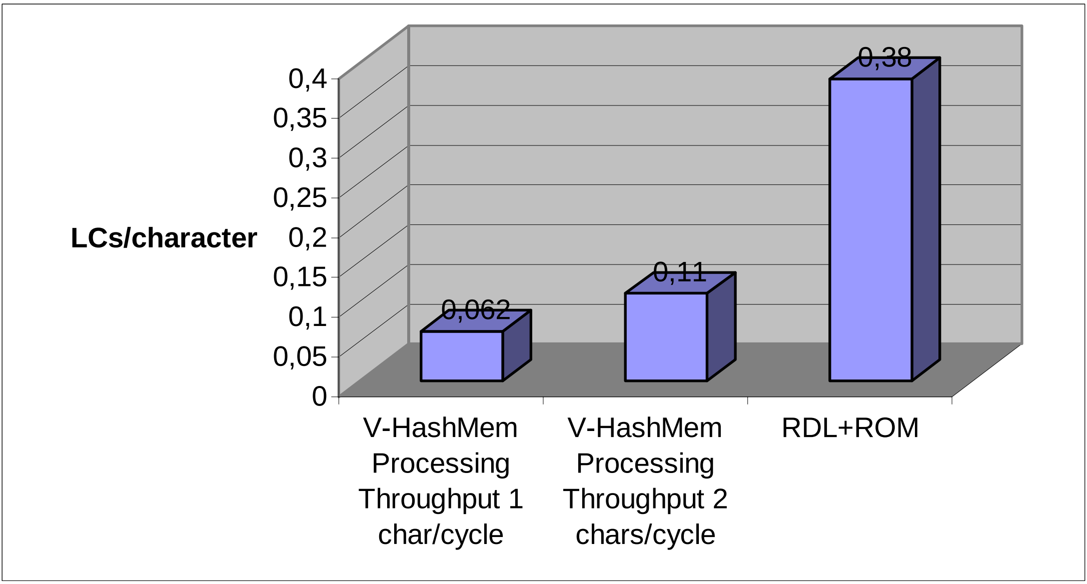

Figure 5.12: *Comparing V-HashMem with the RDL+ROM over LCs/char.*

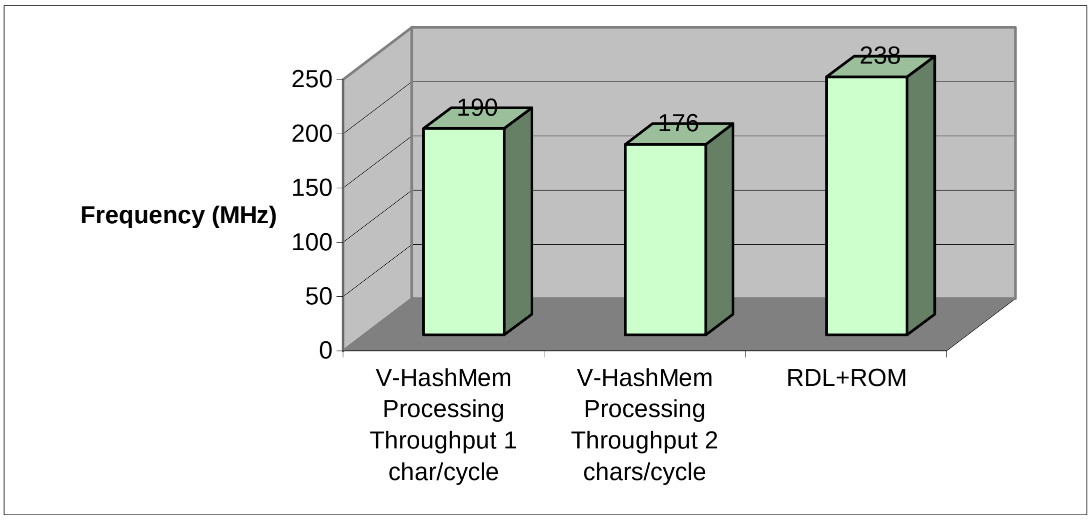

Figure 5.13: *Comparing V-HashMem with the RDL+ROM over Frequency.*

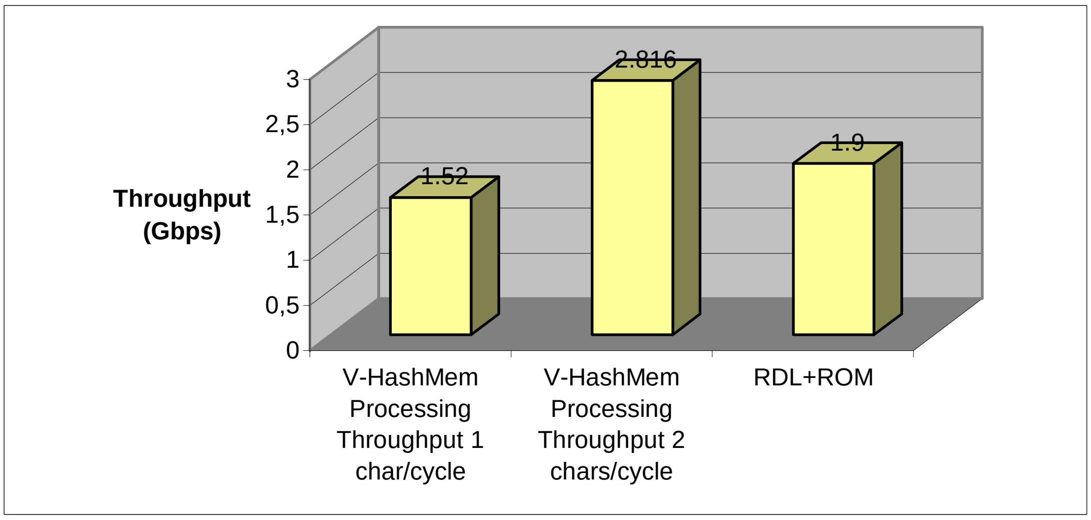

Figure 5.14: *Comparing V-HashMem with the RDL+ROM over Throughput.*

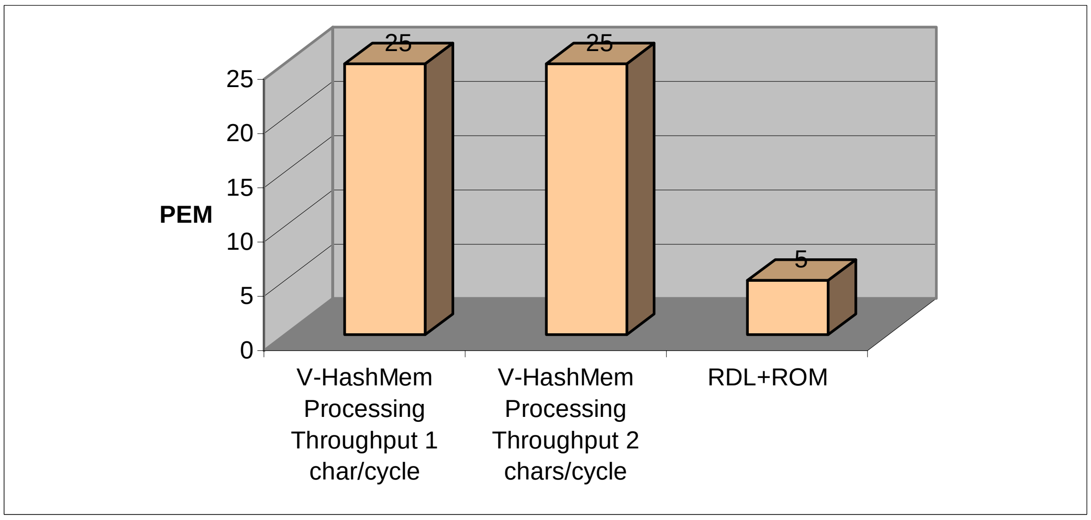

Figure 5.15: *Comparing V-HashMem with the RDL+ROM over PEM.*

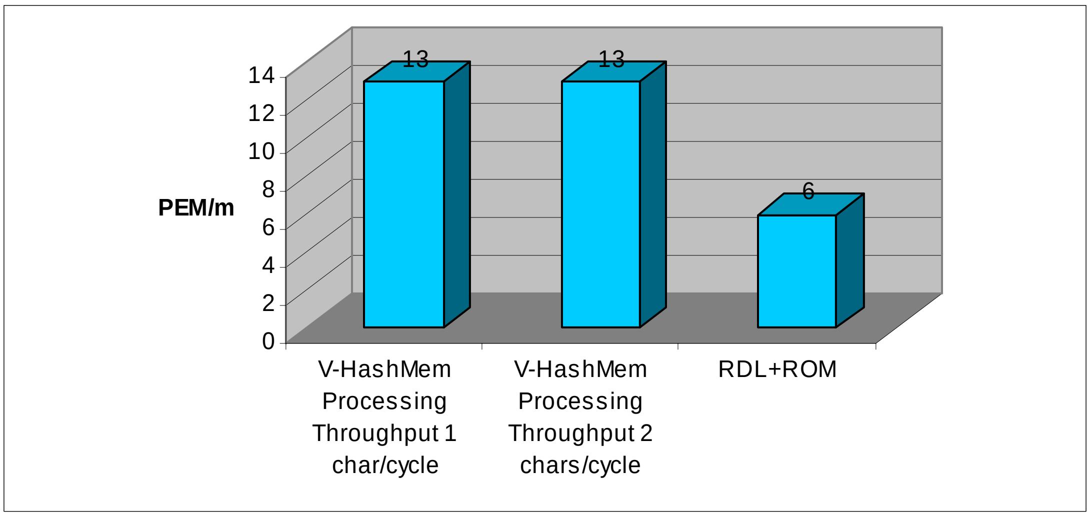

Figure 5.16: *Comparing V-HashMem with the RDL+ROM over PEM/m.*

In figures 5.12 to 5.16 we compare V-HashMem Architecture with RDL+ROM approach using Spartan3 Device Family. Looking at figure 5.12 we use much fewer LCs per character in both approaches than RDL+ROM. As long as Performance is concerned, RDL+ROM achieves much better Frequency than V-HashMem Architecture but observing figure 5.14 we notice that the second approach of V-HashMem Architecture achieves much better Throughput than RDL+ROM. Finally, PEM and PEM/m are about 400% and 100% respectively better than the ones of RDL+ROM approach making V-HashMem Architecture more efficient than RDL+ROM considering also that the stored characters in V-HashMem are about 60% more than the ones in RDL+ROM.

The final comparison will be done with Perfect Hashing which is similar to HashMem and was proposed by Sourdis *et al.* [3]. Sourdis *et al* have used a centralized memory based pattern matching, where the memory location is selected using a *perfect hashing* of selected input bits. This approach shares many of the advantages of HashMem, and achieves even better memory usage but at a higher logic cost.

In figures 5.17-5.21 we compare V-HashMem Architecture with PHmem over the parameters we used in the previous comparisons. Looking at figure 5.17 in each and every approach we achieve much better area cost per character. On the other hand, Frequency and Throughput in PHmem in both cases is better than the ones of V-HashMem Architecture. Also in both cases, less memory blocks are used in PHmem than ours but the stored patterns were 60% less than the ones in V-HashMem. Finally, comparing PEM and PEM/m for the first approach between the 2 architectures we notice that V-HashMem Architecture is better by a factor of 7 and 10 respectively than PHmem. Also, and for the second approach in V-HashMem these values are 5 and 8 times better than the ones of PHmem.

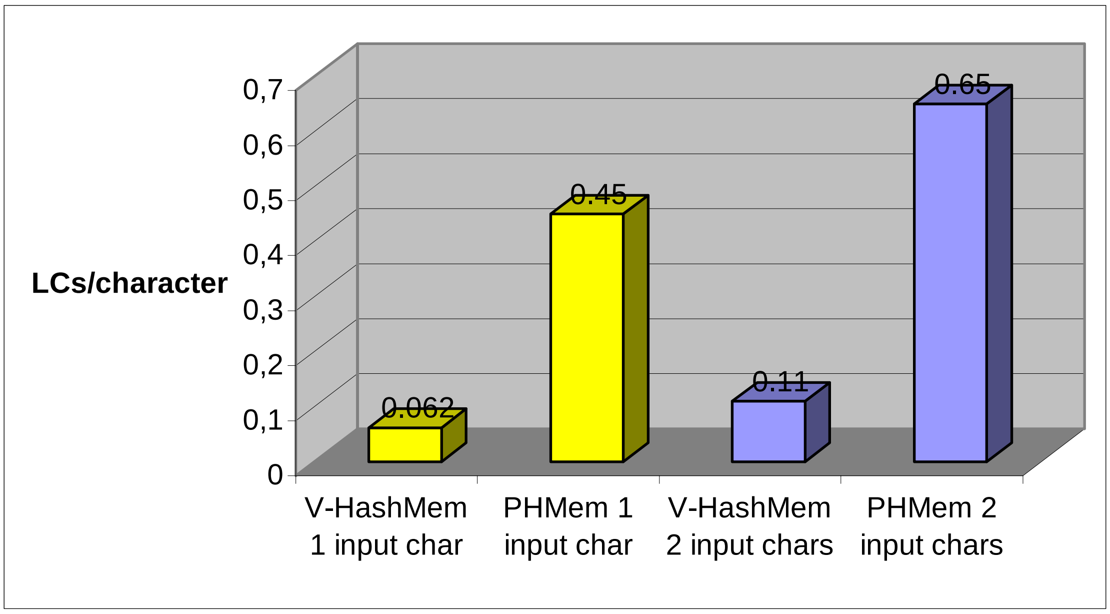

Figure 5.17: *Comparing V-HashMem with the PHmem over LCs/char.*

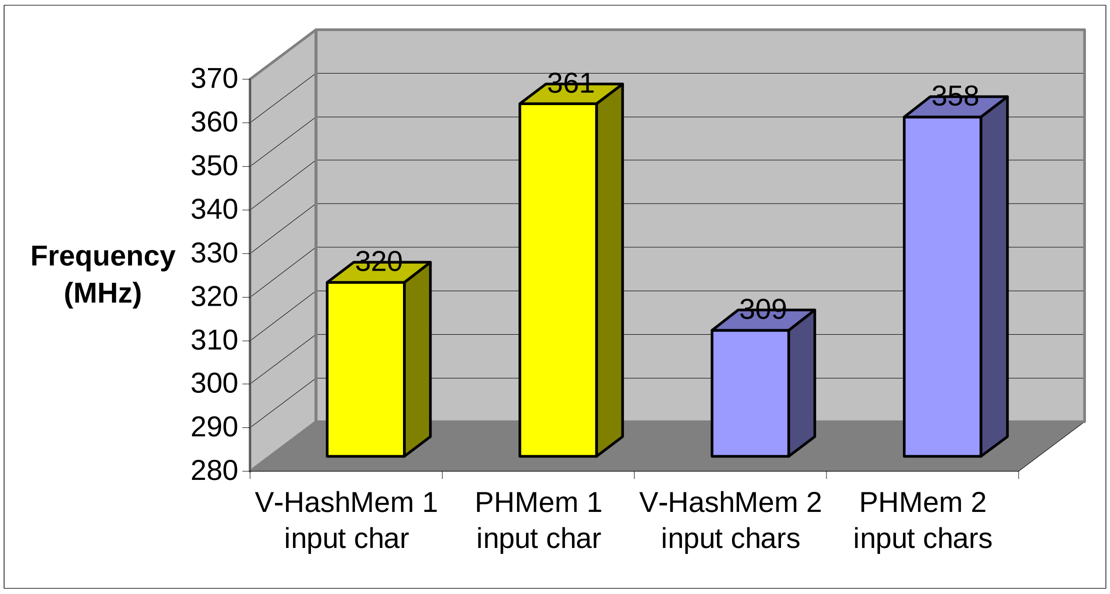

Figure 5.18: *Comparing V-HashMem with the PHmem over Frequency.*

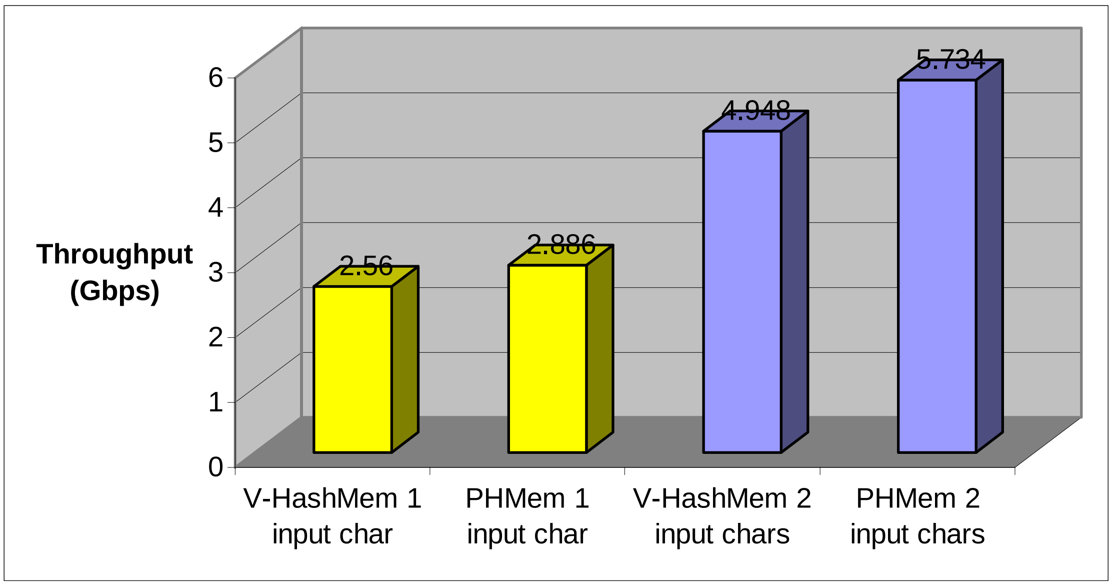

Figure 5.19: *Comparing V-HashMem with the PHmem over Throughput.*

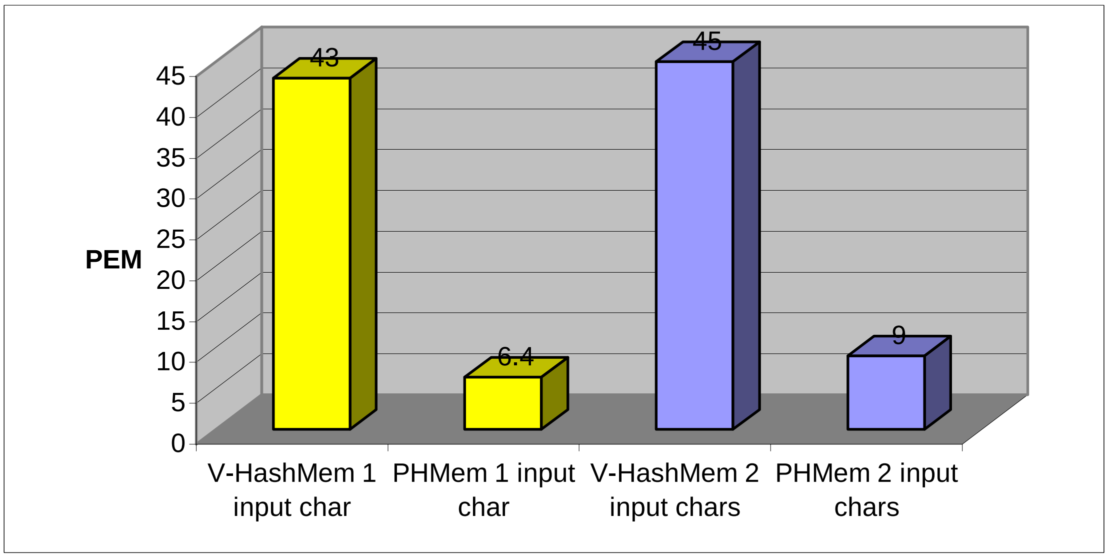

Figure 5.20: *Comparing V-HashMem with the PHmem over PEM.*

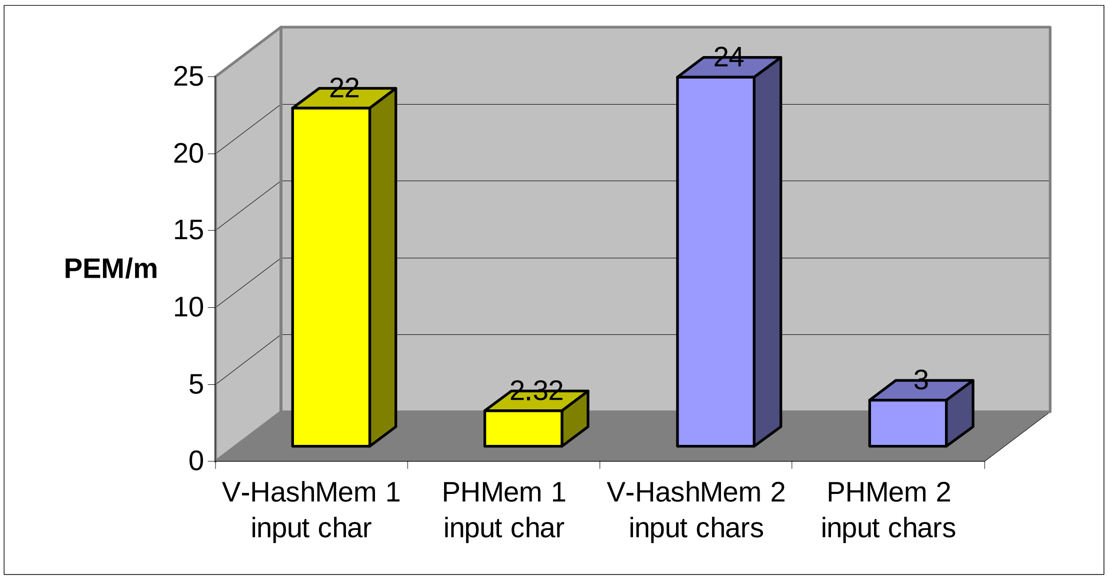

Figure 5.21: *Comparing V-HashMem with the PHmem over PEM/m.*

Until now, we compared V-HashMem Architecture with similar designs. Other approaches are the DCAM which was proposed by Sourdis *et al.*[5], DCAM by Baker and Prassana [12, 13] and Non Deterministic Finite Automata (NFA) by Clark *et al*. [9]. The results of these approaches are shown in Table 5.10. DCAM [5] by Sourdis *et al* achieves better Frequency and Throughput but worse Area cost per character and PEM than V-HashMem. Generally, it is considered as the best FPGA-based intrusion detection architecture but it supports 86% less patterns' characters than V-HashMem Architecture. On the other hand, DCAM, which was proposed by Baker and Prassana, has two approaches: Unary-based and Tree-based. Both approaches of this design not only achieve less Frequency and Throughput than V-HashMem Architecture but they also use much more area cost per character. Finally, NFA achieve worse Frequency than V-HashMem Architecture but it achieves a very large Throughput (about 7 Gbps) since the input bits are 32. This approach, also, has very much area cost per character and it supports much less patterns' characters than V-HashMem Architecture. Finally, these approaches do not use any memory block.

In conclusion, V-HashMem Architecture achieves the best PEM comparing with every related work we mentioned before. Also, it achieves the same or better Frequency and Throughput comparing with the most designs. Finally, V-HashMem Architecture used the less Area than any other design even though the stored patterns' characters are in most cases 70% more than the ones of the other approaches and this is the strong advantage of V-HashMem Architecture.

Detailed Comparison between the Designs over Area, Memory and Performance Parameters

Design	Input bits	Patterns' Chars	LCs	LCs/char	Mem Blocks (kbits)	Frequency (MHz)	Throughput (Gbps)	PEM	PEM/m	Device
V-HashMem	8	33613	2084	0.062	648	320	2.56	42.66	22.45	VirtexIIpro 20
	16		3918	0.11		~309	~4.95	44.98	23.67	
	8		2084	0.062		190.387	1.52	21.98	11.4	Spartan3 3000
	16		3918	0.11		~176	~2.82	24.68	12.8	
	8		2084	0.062		346.021	2.77	44.64	23.49	Virtex4 35
	16		3918	0.11		~340	~5.44	49.45	26.02	
Best HashMem	8	18636	2759	0.15	558	339	2.71	18.3	6.11	VirtexIIpro 7
HashMem unoptimized	8		2632	0.14	1188	333	2.66	18.86	2.96	VirtexIIpro 20
HashMem unoptimized	16		5219	0.28	1188	322	5.15	18.4	2.89	
Bloom Filters	8	420000	36720	0.09	629	63	0.50	5.58	37.24	VirtexE 2000
RDL+ROM	8	20800	>8000	>0.38	162	238	1.9	<5	<6.42	Spartan3 1000
PH-mem	8	20911	9426	0.45	576	361	2.88	6.4	2.32	Virtex2 1000
PH-mem	16		13554	0.65	612	358	5.73	8.85	3.02	
DCAM by Sourdis et al	8	18036	17538	0.97	0	335	2.68	2.75	Undefined	Virtex2 3000
DCAM by Baker and Prassana (Unary-based)	8	19584	8056	0.41	0	185	1.49	3.62	Undefined	Virtex2pro 100
DCAM by Baker and Prassana (Tree-based)	8	19584	6340	0.32	0	237	1.90	5.86	Undefined	
NFA by Clark et al	32	17537	54890	3.13	0	219	7.00	2.24	Undefined	Virtex2 8000

Table 5.11: *Summary Table. In this Table we summarize the results for any evaluation metric for every design we mentioned in this chapter.*

Chapter 6

Conclusions and Future Work

In this work we extended the HashMem Architecture which is an FPGA-based Network Intrusion Detection System. NIDS monitor the incoming packets on the network and compare some specific packets' data with known threats which are stored in a database. In our work, we store the patterns of different but close lengths in a *single* memory structure achieving low area and memory cost. We use the reuse technique which was firstly proposed in HashMem Architecture, with which we break the long patterns into shorter ones and reuse the already existent memory structures. In order to report a long pattern matching we combine the partial matching signals and report the final match. For this work, we used the SNORT rule-set, which is one of the most famous NIDS.

To this end we have achieved savings in the logic cost, while retaining the memory use at levels comparable to that of HashMem despite the 70% larger rule-set. This is a clear indication of the scaling abilities of the overall HashMem approach. The efficiency of V-HashMem is especially evident in the logic area cost per search pattern character (~0.06 LCs/char) and in the expected PEM rating, which about doubled compared to HashMem, and is the highest as we saw in the previous chapter. In addition, another important discovery was that as the SNORT rule-set progresses, new and more difficult rules are included. In our case, the difficulty stems from the fact that very wide patterns increase the cost of glue logic considerably, since some of them, when broken into shorter ones, created tree-like structures. Furthermore, as V-HashMem Architecture exactly defines the intrusion pattern, we eliminate false positives from which many other related works suffer.

We evaluated our system and we proposed two ideas in order to improve it and make it more efficient. Firstly, we proposed the idea improve the throughput of the system but the impact was to almost double the area cost, using dual port memories (same number of memory blocks). Which of the two approaches is better depends on what kind of system we would like to have.

The second idea, which was proposed, was about the support for header matching information. We found that it is possible at the cost of three extra memory blocks and about 80 logic cells for the necessary logic. This is a very small cost compared to the increased accuracy of the pattern matching subsystem. Of course, a complete system would also add the cost of header classification into the header group identifiers, but it was not included in our measurements.

Other potential challenges for the future include regular expressions. Also it would be very interesting to implement some of the options we described in the paragraph 2.1 (distance, within, etc). Another interesting idea for future work in order to improve the efficiency of V-HashMem is to pipeline the memories since the critical path of V-HashMem Architecture is in the memories.

These improvements lead closer to an efficient FPGA implementation of a NIDS system. We believe that such a successful system will rely on memory to store the patterns, and that the HashMem architecture with the proposed variable length extensions is a very competitive approach.

References

[1] V. Dimopoulos, G. Papadopoulos, and D. Pnevmatikatos, “On the importance of header classification in hw/sw network intrusion detection systems,” in *Proceedings of the 10th Pan-Hellenic Conference on Informatics (PCI)*, November 11-13, 2005.

[2] G. Papadopoulos and D. Pnevmatikatos, “Hashing + memory = low cost, exact pattern matching,” in *Proceedings of the 15th International Conference on Field Programmable Logic and Applications*, 2005.

[3] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, “A reconfigurable perfect-hashing scheme for packet inspection,” in *Proceedings of the 15th International Conference on Field Programmable Logic and Applications*, 2005.

[4] M. Attig, S. Dharmapurikar, and J. Lockwood, “Implementation results of bloom filters for string matching,” in *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[5] I. Sourdis and D. Pnevmatikatos, “Pre-decoded CAMs for efficient and high-speed nids pattern matching,” in *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[6] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, “Granidt: Towards gigabit rate network intrusion detection technology,” in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, 2002.

[7] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, “Implementation of a content-scanning module for an internet firewall,” in *Proceedings of the 11th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[8] S. Dharmapurikar, P. Krishnamurthy, T. Spoull, and J. Lockwood, “Deep Packet Inspection using Bloom Filters,” in *Hot Interconnects*, August 2003, stanford, CA.

[9] C. R. Clark and D. E. Schimmel, “Scalable parallel pattern matching on high-speed networks,” in *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[10] Y. H. Cho and W. H. Mangione-Smith, “Deep packet filter with dedicated logic and read only memories,” in *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[11] ——, “Programmable hardware for deep packet filtering on a large signature set,” in *First Watson Conference on Interaction between Architecture, Circuits, and Compilers (P=ac2)*, 2004.

[12] Z. K. Baker and V. K. Prasanna, “Automatic synthesis of efficient intrusion detection systems on FPGAs,” in *Proceedings of the 14th International Conference on Field Programmable Logic and Applications*, August 2004.

[13] ——, “Time and area efficient reconfigurable pattern matching on FPGAs,” in *Proceedings of FPGA ’04*, 2004.

[14] H. Song and J. Lockwood, “Multi-pattern signature matching for hardware network intrusion detection systems,” in *Proceedings of IEEE Globecom 2005*, November 28 - December2, 2005.

[15] R.W. Floyd and J.D. Ullman. The compilation of regular expressions into integrated circuits. In *Journal of ACM, vol. 29, no. 3*, pages 603–622, July 1982.

[16] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[17] R. Franklin, D. Carver, and B. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[18] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns. In *Proceedings of 13th International conference on Field Programmable Logic and Applications*, September 2003.

[19] Young H. Cho, Shiva Navab, and William Mangione-Smith. Specialized hardware for deep network packet filtering. In *Proceedings of 12th International Conference on Field Programmable Logic and Applications*, 2002.

[20] SNORT official web site: http://www.snort.org

[21] EASICS tools web site: www.easics.com

[22] Anonymous. Maximum Security, 3rd Edition.