TECHNICAL UNIVERSITY OF CRETE

ELECTRONIC AND COMPUTER ENGINEERING DEPARTMENT

MASTER THESIS

ON THE WAY OF HASHING FOR LOW COST EXACT PATTERN MATCHING

by

PAPADOPOULOS GIORGOS

CHANIA
JULY 2005

Acknowledgments

Abstract

Network Intrusion Detection Systems monitor all network traffic passing on the local sensor segment, reacting to any anomaly or signature based activity. As networks become faster there is an emerging need for security analysis techniques that can keep up with the increased network throughput. Clasic networkbased intrusion detection sensors can barely keep up with bandwidths of a few hundred Mbps. Therefore, next generation "sensors" should provide deep packet inspection capabilities in order to provide protection at network speed. Since such systems rely on pattern matching techniques to analyze packets, hardware based systems with dedicated pattern matching logic has become very promising.

Currently there are plenty published architectures that implement network intrusion detection. Eventhough, some of them achieve very good performance and efficiency, there are many limits arise from the fact that NIDS perform mostly signature analysis and the number of the signatures continues to increase with great speed. A new approach to intrusion detection is needed to solve the problems of larger and faster networks and mostly the constraints caused by the increase of different kind of attacks, that recently published research suffer from.

In this thesis we propose the combination of hashing and use of memory to achieve low cost, exact matching of SNORT-like intrusion signatures exploiting the benefits of FPGAs. The basic idea is to use hashing to generate a distinct address for each candidate pattern, which is stored in memory. Our implementation, HashMem, uses simple CRC-style polynomials implemented with XOR gates, to achieve low cost hashing of the input patterns. We reduce the sparseness of the memory using an indirection memory that allows a compact storing of the search patterns and use a simple comparator to verify the match. Our implementation uses in the order of 0.15 Logic Cells per search pattern character, and a few tens of memory blocks, fitting comfortably in small or medium FPGA devices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Intrusion detection is needed in today's computing environment because it is impossible to keep pace with the current and potential threats and vulnerabilities in our computing systems. The environment is constantly evolving and changing fueled by new technology and the Internet. To make matters worse, threats and vulnerabilities in this environment are also constantly evolving. Today, firewalls with limited capabilities are built into many commercial network switch and router. Simple firewalls usually examine packet headers for specific information to determine whether to block or allow packet passage. But protection from firewalls as we know them is not enough. As computer networks grow faster, it becomes easier for malicious packets to enter into a network without being detected. The purpose of a Network Intrusion Detection System (NIDS) is to help protect computer network users from malicious attacks. They perform deep packet inspection in order to identify a matching signature. Three primary types of signatures are string signatures, port signatures, and header condition signatures. A signature based IDS will monitor packets on the network and compare them against a database of signatures or attributes from known malicious threats. The currently common known such database is provided by Snort [30] via Snort Rules, on which are based most of the NIDS projects. There are many systems introduced to provide intrusion protection initially in software, or even implementations in general purpose processors. However, both these solution suffer from the growth of the network speed, for which these systems are a bottleneck. What is more, the most computational intensive part of an NIDS is the patterns matching. For this reason, separate hardware based solutions, both in ASIC and FPGAs, have been also developed that perform signature matching and support high throughput. However, FPGAs seem to be more suitable because of their capability to be reconfigured in order to be updated to the current signatures.

1.1 Motivation

The area of hardware NIDS pattern matching has been very active recently. Several architectures have been proposed to implement SNORT-like pattern matching in FPGA. The architectures differ in the approach (finite automata or CAM-like), internal organizations, and of course in their cost- performance tradeoffs [29; 20; 21; 26; 16; 14; 33; 11; 4; 3; 6]. The common factor of these efforts however is the continuous drive for lower cost, at the same or better performance.

Many researchers designed pattern matching architectures based on regular expressions (NFAs / DFAs) [13; 20; 26; 29]. This is a low cost solution, but does not achieve very high performance. It is also difficult to perform more than one character per cycle, and usually, the operating frequency is limited by the amount of combinational logic for state transitions.

Other researchers preferred to implement discrete comparators [4; 5; 12; 32]. This choice leads to designs that operate at higher frequency, using one or more comparators for every matching pattern. This approach uses FPGA logic cells to store each pattern. In particularly, every LUT can store a half- byte of a pattern and the flip-flops that already exist can be used to create a fine-grained pipeline without any area overhead. The result is high speed designs but with an increased area cost. Finally, while the matching rules are very often updated and their number increases quickly, the area will easily become a serious constrain.

Another straightforward approach for pattern matching is the use of regular CAM [8; 21]. Current FPGAs offer the capability to use the integrated block RAMs for constructing CAMs. This is a simple procedure, as the CAM is fed with the input data and reports a match according to its contents. These approaches achieve modest performance but they have increased area cost as well. Another major drawback is the large number of the patterns and also their variable width. Thus, there must exist separate CAMs for patterns of different widths, or construct a CAM that may contain variable width patterns. The first approach requires too many CAMs while the second one is a hard task. But the major drawback of the

CAM approach generally is the width of the CAM in order to also perform long pattern matching.

1.2 Scope

It is clear that the use of CAMs do not offer desirable results, since the logic cost remain high and there is significant a limitation in implementing long patterns. The common factor off all these efforts however is the continuous drive for lower cost, at the same or better performance. This work builds on two distinct ideas: (i) of the use of hashing of the input to retrieve approximate match information used in the Bloom filters [16; 3], and (ii) on the use of memories to provide exact match with fewer gates used by Burkowski [9], and later by Cho and Magnione-Smith [11].

Dharmapurikar et al. proposed the use of Bloom filters for low cost pattern matching [16]. Bloom filters are very elegant in representing set membership, but have two potential drawbacks: (i) they require multiple hash functions and memories, and (ii) they give an approximate match answer since they allow false positives. Attig et al. proposed the use of external SDRAM memory to augment the bloom filters with exact match, but at extra cost, and without offering worst case guaranteed bandwidth since the external memory is not pipelined [3].

Cho and Magnione-Smith used a CAM to match short patterns and to match unique prefixes of longer search patterns [11]. 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 since they require even larger CAMs.

Our proposed architecture, HashMem (Hashing + Memory), attempts to strike a different balance between memory and logic usage. We try to maintain high performance using fine-grain pipelining and also minimize the logic cost by using simple logic functions and taking advantage of the FPGA embeded memory blocks. We use simple hash functions to generate sparse but distinct addresses for each of the search patterns, giving us a hint of whether there is a possible match (with probability proportional to the density of the hash space). We then use an indirection table to “gather” the search patterns in a compact memory, and compare the input against the single possible search pattern to eliminate false positives. Our implementations achieve processing throughputs between 1.95-2.7 Gbps processing a single input character per cycle, for a cost of 0.15 logic cells per search pattern character and a few tens of memory blocks, fitting comfortably in small or medium FPGA devices.

1.3 Outline

The rest of the dissertation is organized as follows. In the next chapter we provide short information about NIDS and describe other NIDS and previous FPGA-based pattern matching architectures. In chapters 3 and 4 we describe the proposed Hash-Mem architecture starting from an initial approach and extend it to achieve better cost and performance. In section 5 we present the implementation results and we attempt a fair comparison with other published results. Finally, we conclude in chapter 6, and discuss some issues about future extensions.

At the end, there is an Appendix which contains some useful information about CRC theory, and also some information about the software assistance in order to build the VHDL cose.

Chapter 2

Network Intrusion Detection Systems

Intrusion detection is a security technology that attempts to identify and isolate “intrusions” against computer systems. Most ID systems identify such attacks using a technique called “signature analysis” (also called “misuse detection”). Signature analysis simply refers to the fact that the ID system is programmed to interpret a certain series of packets, or a certain piece of data contained in those packets, as an attack.

Most signature analysis systems are based off of simple pattern matching algorithms. In most cases, the IDS simply looks for a substring within a stream of data carried by network packets. When it finds this substring (for example, the “phf” in “GET /cgi-bin/phf?”), it identifies those network packets as vehicles of an attack. Snort rules are the most commonly used rules that describe such information about packets.

In the next sections we present some things of the Snort Rules, and then we present some software and hardware approaches that implement intrusion detection based on such rules.

2.1 Snort Rules

One of the best network intrusion-detection systems (NIDS) is the free and open source Snort package. It is an open source network intrusion prevention system, 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, CGI attacks, SMB probes, OS fingerprinting attempts, and much more.

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.

Eash Snort rule can contain header and content fields. The header part containd information about protocol, source and destination IP addresses and port. The content part contains substrings that may exist in packets' payload.

In out work we implemented pattern matcing only for the payload data of packets using the content field of Snort rules. For our implementation we chose to use the Snort ruleset dated early 2004 that consists of 1,474 rules and a total of 18,636 characters. At the end we also used the last published ruleset, dated May of 2005, in order to evaluate our design in upgrading.

In figure 2.1 we present the pattern distribution per width for both of the two rulesets, while in figure 2.2 we perform a comparison of the character distribution per width. We can notice the increase of characters in almost every width group and especially in long pattern groups. What is more, we can see the great increate of the number of characters mostly in long patterns.

2.2 Software Based NIDSs

There have been several software pattern matching algorithms based on Snort opensource rules. Snort NIDSs at its firts versions used bruteforce pattern matching resulting in very slow detection with low throughput. As network speed became

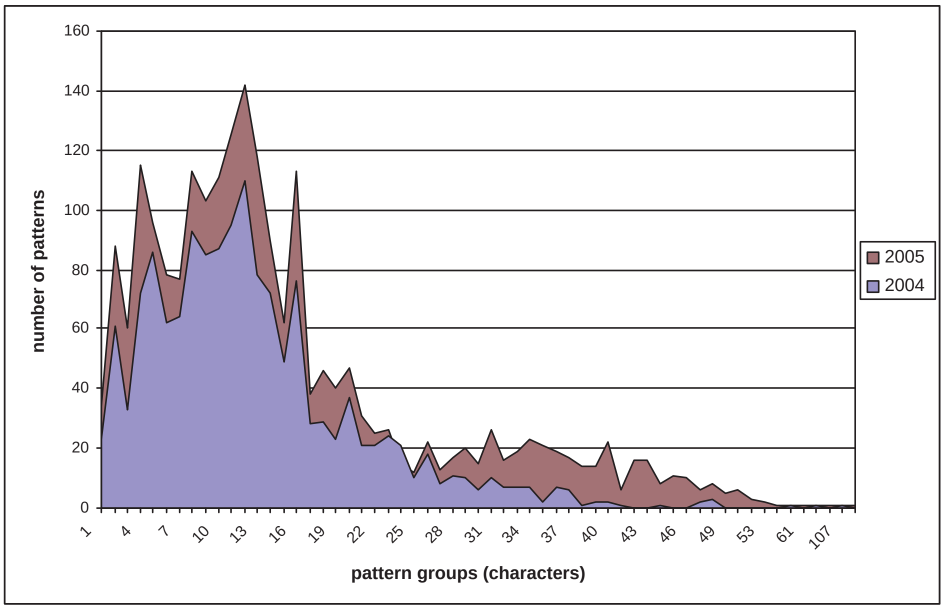

Figure 2.1: The distribution of patterns per pattern width. Comparison of the two rulesets.

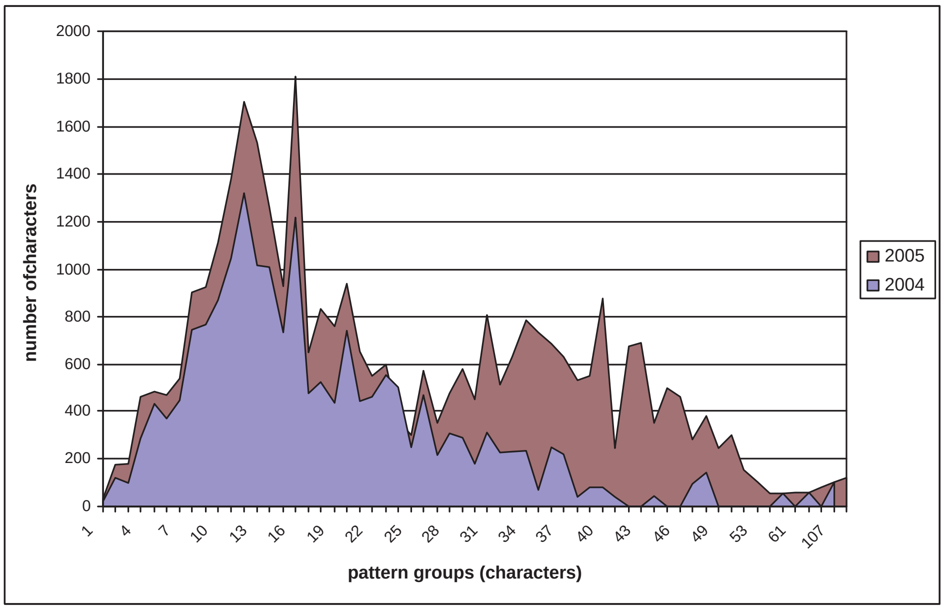

Figure 2.2: The distribution of total characters per pattern width. Comparison of the two rulesets.

greater and attacks were increased, more efficient algorithms that would improve performance were necessary.

A first improvement came from the Boyer-Moore algorithm [7], and later the “2- dimensional linked list with recursive node walking”. This improved the performance at 200% - 500% [30]. The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right.

A variant of the Boyer-Moore algorithm came from Coit et al. [15], who implemented a “Boyer- Moore Approach to Exact Set Matching” as described by Gusfield [22], which uses an Aho- Corassic- like tree [1] to store patterns. Another adaptation of Boyer-Moore was introduced by Fisk et al. [18] which is shown to be faster for matching less than 100 patterns. Moreover, the MWM algorithm (Wu- Mander multi-pattern) [35] forms another implementation of Snort [31]. This implementation is the faster of all the above.

Finally, Markatos et al. [2] proposed $E^2 \times B$ algorithm which provides quick negativeswhen the search pattern does not exist in data. It appears to be faster thanthe Fisk et al. implemenetation and for large incoming packets and less than 1-2k rules it is better than MWM too.

For a few handred Mbit/s, these software approaches provide sufficient power to capture and process the data packets. However, for higher-speed links (gigabit and higher), hardware accelerators have become necessary in order to process packets in real-time (or near real-time).

2.3 Hardware Based NIDSs

Hardware bases NIDS is an alternative solution to overcome bandwidth limitations of software approaches. There have been many hardware approaches developed both in ASIC and FPGA systems.

ASIC systems usually store their rules in large memory blocks and examine incoming data packets in integrated design engines. Such systems are generally expensive and altough they achieve much better throughput compared to software systems, they do not impress with their performance. Moreover, the updating to support new rulesets is achieved only by reloading memories with new data with no cpability of upgrading the search engine in case of changes in the kind of rules because of the complexity of attacks.

On the other hand, FPGAs seem more suitable as they are reconfigurable and also provide hardware speed. Thus, the updating process can be done even by entirely change the search engines with new design, with the only constrain to leave the interface intact. However, this is the hardest updating procedure. If only a small part of the design has been changed or added to the existing structure, it might be possible to perform incremental mapping and P&R, or even to partially reconfigure the device.

Most FPGA-based architectures are based on regular expressions (NFAs and DFAs) in order to preform pattern matching [13; 20; 26; 29]. Other researchers preferred to implement discrete comparators [4; 5; 12; 32]. Another straightforward approach for pattern matching is the use of regular CAM [8; 21] using the FPGA integrated block RAMs. Moreover, some designers implemented CAM-like architectures using dedicated logic [33]. Finally, another architectural way is to perform a kind of hashing on the input patterns in order limitize the logic usage [27; 34]. Next, we will present a brief description of these commonly used hardware approaches.

2.3.1 Non Deterministic / Deterministic Finite Automata

The most common FPGA approach is the regular expressions matching, implemented using Finite Automata (NFAs or DFAs) [13; 20; 26; 29]. Regular expressions are an extremely powerful tool for manipulating text and data. Such designs have low cost but at a modest throughput. The basic idea of is to generate regular expressions for every pattern or group of patterns, and implement them with N/DFA.

A deterministic finite automaton (DFA) is a finite state machine where for each pair of state and input symbol there is a deterministic next state. Such a machine has a collection of states. At each point in time it determines which state to visit next based only on its current state and the next input character. The primary difference is that a NFA can potentially encounter a situation where it can make multiple choices. For example, if it's in state 2 and sees an a, it might be able to proceed to either state 3 or state 4. It even has the ability to proceed from one state to another without consuming any input the edges it follows in doing so are called epsilon edges, because they are usually labelled with the Greek letter ϵ. Again, it may have a choice between following such an edge or following an edge that does consume input. In other words, in deterministic automata, for each state there is exactly one transition for each possible input, while in non-deterministic automata, there can be none or more than one transition from a given state for a given possible input. Thus, NFAs seems to be easier to design but DFAs are easier to implement as there are no choices to be considered.

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 packets[26]. Finally, finite automata are usually restricted in their operating frequency by the amount of combinational logic for state transitions.

The first hardware implementation was introduced by Floyd and Ullman in 1982, implemented in PLA [23; 19]. Much later, Sidhu and Prassanna introduced regular expressions and Nondeterministic Finite Automata (NFAs) for finding matches to a given regular expression [29]. Their work was primarily concerned with minimizing the time and space required to construct NFAs. For a single regular expression, the constructed NFAs and FPGA circuit could achieve 57.5-93.5 MHz in a Virtex XCV100 FPGA device. Then, Franklin, Carver and Hutchings [20] expanding on Sidhu et al. work, used regular expressions, with more complex syntax and meta- characters in order to cover the large set of expressions found in a Snort database. They managed to include up to 16,000 characters requiring 2.5-3.4 logic cells per matching character. They achieved about 30- 100 MHz on a Virtex XCV1000 and 50-127 MHz on a Virtex XCV2000E, and in the order of 63.5 MHz and 86 MHz respectively on XCV1000 and XCV2000E for a few tens of rules.

Later, Moscola, Lockwood et al. used the Field Programmable Port Extender (FPX) platform, to perform string matching for an Internet firewall [26]. They used regular expressions (DFAs) to store the patterns. They used JLex, a lexical analyzer generator for Java in order to convert regular expressions into DFAs. Initially, this implementation could operate at only 37 MHz on a Virtex XCV2000E and serve 296 Mbps. Finally, they described a way to increase bandwidth from 8 to

32 bits. This approach is capable of operating at speeds of 1.184 Gigabits/second for twenty-one ∼20-character regular expressions, and exceeding speeds of 2.5 Gigabits/second for smaller numbers of similar regular expressions. Lockwood also implemented a sample application on FPX constructing a small FSM [24]. Lockwoods FSM had also 32 bits bandwidth. Because of its large input width, this approach is practically unsuitable to implement for many patterns and even more for complicated DFAs. Their circuit operates at 119 MHz on a Virtex V1000E-7 device achieving 3.8 Gbps throughput.

Finally, Clark and Schimmel [13] developed a pattern matching coprocessor that supports the entire SNORT rule-set using NFAs. In order to reduce design area they used centralized decoders instead of character comparators for the NFA state transitions. Their design processes one character per cycle, can match over 1,500 patterns (17,537 characters)achieves frequency 100 MHz having total throughput 0.8 Gbps in a Virtex-1000 device. In FCCM 2004, Clark and Schimmel expanded on their earlier work implementing designs that process multiple incoming bytes per cycle. Their detailed results proved that NFAs and predecoding can produce low cost designs with higher performance, compared to DFAs and simple brute-force approaches.

2.3.2 CAMs and Discrete Comparators

The most straight-forward method to building pattern matching circuits is known as the brute-force approach. The brute-force algorithm produces circuits that perform a full comparison of every target pattern against the input in each clock cycle (discrete comparators). In other words, no match state is saved across cycles. The input text from a packet payload is broadcast to all of the pattern matchers and shifted past the target patterns at a rate of one character per clock cycle. A pattern matcher for a length m string usualy contains an m-character shift register for buffering input characters, m character comparators, and a match output that is true when all of the comparators signal a match between the contents of the input shift register and the target pattern. Once the shift register is full, the pattern matcher performs m parallel character comparisons per clock cycle.

The algorithm can be generalized to process i characters per clock cycle by instantiating i copies of each pattern and shifting the content of the buffer i characters in each cycle. To properly detect all possible positions of the pattern in the input stream, each copy of the pattern must start at a different offset relative to the beginning of the input buffer. The first copy starts at offset 0 and the ith copy starts at offset i-1. Cho et. al. [12] implemented a brute-force design that processed 4 characters (32 bits) per clock cycle. Sourdis et. al. [32] increased the throughput of Chos design significantly through aggressive pipelining, which also resulted in increased latency and lower character density.

Moreover, current FPGAs give designers the opportunity to use integrated block RAMs for constructing regular CAM. This is a simple procedure, that achieves modest performance, in most cases better than simple N/DFAs architectures. Gokhale, et al. [21] used CAM to implement Snort rules NIDS on a Virtex XCV1000E. They performed both header and payload matching on CAMs, however, increasing CAMs depth over 32 entries, resulted in unacceptable operating frequency due to routing. They achieved frequency of 68 MHz wiht 32 bits input data, resulting in 2.2 Gbps.

Cho et al. improved their earlier architecture introducing a ROM-based solution [10] to perform packet filtering. They shared partial data comparators and also dedicated decoders for character matching in combination to the partial matches. A further improvement was presented a little later achieving better results [11]. In their final design 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.

Finally, another CAM-based solution was introduced by Baker et al. [5]. They perform an entire ruleset processing so they can partition a ruleset in order to optimize the area and time requirements of the system. They apply graph theory techniques early in the circuit design problem, preprocessing the ruleset, in order to achieve high performance. However, this architecture allows false positives, and this is what they trade to rduce logic cost.

2.3.3 Hashing

Hashing may be a promising approach for implementing pattern matching hardware. The idea is based on the fact that using hashing on the input we may retrieve approximate match information. This of course allows false positives, since the hash function does produce the same value for other patterns too.

This idea was firstly used in the Bloom filters by Dharmapurikar et al. The proposed the use of Bloom filters for low cost pattern matching [16]. Bloom filters are very elegant in representing set membership, but have two potential drawbacks: (i) they require multiple hash functions and memories, and (ii) they give an approximate match answer since they allow false positives. Attig et al. proposed the use of external SDRAM memory to augment the bloom filters with exact match, but at extra cost, and without offering worst case guaranteed bandwidth since the external memory is not pipelined [3].

Another architecture that uses hashing to narrow the search character width is introduced by Sourdis et al. This approach is very close to ours as they use a hashing function and memory to retrieve final match information. However, in their architecture they try to extract perfect hashing function in order to perform search pattern encoding. Then they address an indirection memory that directs to the corresponding pattern in the pattern memory. In opposite to our design, they use a centralized, banked memory for efficient pattern storage, and they trade logic to reduce the memory size.

Chapter 3

The HashMem Architecture

In this chapter we descuss the initial idea on which the HashMem approach is based. Then, we describe the initial implementation presenting some issues that affect the efficiancy of the approach. Finally, we evaluate the implementation for ways to improve upon it.

3.1 The Basic Idea

The basic idea of HashMem is to use the input pattern to generate a unique candidate pattern address. Lets assume we want to match a set of patterns of length L characters. We feed L input bytes into a hashing module to generate the unique address of the candidate pattern. We then read the candidate pattern from the memory and compare it with the (delayed) input to verify the match.

This overall structure, shown in figure 3.1, has been used by Cho et al. [11] using prefix match logic, and an encoder to generate the unique address. Unlike this work, we use a CRC-type computation on the entire length of the match and avoid the use of encoders. CRCs have two advantages:

1. They are simple functions with small implementation cost,
2. They produce a “randomized” result, with a uniform distribution of all possible patterns of a specific width into each CRC value.

Depending on the polynomial used, each CRC function will produce a different mapping of patterns to locations. This gives us a way of adapting to different

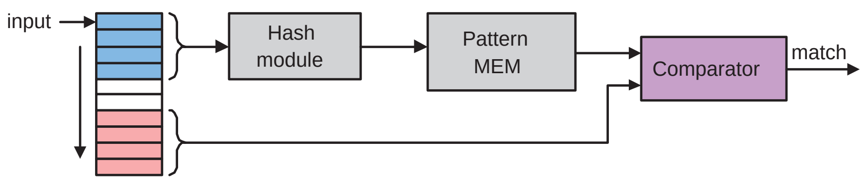

Figure 3.1: The basic HashMem idea. To search for a pattern of length N characters, we hash N input characters and produce a memory address. If the stored pattern matches the input pattern, we have a match, otherwise no.

pattern sets. Given a set of patterns, we can select a polynomial that produces distinct CRC values for every search pattern. Since a match can begin anywhere in the input stream, for our L character search, we have to check each of the L-character substrings starting at offsets 0, 1, ..., up to the end of the input packet. To achieve this, we pipeline the CRC generator.

Finding a polynomial that guarantees distinct addresses for each search pattern is easier when the density of the hash space is smaller. Experimentally we found that using 12 bits for Snort patterns, we can use simple polynomials that achieve this guarantee. However, a small memory density means that many memory locations are not used. Using 12-bit polynomials means that the address of the pattern memory is also 12 bits where the memory length is 4096. Supposing that the stored patterns are usualy less than 100, the utilazation of the memory is less than 2.5%. This problem is exacerbated for longer patterns since the memory is as wide as the search patterns and more blocks are required to set up the specific memory. To alleviate this overhead, we introduce a level of indirection, using 12 bits of CRC address, and packing the search patterns in a shorter memory to improve its utilization. The width of the indirection memory is related to the number of stored patterns, with 8 bits being more than enough for patterns of a given width. Thus, for the indirection memory, although the length is also 4096 and the utilization will be the same, the necessary memory blocks will be at most two since the width is at most 8 bits. Moreover, the pattern memory has now the minimum number of entries required to accomodate the search patterns, and as a result a better utilization. Figure 3.2 shows the process of using CRC generators as hashing functions and transforming the detection path with the use of the indirection memory level.

Until this point we discussed patterns of equal width (L). Dealing with multiple width patterns requires to (i) know the width of the possible matches, and (ii) the ability to read all these possible patterns. Since any given character of the input

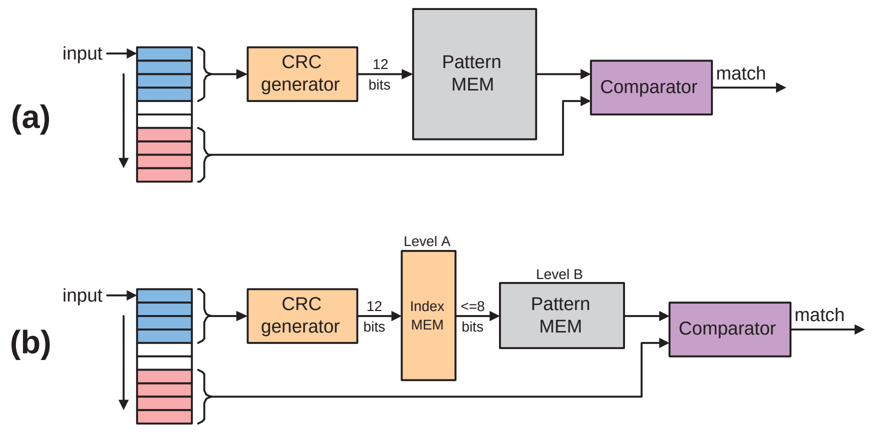

Figure 3.2: The process that turns out to the HashMEM architecture. **(a)** A CRC generator implements the hash module but the 12-bit output requires long pattern memory. **(b)** The insertion of the indirection memory level to alleviate the problem and shorten the pattern memory.

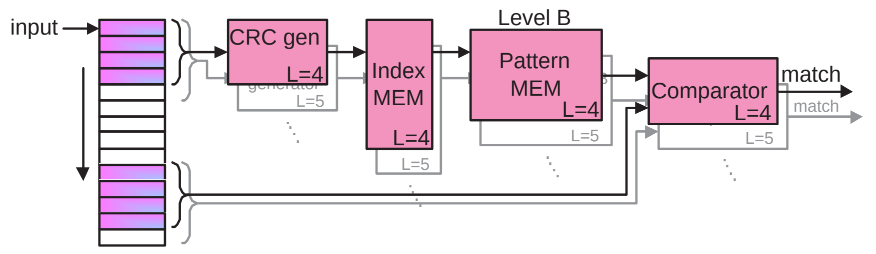

Figure 3.3: Detailed HashMem Architecture. The entire structure is replicated to detect multiple width patterns concurrently.

stream can be the last character of a pattern of arbitrary width, we use the simple approach of replicating the entire structure once for each of the different pattern widths. The resulting architecture is show in Figure 3.3.

3.2 Initial Approach

For the construction of a HashMem system, we first group the search patterns according to their width L. Then, for each group, we identify a 12-degree CRC poly-

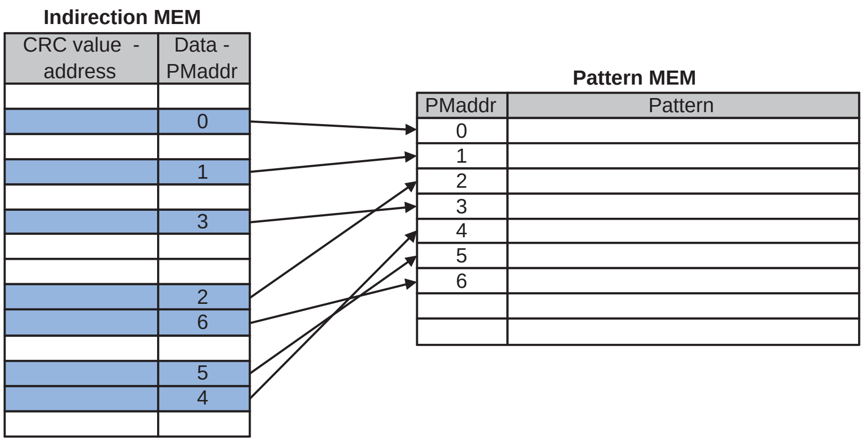

Figure 3.4: The indirection procedure. Data from the indirection memory is used to address the pattern memory

nomial that produces distinct CRC values for each pattern in the group. Using a software CRC generator based on this polynomial, we calculate the CRC values for all the patterns of each group. Then, the search patterns are packed in a pattern memory of width L without any restriction on their location. Finally, the indirection memory is initialized based on the calculated CRC values and the location of the corresponding patterns in the mattern memory, as shown in figure 3.4.

Initially the indirection memory is initialized to a special value “No- Match”. Then for each search pattern P that is stored in location PMAddr(P) in the pattern memory, we set location CRC(P) of the indirection memory to PMAddr(P). This essentially creates a pointer to the stored search pattern, if there is one.

The whole detection process is discrete for each width. Thus, there is one CRC generator for each width group, one memory for the indirection level, one for the pattern group storage and one comparator for the specific width for the final match.

3.3 Efficient CRC Generation

The HashMem architecture uses one CRC generator for each search pattern width, and each generator has to produce the CRC of L characters in one cycle. To achieve a CRC implementation according to these requirements, we first imple-

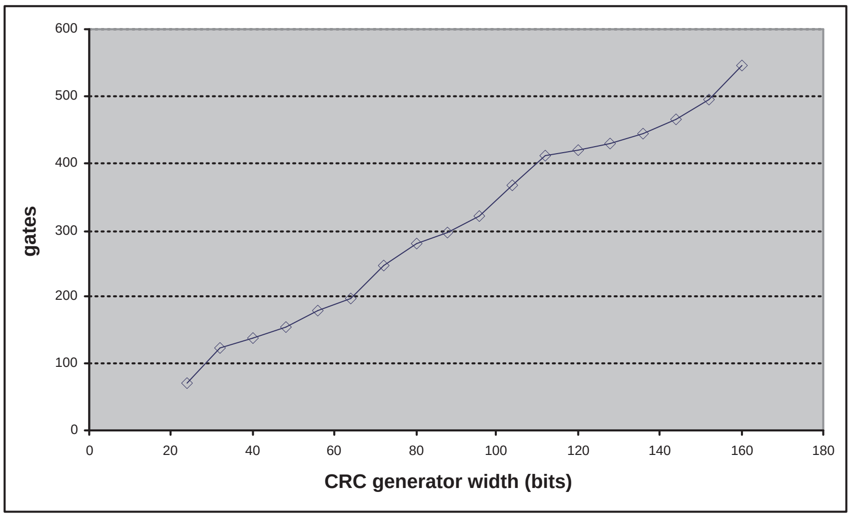

Figure 3.5: The area cost in gates for various input widths of CRC generators that use the polynomial [x^{12} + x^2 + 1].

mented a fully parallel, unpipelined generator, and then we pipelined it to achieve (i) throughput of one full hash per cycle, and (ii) good cycle time.

While each CRC generator is relatively simple, its cost is proportional to the input width. Some Snort search patterns exceed 50 characters or 400 input bits, for which even simple CRC generators use many gates. The diagram in figure 3.5 shows the increase of the area cost in gates of variable width generators of the same polynomial [x^{12} + x^2 + 1].

To reduce the cost of wide CRC generators, we produce the hash value for wide patterns reusing the narrower CRC values as partial hashes. We implement full CRC generators for the widths of 3, 4, 5 and 6 bytes. Then, to produce a CRC value for say 7 character wide pattern, we perform a CRC function on the CRC results for the first 3, and the next 4 input stream characters. The partial CRC values should be delayed appropriately so that they arrive at the same time to the second level CRC generator. Figure 3.6 illustrates this 7-character wide example.

This hierarchical CRC calculation and the reuse of partial CRC values results in significant area savings reducing the input of the generators to a more manageable width. Particularly, the combinational generators are 24-bits wide for patterns of widths from 7 to 12, while these widths are combined from two width groups, each of which produce a 12-bit CRC. Moreover, the rest groups from 13 to 32 are generated by the combination of four groups which results to a 48-bits wide input

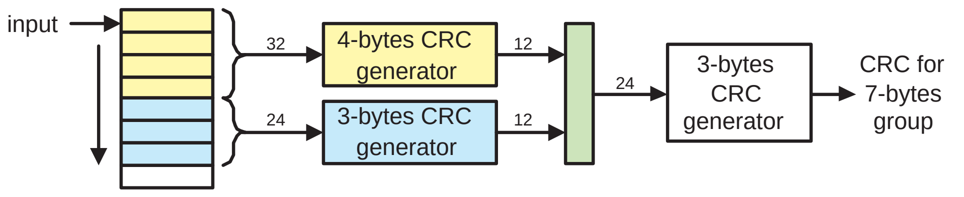

Figure 3.6: Hierarchical CRC generation for a 7 character pattern using the partial CRC results for the first 4 and the next 3 input characters.

for their generators. The reusing organization, which presents the group combinations for widths from 12 to 32 and also the width of the CRC generator required, is shown in table 3.1.

Another parameter affecting the cost of the CRC computation is the width of the CRC value. Smaller widths reduce the size of indirection memory, but make it harder to guarantee distinct addresses for each search pattern. We experimented with various CRC widths, and found that a larger space density, increases the cost (in number of LUTs) of the CRC generators. In practice, we found that the best compromise between CRC implementation cost and memory size is at 12 bits (i.e. using polynomials of degree 12). For small pattern widths, we can use smaller degrees (10 and 11), but in general we used 12 bits for every pattern width.

3.4 Handling Very Short Patterns

Very short patterns (1-2 characters) offer very few input bits, making the CRC calculation an overkill. Furthermore, the total pattern characters are very few, underutilizing the indirection and pattern memories. To address this inefficiency, it would be better to address the pattern memory directly with a minimum amount of logic. To achieve this we use a simple lookup table of 256 entries to match the single character patterns. We use the input byte to address directly the LUT, which is initialized with ’1’ in the addresses that represent the search patterns.

For the two character patterns, we notice that the total distinct patterns characters are less than 64, allowing an encoding with 6 bits. Based on this observation, we added a recoding function in the unused bits of the single character lookup table, recoding the 8 bits input into a 7 bit code. The most significant bit represents

group width	CRC input width
3	3
4	4
5	5
6	6

(a)

group width	partial groups	CRC input width
7	3, 4	3
8	4, 4	3
9	4, 5	3
10	5, 5	3
11	5, 6	3
12	6, 6	3

(b)

group width	partial groups	CRC input width
13	3, 3, 3, 4	6
14	3, 3, 4, 4	6
15	3, 4, 4, 4	6
16	4, 4, 4, 4	6
17	4, 4, 4, 5	6
18	4, 4, 5, 5	6
19	4, 5, 5, 5	6
20	5, 5, 5, 5	6
21	5, 5, 5, 6	6
22	5, 5, 6, 6	6
23	5, 6, 6, 6	6
24	6, 6, 6, 6	6
25	6, 6, 6, 7	6
26	6, 6, 7, 7	6
27	6, 7, 7, 7	6
28	7, 7, 7, 7	6
29	7, 7, 7, 8	6
30	7, 7, 8, 8	6
31	7, 8, 8, 8	6
32	8, 8, 8, 8	6

(c)

Table 3.1: The CRC generation organization. Table (a) shows the straightforward generation groups, table (b) contains the groups that use combination of two already generated CRCs and table (c) the groups that use combination of four CRCs.

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 matches. Each of these two lookup tables uses one memory block and a minimal ammount of logic for the pipeline delay. This procedure is shown in figure 3.7.

Actually, this method for matching dual-byte patterns is usuable for more than 64 patterns too. To be more specific, if these patterns amount at most to 128 they can be recoded in 7 bits each + 1 bit for the single byte match. This does not increase the memory block requirements in none of the two lookup tables, since the memory required for the two-bytes LUT fits in the shortest primitive memory block, which is 16kx1, and the widest block has 512 entries of 36 bits which is more than enough for the single character patterns LUT.

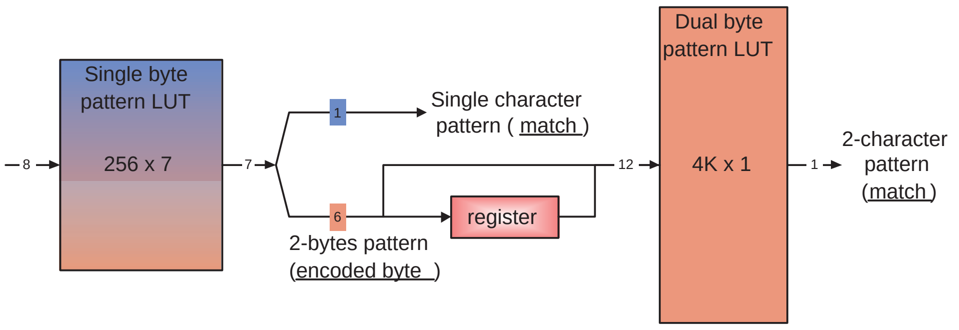
Single byte pattern LUT
256 x 7
8
7
1
Single character pattern (match)
6
register
2-bytes pattern (encoded byte)
12
Dual byte pattern LUT
4K x 1
1
2-character pattern (match)

Figure 3.7: Short patterns matching procedure using two LUT memories.

3.5 The Final Implementation - What's Wrong With It

In this first approach of the HashMem architecture we implemented a single-byte throughput design. Data are input in a 1-byte wide shift register once per cycle. Then, one CRC generator for each width begins the pipelined process by calculating one CRC value per cycle feeding the indirection memory. The indirection memory addresses the pattern memory and finally the requested pattern is compared to the corresponding delayed input string.

In this design we implemented the detection of patterns which width varies from 1 to 32 bytes. For the short patterns of 1 and 2 bytes wide we used described the look-up table approach. Then, we implemented the straightforward description shown in figure 3.2 (b) for pattern widths 3-6 and finally we used their CRC generator ouputs to calculate the CRC values for the rest groups.

This whole process, as already been discribed in details, forms a quite simple design to implement but also has two major drawbacks. One is the high area cost in memory blocks, which amounts to 181 for both the two memory levels. The greatest portion of the memory blocks is used in the second memory level, in the pattern memories. Since none of the groups is consisted of more than 128 strings, the primitive memory block we use is the 512 $\times$ 36 which is the widest and also the shortest one. This means that the pattern memory utilization is less that 25% and becomes even lower for wide string groups where the strings are even less. Moreover, wide strings require a large number of memory blocks as well.

Another major drawback of the initial implementation of the Hash+Mem architecture is the logic cost. Similarly to above, the greatest cost is particularly at

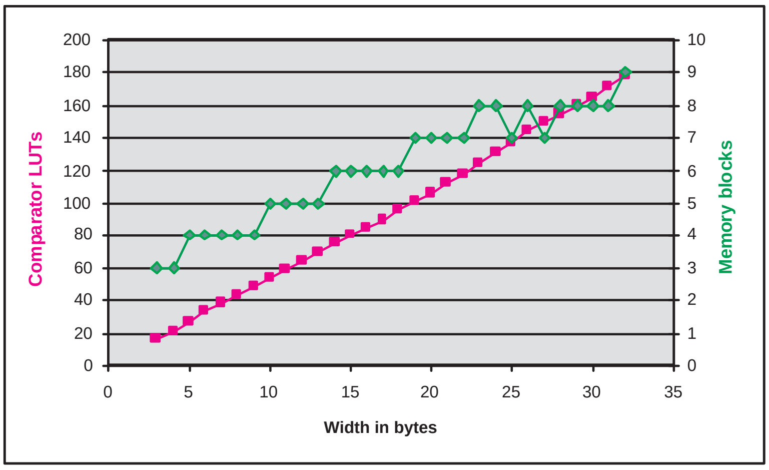

Figure 3.8: Area cost relatively to the pattern width. The memory block line is shown in green color and in pink is the distribution of the number of the comparator LUTs.

	implementations		
string widths (bytes)	1..12	1..24	1..32
implemented strings (total: 1,476)	873	1,352	1,432
number of bytes (total: 18,636)	6,526	14,633	16,879
logic cells	925	3,153	5,172
MEM blocks	42	120	181
Logic Cells / char	0.14	0.21	0.31

Table 3.2: The area characteristics of the initial design implementating for variable width groups.

the large comparators since the LUTs required for their implementation is about proportional to the width of the patterns in comparison. Figure 3.8 describes how the cost of the comparators and the number of the memory blocks are increased by the increase of the group width. Finally, it is important to notice that the distribution of the logic cost among the main logic modules used in this implementation is almost 85% for the comparators and only the rest 15% is utilized by the CRC generators.

In table 3.2 we demonstrate the area characteristics of this implementation including the memeory blocks and the logic cells required. There are also values of designs for less width groups implemented. It is clear how the cost increases proportionally to the width included in the design. We notice that the number of the logic cells per character is increases as more groups are added in the design. Moreover, the use of a large FPGA device is obligatory because of the memory needs, since the amount of the memory blocks is very large, eventhough the logic utilization covers only a small percentage of its logic cells.

Chapter 4

Improvements and further implementations

In this chapter, we address initial approach drawbacks' in order to improve upon the design. Then, we present some new HashMem variants each of which implemement an additional improvement.

4.1 Processing Two Characters Per Cycle

A first improvement to the initial implementation is achieved by processing two characters per cycle. The performance of a HashMem system can be doubled exploiting the fact that Xilinx memory blocks provide two read ports. Hence, we can double the processing throughput to 2 input characters per cycle, processing two patterns at offsets 0 and 1 per cycle. To achieve this, we need to implement a separate hash+mem path for each of the two processing patterns. Since the memories are dual ported we may use the existing memory assigning each port to each one of the two patterns.

Therefore, we need to provide two addresses into the memories per cycle. This means that the CRC generator logic has to be replicated, for offsets 0 and 1 in the input stream. The two CRC values, which are used as addresses, are fed to the two ports of the same memory (as they refer to the same pattern width). The indirection memory provides two pattern memory addresses, which are used at the

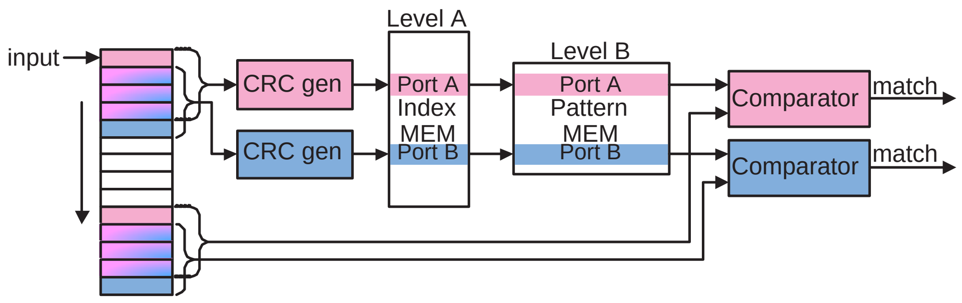

Figure 4.1: The 2-bytes per clock cycle overall structure for a given pattern width. The two memory ports are used for the patterns at offsets 0 and 1, while the logic is replicated.

two address ports of the same pattern memory as well. The two possible search patterns provided by the two output ports of the pattern memory must be matched against the input for offsets 0 and 1. As a result there are also required two copies of the comparators. The overall structure for a given pattern width is shown in figure 4.1.

One advantage of this technique is definitely that the throughput is doubled. What is more, the memory requirements remain the same since we take advantage of the dual ported memory blocks. On the other hand, the logic cost is almost doubled. The CRC generator and the comparator at the final stage, which are the main logic units of the design, are now duplicated.

4.2 Reusing Logic and Memory for Wide Patterns

Another improvement may be achieved aiming to a better utilization of the memory blocks and also by decreasing the comparators cost. As I have already mentioned very long strings are few but use very wide pattern memories. The widest Xilinx memory block has 512 entries of 36 bits [36; 38; 37]. The few wide patterns will leave almost the entire memory empty. Even in the indirection memory level which length depends on the CRC width the utilization is also very low.

In addition, most of the logic in the HashMem architecture comes from CRC generators, but mostly from the pattern comparators. Making a more CRC efficient generation, as discribed in section 3.3, we reduce the cost of wide CRC generators by making use of existing narrower CRC values as partial hashes. However, the higher cost in logic comes from the comparators and especially from the long ones, since their cost is proportional to their width.

Long patterns, due to their small number in the Snort rule set, offer us with this opportunity to reuse both the memory and the comparator structures instead of replicating them. The idea is that instead of matching a wide pattern in a separate structure, we split it in smaller pieces in order to fit in existing narrower structures. Thus, we match each of them in the existing structures, and use extra glue-logic to identify the long pattern parts and to combine the partial matches into the complete match.

Figure 4.2 depicts a short example where a 7 character pattern in matched as two sub- patterns of 3 and 4 characters. Note that we use this small pattern width only for reasons of clarity and brevity; we actually used this technique for widths 17 and up. The two sub-patterns are added into the existing structures for widths 3 and 4, reusing the CRC generators, memory, and comparators. First of all, we break the 7 character patterns into the 3 and 4 character parts. Mixing the 3-character parts with the original 3-character patterns we may look for a CRC polynomial for the new 3-character set. Then, we initialize the inderection and the pattern memory with the new inderection values and the new pattern set respectively. Finally, whenever a match takes place we have to identify whether is refered to a complete (original 3 or 4 character pattern) pattern match or to a partial match. We also have to be able to pair the correct parts for each long pattern. For that reason we use the indirection address in conjunction to the reported match in order to generate a recoded value. The match of these recoded values generates the composite group match. Still, there are two issues that we have to address in order to make this approach work.

First, in order to add a new pattern in an existing structure, the CRC of each pattern part should not conflict with any preexisting pattern in the original set. The existence of dublicates between parts or between a part and a preexisting pattern is not a problem because we just keep only one of each. However, the above restriction is addressed in two ways: first, the density of the hash space still remains small, so probabilistically our chances are good. Second, if we are indeed unlucky, we can always change the CRC polynomial and find one that removes the conflict. Should both of these options fail, we still have alternatives. We can partition the pattern in a different way (in our example perhaps in 4+3 or 2+5 characters instead

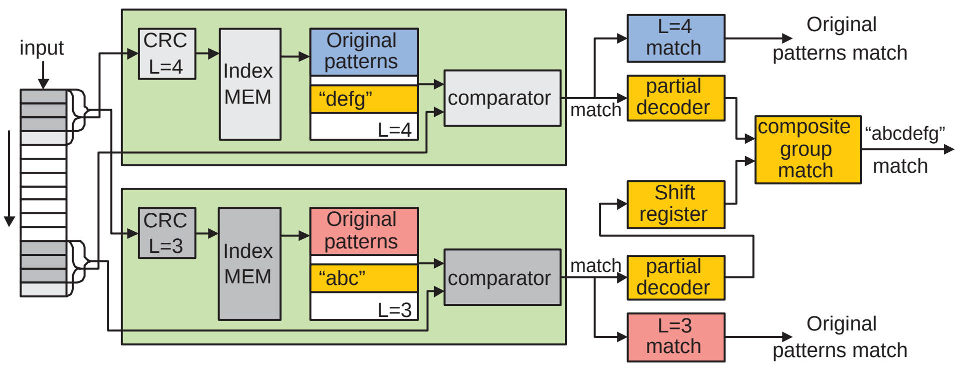

Figure 4.2: A 7 character pattern ("abcdefg") is partially matched as two patterns of 3 ("abc") and 4 ("defg") characters. The partial matches are then combined to determine the overall match.

of the 3+4 used in our example). The reuse of the resources can happen in many different ways increasing our chances of finding a convenient mapping. In our experience, the complicated options are not needed and it is straightforward to add the sub- patterns in existing structures.

The other issue is the glue logic that combines the partial matches into overall matches for the pattern. Since each of the two sub-patterns are detected at offset 0, we must delay the match information for the first sub-pattern to AND it with the corresponding data of the second sub- pattern and determine the overall match. In our example we must delay the first 3 characters match data by 4 clock cycles.

This implementation approach can be extended to further improvement applying the doubling of the throughput as described in the previous section. This means that even the glue logic, used to identify the long pattern parts and to combine the partial matches into the complete match, should be replicated.

4.3 Sharing Memory Structures between Different Pattern Widths

To further reduce the amount of memory needed, we can exploit even more two already noticed facts: (i) the low density of the indirection and data memories, and (ii) that the Xilinx memories are dual ported. The idea for sharing the data memory is simple: we partition the memory in two independent portions. The “upper” portion is used for patterns of width X and the “lower” portion is used for patterns of width Y (usually $X + 1$). Consider for a moment the data memory for say width 3. The minimum dimension of a single Xilinx memory block is 512x36 bits (enough for 4 characters), while the number of patterns of widths 3 and 4 is 33 and 72 patterns respectively. It is clear that both these sets of patterns can coexist in the same memory block, *without* any overhead. However, since we use different CRC generators for 3 and 4 characters, the addresses for each of the widths will be different. Here we can use the two read ports of the memory: we statically assign each read port to a given size, and arrange the patterns in two separate portions of the memory space.

While sharing the data memories is fairly straightforward, sharing the indirection memories is a bit more involved. Consider a simple example where there is only one string for each of the widths 3 and 4. Each of the indirection memories for widths 3 and 4 will have a single non-empty entry at the location determined by the CRC function of the corresponding width. These non-empty locations will point to the data memory locations that the actual strings will be placed, say X and Y. If the locations of the two non-empty entries are distinct, we can merge the two memories, creating a simple indirection memory, with *two* non-empty locations, with contents X and Y at their original locations. Moreover, since we devide the pattern memory in two portions, we may assign the prefix of the addresses of each port directly to a value that declares the memory portion used from each one of the two width sets. This allows as to store in the indirection memory only the relevant part of the pattern address as if they were stored in separate memories. Now we can use the two read ports for each of the pattern widths. Figure 4.3 outlines the sharing of both levels of memory in our architecture.

To show that this scheme does work, we consider two cases, of a pattern match and of a pattern mismatch. First, if a pattern match input appears for say size 3, the hash value will point to the location containing X. The data memory string will be read for width 3 from the specific port, and a match will be reported. For an arbitrary non-match input, the CRC value will most probably point to an empty location, resulting in a mismatch. However, there is the possibility that for some 3 character input, the CRC value will point to the location initialized to the 4 character pattern. To avoid any false matches, we augment the comparator with an additional bit that indicates the portion of the memory that the pattern belongs to. In this way, when we read location Y of the data memory that contains in a 4

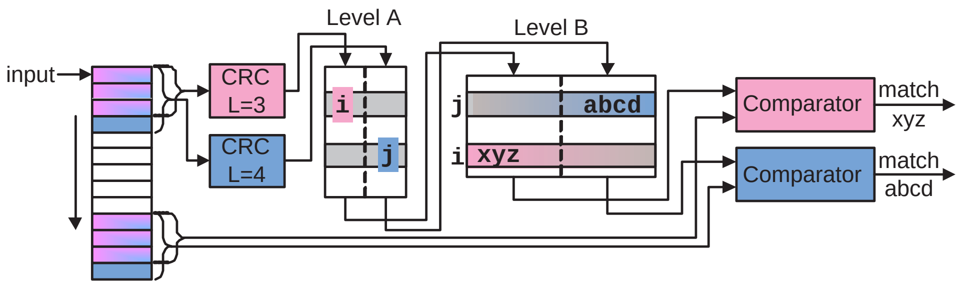

Figure 4.3: Sharing of indirection and data memories for patterns of different widths. While conceptually the two ports point to the two original memories, the data is merged and the dashed separating lines indicate the separate read ports and not distinct data

character pattern, it is impossible to report a 3 character match.

If there are same CRC values in patterns of the two sets then we may have a conflict in addressing the pattern memory. In that case we should change one or both of the CRC polynomials to avoid the conflict. Another course of action is to take advantage of the CRC conflict by allocating the conflicting patterns in matching relative addresses according to the portion of the memory that they belong. In practice, in all the cases we found that the two sets did *not* conflict, and we used a simple merging of the memories.

Unfortunately, this implementation approach cannot coexist with the design for doubling the throughput as described in section 4.1. This is absolutely logical because both improvement approaches exploit the dual port characteristic of the Xilinx memories.

4.4 Further Area and Speed Optimizations

4.4.1 Pipeline

In order to achieve better performance and decrease the area cost we used some techniques that are appicable to all our approaches. First of all, we pipelined the whole hashmem path using fine-grained pipeline. This pipelining approach is based on the observation that the minimum amount of logic in each pipeline stage can fit in a 4-input device LUT and its corresponding register. This is a primitive characteristic of the structure of Xilinx CLBs and not only, since many Altera devices have also similar structure. Thus, the resulting operating frequency of this pipelining method is limited only by the latency of one logic cell and the interconnection network. Moreover, there is no area cost overhead, since each logic cell contains both the parts of that minimum logic unit, the LUT and the flip-flop. The only drawback of this method is the big pipeline depth which results in higher latency. However, latency is not critical for our design. As a result, we pipelined this way all the logic units that our implementations include. Thus, we pipelined the CRC generators and the comparators at the end of the path, but also the necessary glue-logic required in some of our designs.

4.4.2 Xilinx SRL16

As already described, the same data that are used in the CRC generator are finally compared to the output data of the pattern memory to detect a match. This means that the data must be registered to achieve the required delay until the process reaches the comparator. Moreover, partitioning wide pattern into shorter ones also requires a delay, which is shown in figure 4.2, in order to determine the overall match. These registers are quite many and also wide which increases a lot the logic cost. The Xilinx SRL16 cell is a customizable up to 16 bits shift register. It is an alternative mode for the look-up tables and as a result it is implemented in single logic cell. Moreover, the output of the shift register may be stored in a flip-flop which improves the total performance and also increases the shifts to 17 [36; 25]. As, a result, we used these SRL16 LUTs in order to minimize the logic cost in the necessary delays whenever possible.

4.5 Upgrading

As we have already commented, an important advantage of our architecture is the ability of adapting to new patterns sets. The CRC generation process is very easy and fully automated due to software assistance and also the initialization both for the indirection and the pattern memories. Moreover, supposing that the utilization of the memory blocks is quite low, our chances in fitting new search patterns are pretty good. Thus, upgrading our designs with new rulesets is an easy task and in some cases with no extra memory cost, but only with limited increase in logic cost.

Chapter 5

Evaluation

We may measure the quality of an FPGA-based intrusion detection system using performance and area metrics. We measure the performance presenting the operating frequency and calculating the throughput of each implementation ($throughput = frequency \times inputbits$). Moreover, using the information about the logic cost we calculate other metrics in order to show the quality of our designs relatively to other parameters such as the number of intrusion patterns implemented and also the total number of their characters.

In our results, we use the number of Logic Cells, i.e. $ReportedSlices \times 2$. The area cost is measured in terms of logic cells needed to match a single character, produced by the ratio of the total logic cells over the total search characters implemented. It is also useful to find a metric that takes into account both performance and area cost. Thus, we used a $PerformanceEfficiencyMetric(PEM)$, as already used by other researchers [5; 6; 14] in order to evaluate designs efficiency. PEM is discribed by the ratio of performance over the area cost, in other words we calculate it dividing the throughput by the logic cells per character. Therefore, it rewards architectures that strike a balance between throughput and area cost. Moreover, since we attempt to strike a different balance between memory and logic usage, it would be also important to have a criterion to tell about the efficiency not only rewarding the designs with less area cost but with less memory usage as well. Therefore, introducing PEM/m as a new performance metric gives us the ability to measure the efficiency of our designs according to both of these parameters. PEM/m comes from the ratio of $[PEM/(MEM(kbits)/characters)]/100$.

We evaluate the HashMem architecture and our implementations using the official Snort rule set [28] dated early 2004 that consists of 1474 rules and a total of 18,636 characters. We implemented the HashMem architecture in VHDL using the Xilinx ISE 6.2 tools using Spartan 3, Virtex 2 and Virtex2Pro devices, all in the highest speed grades (-5, -6, and -7 respectively). We also used automated tools that given a set of patterns discovered the proper 12-bit polynomials, in order to generate the VHDL code for the CRC generators, and the corresponding mapping of patterns to memories.

At first, we will present our architecture results evaluating and comparing the various implementations as been described in previous sections. At the end we will attempt a comparison, in terms of performance and area, with previously published research.

5.1 The HashMem Architecture Variants

Since, in this first part, we want to demonstrate the evaluation results of the variants of our architecture, we decided to use the same device family for all our variants. This puts us in a better point of view in order to be able to make easier comparisons between these implementations. Thus, we used the synthesis results for Virtex2Pro family, choosing the smallest device possible for each implementation.

5.1.1 Performance and Area Evaluation

Table 5.1 compares the variants of the HashMem architecture for a Virtex2Pro device. The first is the basic architecture that uses a full structure for each of the search pattern widths. This variant is complete and scalable and is implemented in two variants which process one and two characters per cycle respectively. We should also take into consideration that in these two approaches we didn't implement the complete Snort ruleset but only the patterns sets of width from 1 to 32 bytes. However, it is also clear that both designs have a significant logic and an excessive memory cost (181 memory blocks). Still these designs fit in a medium Virtex2Pro device. Moreover, they achieve a quite high operating frequency, 292 and 285 MHz. We notice that it is almost same for both variants, which results

Design	Input bits	Frequency (MHz)	Throughput (Gbps)	Logic Cells	Pattern chars	LC / char	MEM blocks (kbits)	PEM	PEM/m
HashMem	8	292	2.336	5,172	16,879	0.31	181 (3,258)	7.5	0.39
	16	285	4.560	10,160		0.60		7.6	0.39
HashMem + Reuse	8	333	2.664	2,632	18,636	0.14	66 (1,188)	18.9	2.96
	16	322	5.152	5,219		0.28		18.4	2.89
HashMem + Reuse + Share	8	338	2.704	2,570		0.14	35 (630)	19.6	5.80
HashMem + Reuse + Share + Small CRCs	8	339	2.712	2,759		0.15	31 (558)	18.3	6.11

Table 5.1: Comparison of the HashMem architecture variants.

in almost double throughput for the two-characters per cycle variant, on order of 4.560 Gbps. Finally, the Performance Efficiency Metric (PEM, i.e. ratio of performance over the logic area cost per character) for both designs is around 7.5, while the new efficiency metric PEM/m that includes the memory cost per character is at 0.39.

Adding memory reuse (+Reuse) for wide patterns reduces the number of memory blocks to 66, and requires only ∼2,600 logic cells of logic for processing of a single character per cycle, or 0.14 logic cells per character.Considering that in these two variants we implement the complete Snort ruleset from 1 to 107 character width, both the areac cost and the memory usage show a great improvement against the initial approach. Moreover, at a frequency of ∼330MHz, the processing throughput exceeds 2.5 Gbps. Doubling the processing throughput doubles the logic cost, but without additional memory use, as expected. The Performance Efficiency Metric (PEM) for these two designs is around 18.9 and 18.4 respectively, which is ∼250% grater than above. Finally, the PEM/m value is much larger than the PEM value, achieving2.96 and 2.89 for the two designs of this variant which is ∼7.5 times greater than the previous design.

Adding memory sharing between patterns of different widths (+Share) offers a big improvement in the memory use, reducing the number of required blocks to 35. Again we included the complete Snort ruleset as in the +Reuse variants. We have to remind that in this design we cannot take advantage of the dual-ported memory blocks to double the throughput since the two read ports are used to prerform the memory sharing between patterns of different widths. Thus, we input a

Design	Input Bits	CRC generators	Comparators	SRLs	Glue Logic
HashMem + Reuse + Share + Small CRCs	8	32%	42%	13%	13%
HashMem + Reuse + Share	8	32%	42%	14%	12%
HashMem + Reuse	8	32%	42%	14%	12%
	16	33%	43%	14%	10%

Table 5.2: Area utilization ratios.

signle character per cycle. However, the logic cost remains about the same, due to the additional glue logic for the overall match determination. The smaller design size pushes the frequency up to about 340 MHz, slightly increasing the processing throughput to 2.704 Gbps and the PEM value to 19.6. Double than the previous design is also the PEM/m value which shows the improvement rate of sharing memories.

The last line of Table 5.1 uses smaller indirection tables for the smaller pattern widths (3-6). For these widths we can identify CRC polynomials of degree 10 and 11 that provide distinct addresses for all patterns. With fewer address bits the indirection memories use 1 instead of 2 memory blocks (2Kx9 instead of 4Kx9 memory), saving a total of 4 memory blocks. This is our best design, and achieves an operating frequency of $\sim$340MHz at a cost of 0.15 logic cells per character, and a PEM value of 18.3. The area cost of the design is divided as follows: CRC generators 32%, comparators 42%, Glue logic 13% and SRLs 13%. Finally, a slight improvement is also shown in the PEM/m value which supports our claim that this design is the most efficient.

In table 5.2 we present ratio of the area cost utilized by CRC generators, comparators, SRLs and glue logic for the three improved HashMem variants. Furthermore, an overall comparison in performance of the described HashMem variants is figured in the graph in figure 5.1, which plots the frequence and the throughput of each implementation. Moreover the diagram in figure 5.2 compares the area cost in terms of logic cell per character, and also the usage in memory blocks of the 6 variants. Finally, the last two diagrams in figures 5.3 and 5.4 perform a graphical comparison of the PEM and PEM/m values respectively.

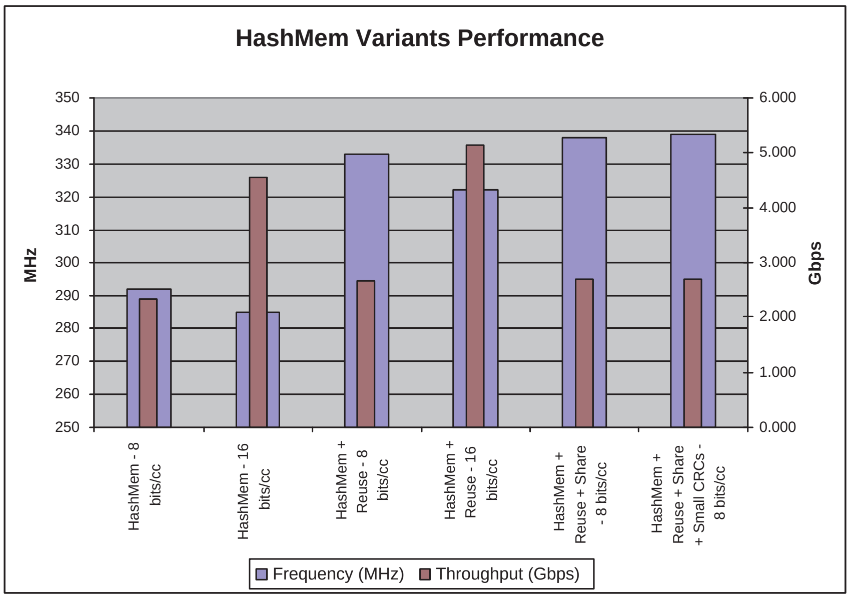

Figure 5.1: A plotting diagram that compares the preformance in terms of frequency and throughout.

5.1.2 Area Utilization Analysis

Table 5.3 presents the utilization of the implementations for both the logic cost and the memory usage for the corresponding device. As we can easily notice, in the initial implentation the large number of required memory blocks force us to use a device that has enough memory to accomodate our needs. However, this device has more than enough logic cell available for our design. Things seem to be better when we double the input bits per cycle, where the area utilization is doubled for the same device memory utilization for the same device. Still, the memory utilization of the occupied memory blocks is extremely low, under 5%, which proves the chances we had in reusing the memory blocks for partial pattern detection, described in next line.

In the HashMem + Reuse design the memory requirements have been decreased and so it fits in a smaller device which has at least 66 memory blocks to provide. Eventhough the logic cost has become almost half, the area utilization for the device used in this case remains almost the same as in the previous design. Finally, partitioning long patterns into shorter and packing them into the existing

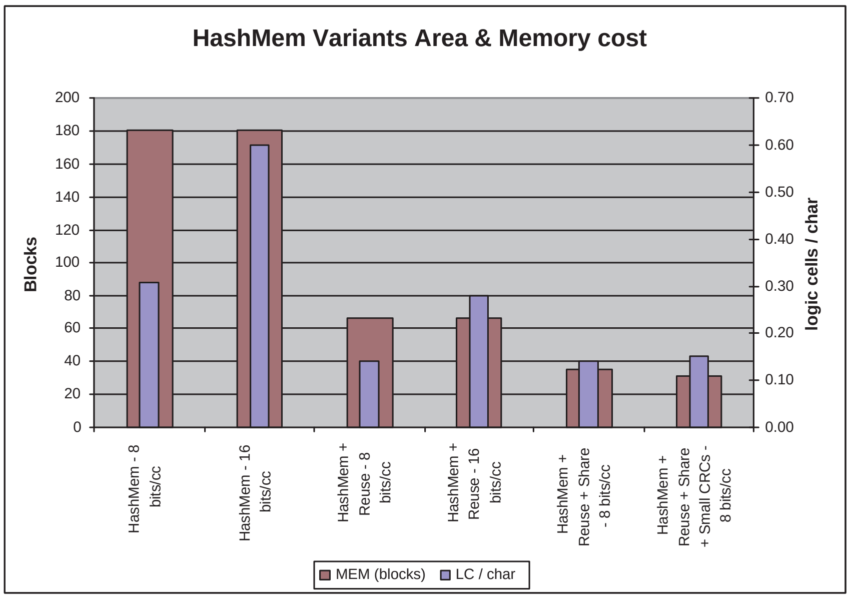

Figure 5.2: A diagram that compares the area cost and the memory block usage.

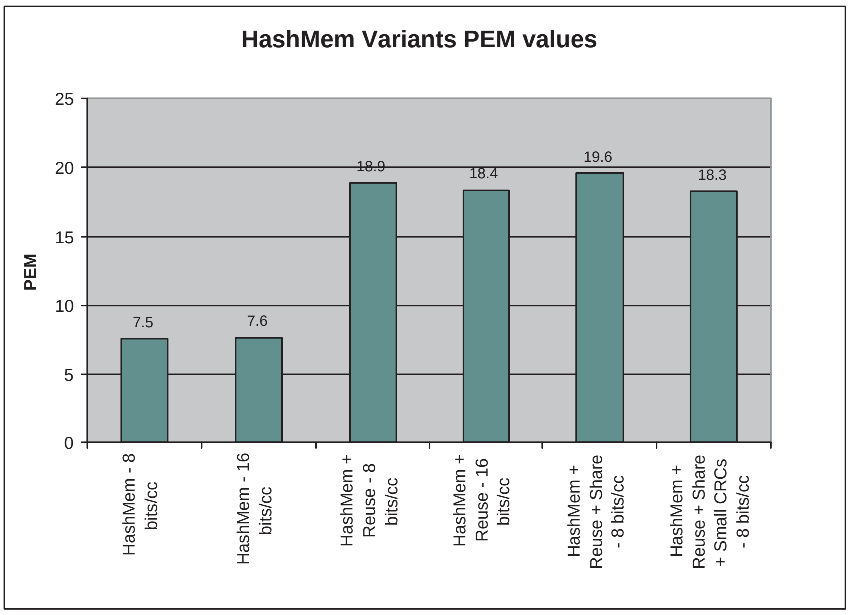

Figure 5.3: A diagram that compares the PEM values.

memory structures, not only reduced the memory blocks used, but also increased

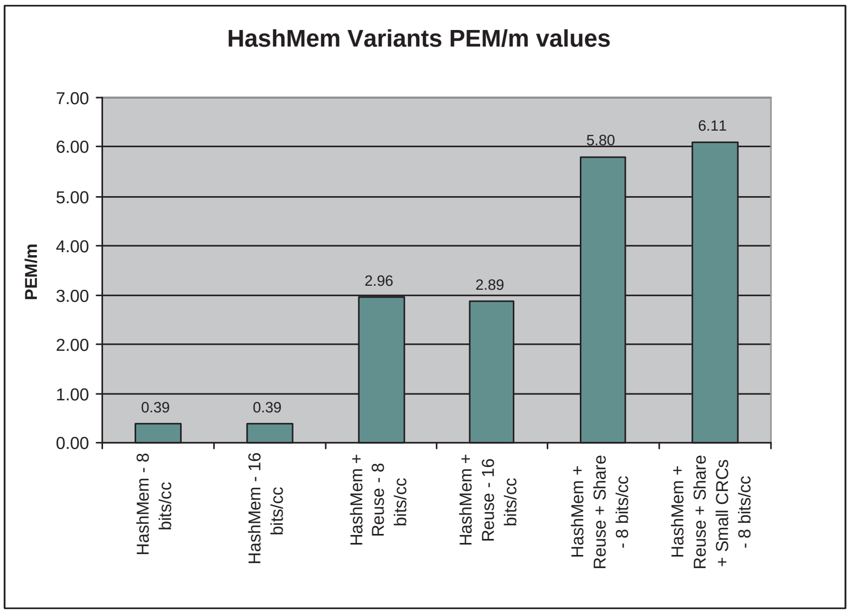

Figure 5.4: A diagram that compares the PEM/m values.

the utilization of the existing structures. Actually, it has rised by three times. Still, 85% of the used memory remains free to further improvements. And here comes the memory sharing option.

In the next line, where the +Share option is figured, we share memory blocks between two pattern widths using less memory blocks and as a result, extending the use of the remaining blocks. Thus, the used memory utilization has now reached 27.4%. The logic cost is almost the same with the previous implementation, that of throughput of 8 bits per cycle. But, taking into consideration that the total number of memory blocks is 35, this design fits in an even smaller device, keeping the area and memory utilization for this device at 26% and $\sim$80% respectively.

The last line of the table 5.3 presents the utilization of the final implementation adding the small CRCs feature. Fitting in the same device as above utilizes even less the available memory blocks of the device increasing the utilization of the memory used up to 30.9%. On the other hand, the area utilization remains at almost same rates.

Finally, a more efficient comparison is achieved graphically, using the two graph in figures 5.5 and 5.6. The first shows the area utilization and the other figures thw memory utilization in two aspects as described in table 5.3.

Design	Input bits	Device	Area Utilization	Device Memory Utilization	Design Memory Utilization
HashMem	8	Virtex2Pro 40	13%	94.30%	4.50%
	16		26%	94.30%	4.50%
HashMem + Reuse	8	Virtex2Pro 20	13%	75.00%	14.10%
	16		27%	75.00%	14.10%
HashMem + Reuse + Share	8	Virtex2Pro 7	26%	79.50%	27.40%
HashMem + Reuse + Share + Small CRCs	8	Virtex2Pro 7	28%	70.50%	30.90%

Table 5.3: Area and Memory utilization of the HashMem architecture variants.

5.2 HashMem Compared to Recent Related Work

In this section we attempt to make a fair comparison between our architecture implementations and other researchers' architectures. We evaluate and compare our architecture with two kinds of architectures. Firtsly with architectures that seem to be closer to our architectural ideas, such as the use of hashing to retrieve approximate match information and especially the use of pattern memories to provide exact match with fewer gates. For this comparison group we present the Bloom Filters Approach introduced by Dharmapurikar et al. [16; 3], the RDL+ROM architecture implemented by Cho and Magnione- Smith [11] and finally PHmem scheme introduced by Sourdis et al. [34]. Due to some similarities between these architectures and our we will try to make a more detailed comparison explaining any architectural and performance differences as well. Then, we will compare with some other architectures that are propably the best of the recent works around the pattern matching area.

In order to attempt a more fair comparison with those architectures that use a combination of memories and logic in order to perform a intrusion detection, we include the previously defined performance metric PEM/m. Using both the PEM and PEM/m values makes it easier to evaluate and compare the efficiency of each architecture vs. HashMem.

A firstly proposed scheme that performs hashing in order to retrieve approx-

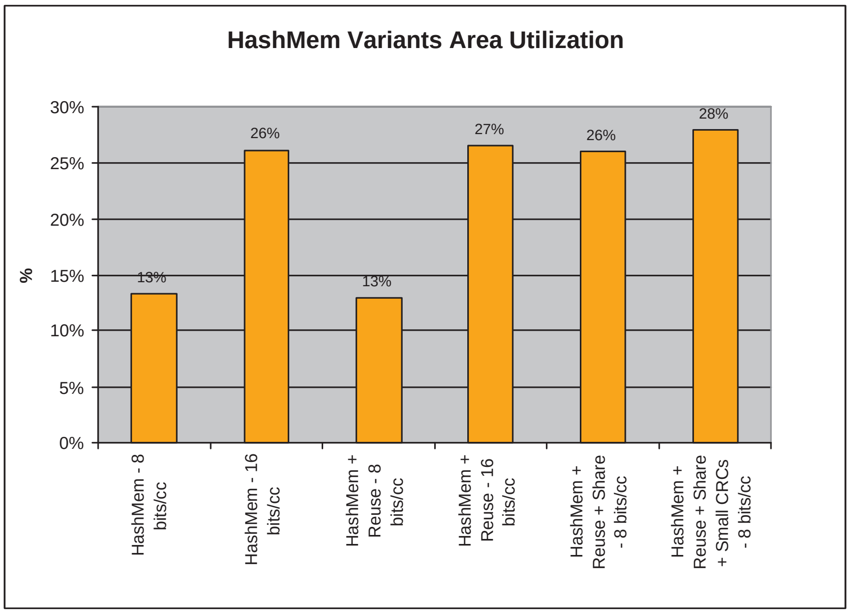

Figure 5.5: The area utilization of the total device area for each implementation variant.

imate match information of intrusion patterns is that of Bloom Filters introduced by Dharmapurikar et al. Their basic characteristics are that they require multiple hash functions and memories and also that they allow false positives, while we eliminate the extend approximate filtering by exactly defining the intrusion pattern using the two memory levels. This architectures claims that is able to store up to 35,475 patterns using 25 bloom filters, but including only patterns between 2 and 26 characters long, at a total amount of 420,000 characters. However, only ∼1,400 patterns where actually stored in their implementation. On the other hand, our architecture implements the complete Snort ruleset, performing the detection of patterns of width between 1 and 107 characters. Eventhough we have included almost the 1/20 of the patterns they may store, our design can be easily extended including even more patterns without significant overhead in area cost and with no increase in memory cost. However, our area cost in terms of logic cell per character is slightly greater. Moreover, our memory use is slightly lower compared to the Bloom filter implementation, that also has potential to add more patterns without increasing the cost. But, since their throughput is limited in 0.5 Gbps, our PEM value is much better. In terms of fairness, we should consider that VirtexE is a little slower than the devices we used, but not slower than 35% while our slowest design

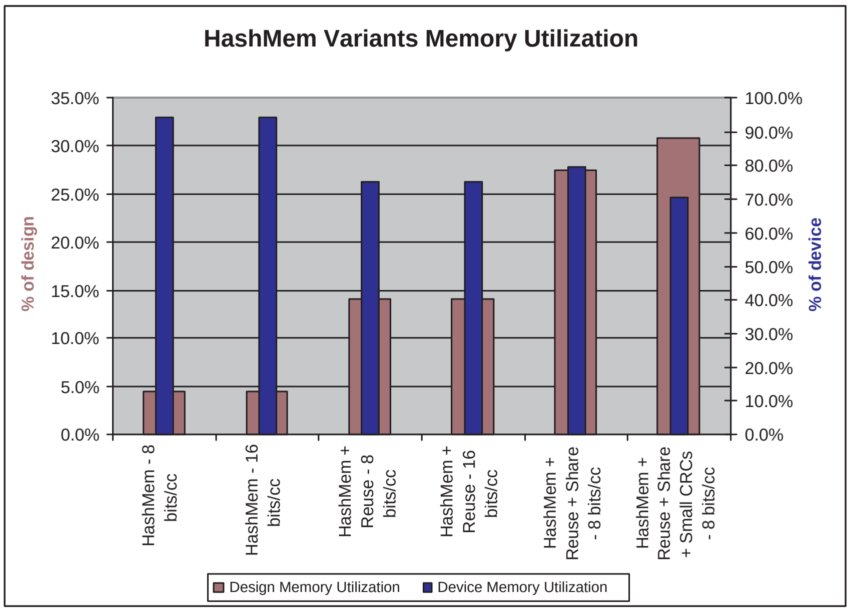

Figure 5.6: The memory utilization of the total device memory (right) and the used blocks of memory (left).

has about 200% higher frequency. Finally, Bloom Filters appear to have a great PEM/m value since they claim that it can accomodate up to 420k characters. Thus, making the calculations using the implemented number of characters, the PEM/m value would become much more low. The diagrams in figure 5.7 shows a graphical comparison between our designs and the Bloom filter approach, in terms of the PEM and PEM/m value, throughput and area cost both in logic and memory.

Cho and Magnione-Smith used a CAM to match short patterns and to match unique prefixes of longer search patterns. Their RDL+ROM architecture occupies more area in logic as it was expected since it uses memories only for wide patterns and continues to use (reconfigurable) distinct logic filters for shorter patterns groups. Actually, in their work they report directly the reported logic cells in order to measure area, but they also report the percentage of the occupied slices which number much larger than the half of the number of the logic cells. This happens because in some cases in one slice only one logic cell is used. Therefore, we took the number of reported slices as a metric in order to calculate the real approximate number of the occupied loic cells since each slice contains two logic cells. Our area cost superiority is shown more clear at the logic cells/character value, where we cost less than half. Still, they achieve an operating frequency (and throughput as well) close to ours we used a device from the same family. As a result, the PEM value is also much lower than our coresponding value. Finally, the point that this design seems to have advantages againts ours is that of the memory usage. They achieve to use only 162 kbits of built in memory since they rearrange their stored data in such a way in order to improve utilization and to reduce the required memory blocks. This is also shown more clearly in the PEM/m value where it seems to achieve almost 6.4. In the diagrams included in figure 5.8 we can also see a graphical representation of their evaluation results against ours.

The Perfect-Hashing sceme introduced by Sourdis et al. is even closer to our design which makes the comparison is much easier and more interesting. He also uses an indirection memory and a pattern memory that contains the stored patterns for the final match. Our HashMem scheme generally requires almost two times more memory to accomodate the search patterns, but not in every one of its variants. However, we believe that because of our low memory utilization we may easily upgrade our scheme and accomodate much more new intrussion patterns in the same memory structures. Moreover, it seems that the perfect-hashing modules requires more logic cells to be implemented than our simple CRC generators, resulting in more than double logic cells and also logic cells per character. As long as the throughput is conserned, the two architectures are pretty close to each other. Finally, according to the PEM results, our architecture seems to be better by a factor of 2 to 3. In addition to all the above, one of the perfect-hashing variants uses double memory in order to increase the performance. In that case the memory needs are equivelant to ours, achieving better throughput, but with a significant area cost, which doesn't manage to affect the PEM value. Even the PEM/m value is quite good but is still slightly lower than ours. These implementation variants are compared against ours also in the four diagrams about PEM, PEM/m throughput and area cost, in figure 5.9 but only those implemented in Virtex2 devices. Next, in figure 5.10 we present the corresponding diagrams for implementations in Spartan3 family devices too.

After having compared the HashMem arcitecture to other somewhat similar to ours designs, we continuing the comparison with architectures that have different approach. To begin with, propably the best FPGA-based string matching architecture is the DCAM approach (pre-decoded CAMs) proposed by Sourdis et al. Generally, this approach achieves same and some times slighlty better throughput, comparing to the implementations that have same number of input bits per cycle. Due to its fine grained pipeline it reaches high frequency but have a significant cost in area. Therefore, the implementation with the best performance has PEM value at 4.05 which is quite low comparing to those architectures that use memories to limitize the detection logic.

Another architecture that is more close to the previously described DCAM architecture was presented by Baker and Prasanna [4; 5]. Comparing between both of their main presented approaches, the unary-based and the tree-based, they support similar throughput with the second one having slightly better area cost. However, they both have lower throughput comparing to out HashMem architecture using devices from the same family. Moreover, they also have a greater area cost, as expected to be for this kind of architecture. Finally, their PEM values are similar and sometimes worse than the DCAM architecture, and as result even worse vs. the HashMem architecture.

At last, another architecture which uses decoded NFAs (Non-deterministic Finite Automata) is presented. There are some aproaches on this architecture that support input width up to 512 bits. Making comparisons of our architecture to NFAs variants of equal input widths we may easily notice that the two architectures achieve almost equal throughput but very large difference in area cost. As a result, the performance (PEM value) is much worse for NFAs comparing to ours.

Table 5.4 summarizes performance, cost, PEM and PEM/m results of HashMem and all the above FPGA-based string macthing architectures. In most cases, the throughput remains better or equal to other architectures that also store the complete Snort ruleset. The area cost however, is one of our strong points of our proposal being almost at half value comparing to next lowest cost value (except that of Bloom Filters which allows false positives). Based on performance, HashMem is still the best being the only architecture that achieves PEM value over 10.

5.3 Evaluation of the New Ruleset Upgrade

All the HashMem variants were implemented using the official Snort rule set dated early 2004 that consists of 1,474 rules and a total of 18,636 characters. These patterns are of 47 distinct widths varying from 1 to 107 bytes. We attempted to upgrade our designs using the last published rule set that is dated May of 2005. This new set of patterns has a significant larger number of patterns reaching up to

Design	Input bits	Device	Frequency (MHz)	Throughput (Gbps)	Logic Cells	Chars	LC / char	MEM (kbits)	PEM	PEM/m
HashMem + Reuse + Share + Small CRCs	8	Virtex2Pro 7	339	2.712	2,759	18,636	0.15	558	18.32	6.12
		Virtex2 500	251	2.008	2,760		0.15		13.56	4.53
		Spartan3 1500	244	1.952	2,766		0.15		13.15	4.39
HashMem + Reuse + Share	8	Virtex2Pro 7	338	2.704	2,570	18,636	0.14	630	19.61	5.80
		Virtex2 500	250	2.000	2,570		0.14		14.50	4.29
		Spartan3 1500	244	1.952	2,570		0.14		14.15	4.19
HashMem + Reuse	8	Virtex2Pro 20	333	2.664	2,632	18,636	0.14	1,188	18.86	2.96
		Virtex2 3000	244	1.952	2,636		0.14		13.80	2.16
	16	Virtex2Pro 20	322	5.152	5,219		0.28		18.40	2.89
		Virtex2 3000	232	3.712	5,230		0.28		13.23	2.07
HashMem	8	Virtex2Pro 40	292	2.336	5,172	16,879	0.31	3,258	7.54	0.39
	16		285	4.560	10,160		0.60		7.60	0.39
Attig et al. Bloom Filters	8	VirtexE 2000	63	0.502	36,720	420k	0.09	629	5.58	37.24
Cho et al. RDL+ROM	8	Spartan3 400	200	1.600	~8000	20,800	~0.38	162	~4.21	~5.41
		Spartan3 1000	238	1.900	>8000		>0.38		<5	<6.42
Sourdis et al. PHmem	8	Virtex2 1000	264	2.108	6,832	20,911	0.33	288	6.45	4.68
			361	2.886	9,426		0.45	576	6.40	2.32
		Spartan3 1000	216	1.724	7,888		0.38	288	4.57	3.32
	16	Virtex2 1000	260	4.167	11,224		0.54	306	7.76	5.31
			358	5.734	13,544		0.65	612	8.85	3.02
		Spartan3 1000	207	3.317	12,200		0.58	306	5.69	3.89
Sourdis et al. DCAM	8	Virtex2 3000	335	2.678	17,538	18,036	0.97	0	2.75	undef.
		Spartan3 1500	250	2.000	19,902		1.10		1.81	undef.
Baker et al. Unary	8	Virtex2Pro 100	185	1.488	8,056	19,584	0.41	0	3.62	undef.
Baker et al. Tree-based			237	1.896	6,340		0.32		5.86	undef.
Clark et al. NFAs	32	Virtex2 8000	219	7.004	54,890	17,537	3.13	0	2.24	undef.

Table 5.4: Detailed comparison of HashMem and previous FPGA-based string matching architectures.

2,187 rules. Moreover, the amount of separate widths is now 59 while the wider pattern counts 122 characters. Meanwhile, the higher increase comes from the total number of the matching characters which has become almost double, counting 33,613 characters.

The only factor that defines the upgrade capability of our architecture is the existence of the suitable CRC polynomials. Thus, what we had to do at first was to look for the CRC polynomials that satisfy the requirements of our architecture. The polynomial detection software needed less than an hour to find polynomials

Design	Input bits	Logic Cells	Chars	LC / char	MEM (kbits)	kbits / char
HashMem + Reuse	8	4,190	33,613	0.12	1188	0.035
	16	8,271		0.25		
	8	2,632	18,636	0.14		0.064
	16	5,219		0.28		

Table 5.5: Area cost details and comparison of the “HashMem + Reuse” implementation between the old and the new ruleset.

for the new ruleset. Considering that the number of the long strings has grown up more than the shorter ones, since the number of characters has become almost double, the strings contained in each group have been increased much and in all cases there was needed new CRC polynomials.

Unfortunately, the number of patterns and parts that are to be stored in the pattern memory structures exceed the number of 256 at most cases. Since the widest memory block primitive is 512 entries deep, we can only implement the “HashMem + Resuse” variant for 8 and 16 bits input per cycle. This happens because two distinct widths cannot fit in the existing memory blocks, that have 512 entries, which is necessary for the “sharing” improvement variant. Thus, we only present some area cost results for this HashMem variant.

Table 5.5 shows the area cost in terms of cogic cells and memory and also the ratio of LC/char. We have also added the old values of the same metrics in order to compare and evaluate the addition of the new rule set. We notice that the number of the logic cells has increased by a factor of $\sim$1.6 while the total number of characters has increased by $\sim$1.8 and therefore the ratio logic cells per character has dropped down to 0.12 and 0.25 respectively for the two variants.Moreover, since we used the same memory structures the kbits of memory per character has reduced almost at half value.

5.4 Summary

In this chapter we presented the synthesized reports of HashMem architecture variants. We tried to evaluate and compare these designs analyzing the area cost in terms of logic and memory usage, which sometimes varies a lot between each approach. Then, we attempted a fair comparison of our work with other pattern matching approaches, firstly with those which perform using memories and then with other architectures which appear to be the best representative exact matching approaches. Additionally, a new performance metric including both logic and memory cost was introduced for a better evaluation.

Considering the area cost in terms of logic cells per character, our HashMem approach achieves the best ratio in all the variants we have implemented, except from our initial approach. The only architecture that seems to have a better value in this metric is the “Bloom Filters”, but this happens because we have calculate this ratio using the theoritical amount of characters that it may accomodate (420,000 characters) while they have actually implemented only 1,400 patterns which total number of bytes is much lower. We, should also notice that the also the two efficiency metrics have been calculated on the same assumption.

Additionaly, as long as the PEM is concerned, our three improved designs are much more efficient than any other published works, with values over 13. The next best published value is that of “PHmem” that achieves a value of 8.85. This happens because of the different balance we achieved between memory and logic usage. Including the memory cost, our architecture’s efficiency is still comparable or better than other recent researchers’ results, as described by the PEM/m.

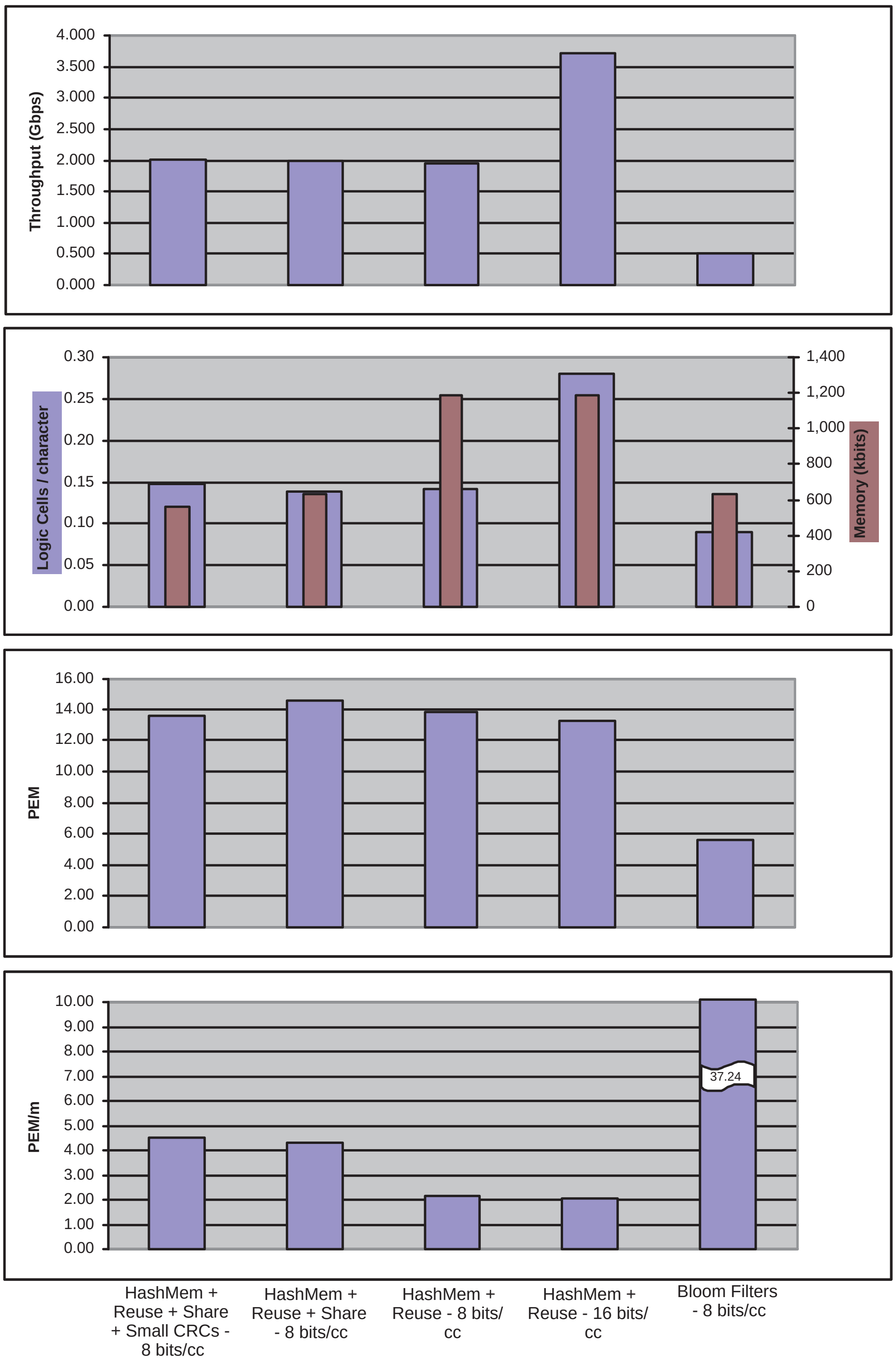

Figure 5.7: HashMem (in Virtex2) compared to Bloom Filters architecture (in VirtexE).

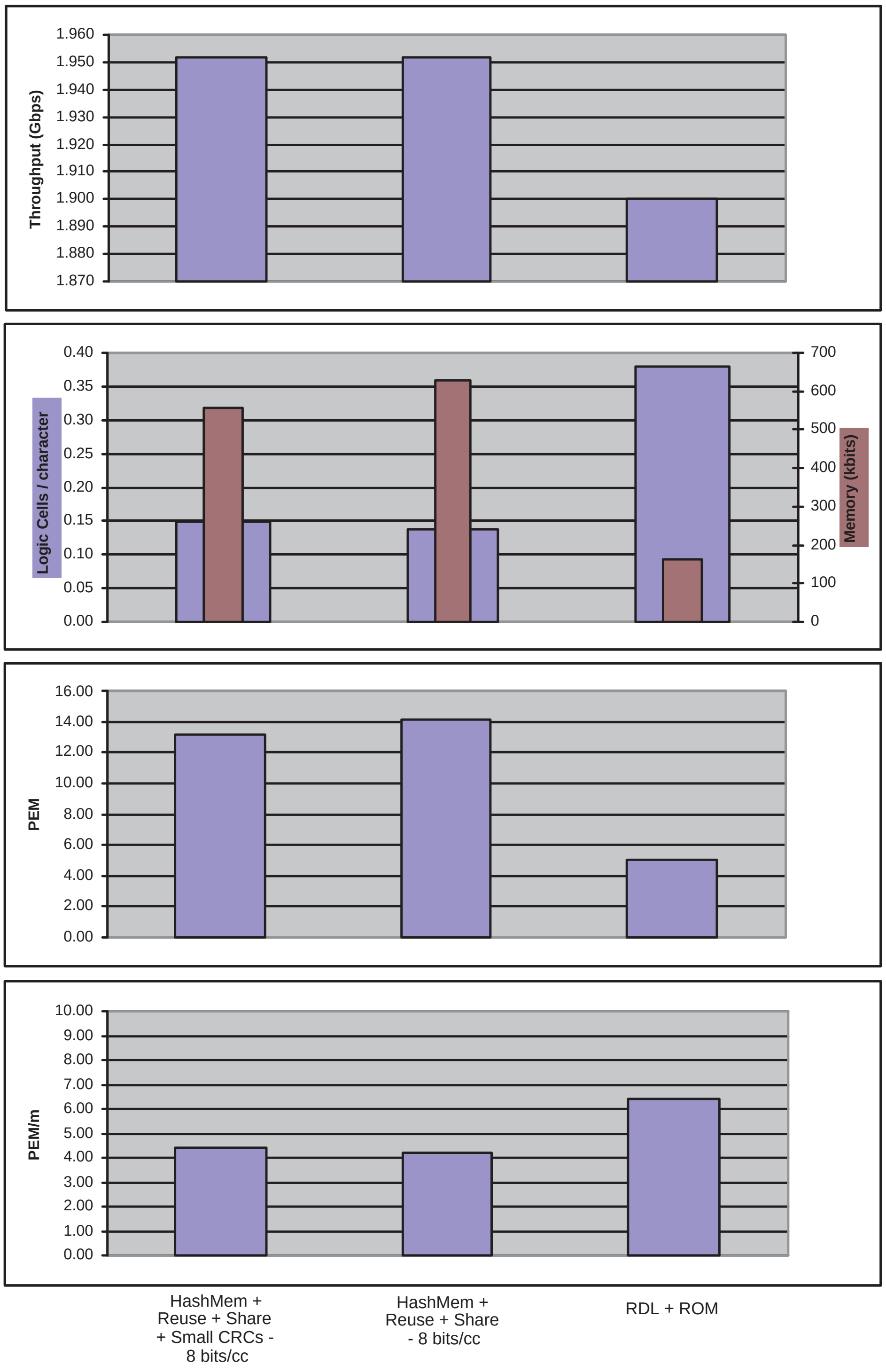

Figure 5.8: HashMem compared to Cho's RDL+ROM asrchitecture, both implemented in Spartan3 family devices.

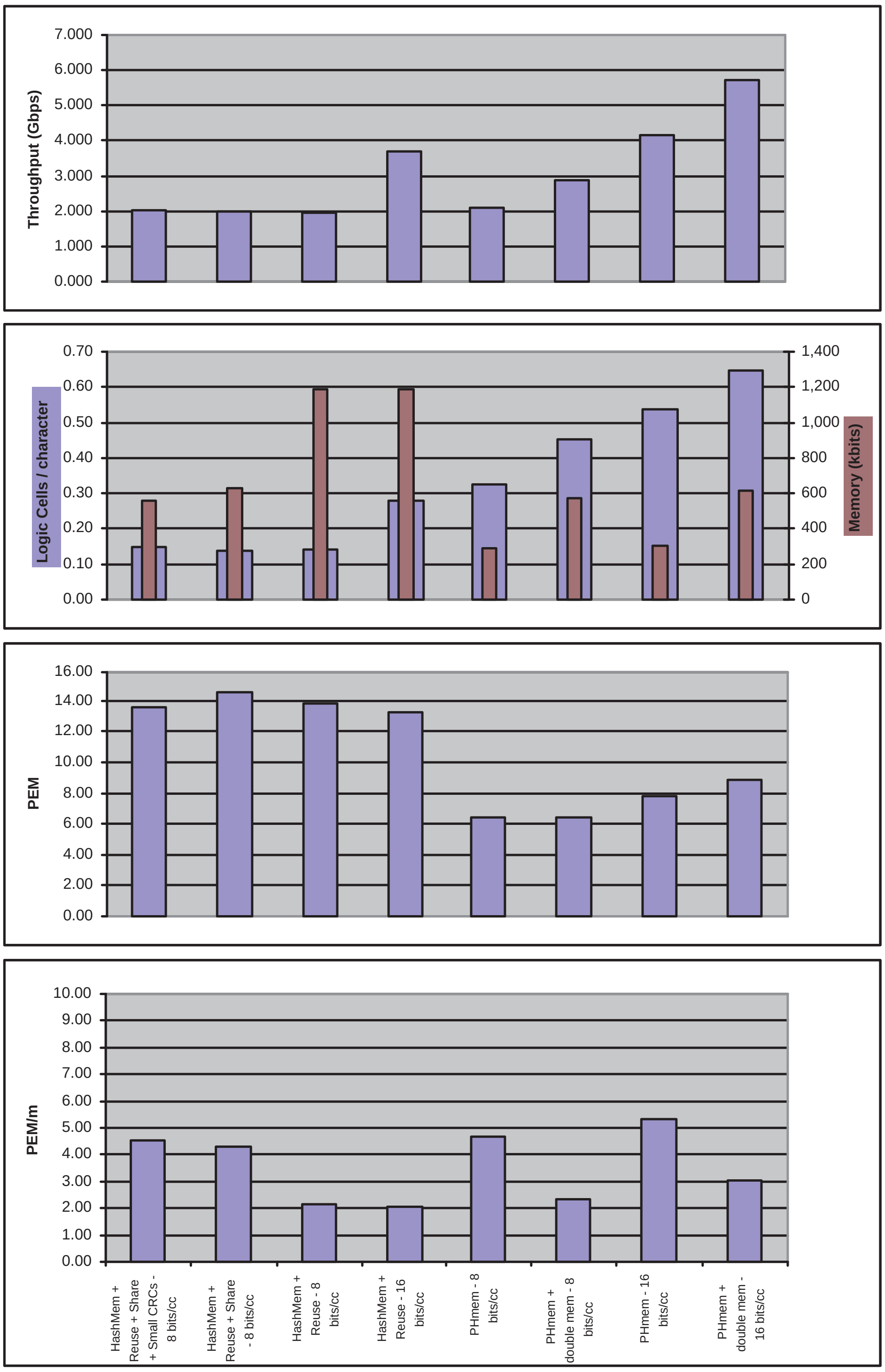

Figure 5.9: HashMem compared to the PHmem architecture, for implementations in Virtex2 family devices.

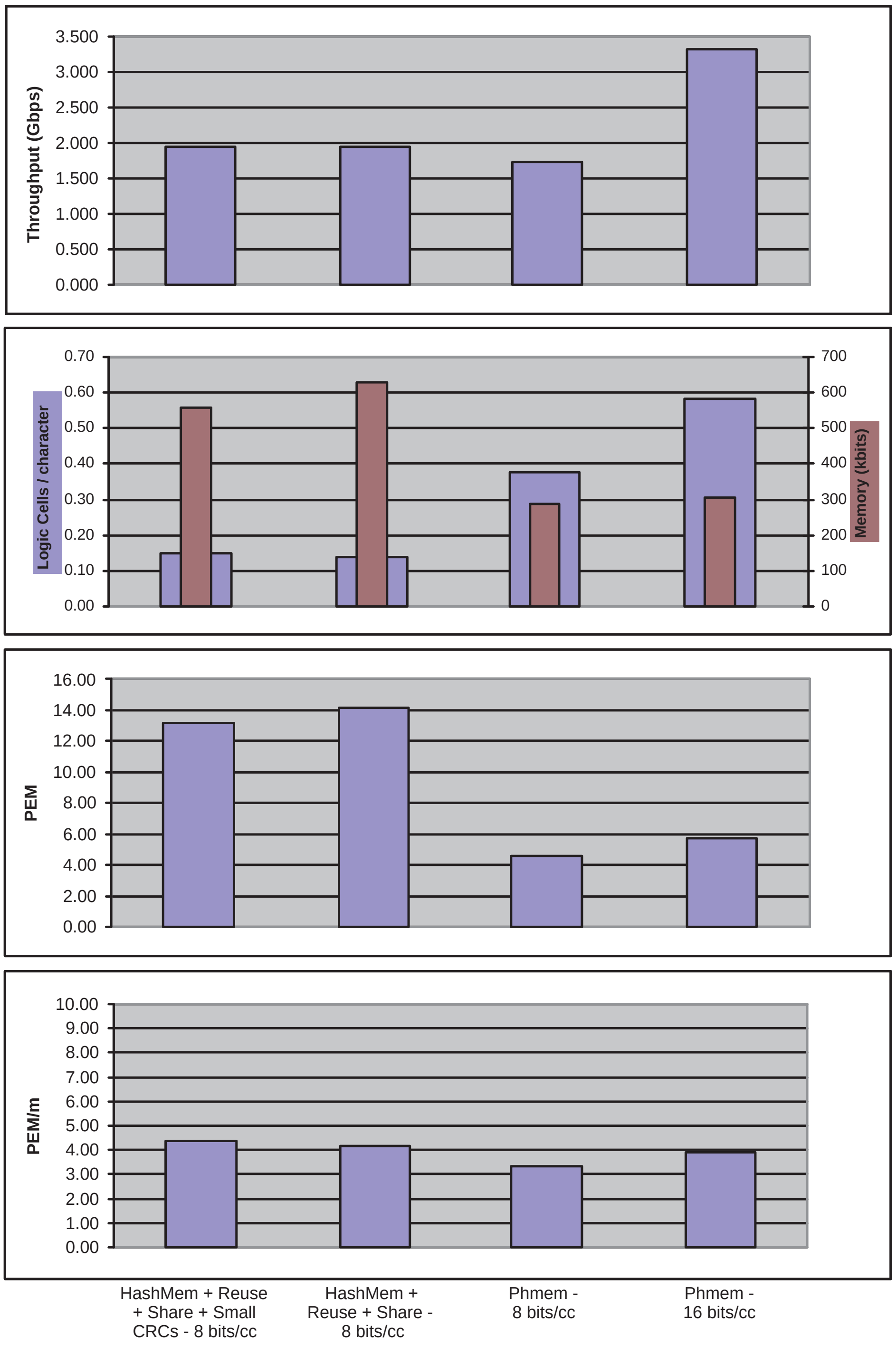

Figure 5.10: HashMem compared to the PHmem architecture, for implementations in Spartan3 family devices.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this dissertation we entered the area of networks intrusion systems and discussed about hardware approaches in pattern matching as this constitutes NIDS' bottleneck. Signature based IDS monitor incoming packets on the network and compare them against a database of known threats. Therefore, pattern matching must be performed at network speed and this constitutes it as the most computational part. Here comes the hardware assistance. We analyzed the common factors of such hardware approaches and investigated their efficiency. We investigated the Snort rule set, as one of the most popular NIDS used by other hardware approaches too, and used its patterns in our implementations. Moreover, since the signature database is often updated, FPGAs are offered as a flexible solution as they can be reconfigured.

The published works around Snort-like patterns matching have different approaches implementing finite automata, dedicated CAM logic or other CAM-like structures including memories and hashing. Moreover, some are limited in performing fast but approximate matching allowing false positive. The goal of this work was to combine hashing and memories in order to perform cost effective, exact pattern matching. The HashMem architecture is indeed very effective in achieving low logic cost for snort-like pattern matching, with about half the cost of logic that other approaches. The amount of memory used though is clearly larger that the minimum necessary (19Kbytes, or 152Kbits). The occupied memory however, is not necessarily wasted. Theoritically, it offers the opportunity to accommodate future expansions in the pattern sets without any increase in the amount of memory or logic. In fact, adding the last published Snort rule set, the increase of the total number of the search characters was so great that eventhough they fit in our memory structures, we couldn’t use the improved design that reduces the memory usage using the “sharing” feature. However, even in the medium rated design, the results were encouraging, giving spark for further improvements.

While in practice we found it is easy to find small and efficient CRC polynomials to achieve the desired property of unique addresses for the search patterns, this is a critical requirement of HashMem. However, adjusting the new rule set in our architecture is an evidence that the polynomial search is still easy since we achieved the desired target using 12-bit CRCs and as a result with limited changes in our logic.

To sum up, HashMem architecture designs achieve comparable or better throughput but what is worth noticing is the performance efficiency metric (PEM), which is commonly used by other researchers as a fair efficiency metric. Compared to other works, HashMem achieves the higher values in all variants, while having good standing in the PEM/m metric that we introduced as another efficiency metric. Finally, we believe that we achieved to offer a different balance between logic and memory usage, introducing another and simple way of hashing which may be used to further investigation and improvement.

6.2 Future Work

Viewing again the search for small and efficient CRC polynomials as a critical requirement of HashMem, we are motivated for future reaserch. We may also investigate ways to further improve the flexibility of the existing structures. Some initial ideas even come from the results we took by adding the new Snort rule set in our architecture. In that case the problem wasn’t met at the CRC search but from the great increase of the total pattern characters from the initial ruleset we used to begin our architecture. The requirements in the pattern memories made the use of the existing implementations based on memory sharing prohibitive. Thus, a way of better utilizing the memory blocks is necessary since the need of storing more and more characters may require too many memory blocks.

One choice in order to overcome such kind of problems is to extend the basic groups of width that use dedicated memory structures. Till now we implemented only the straightforward structures for widths of 3 to 16 bytes and then performed partitioning of the wider patterns into parts that fit in these structures. Thus, we could include some more basic structures for widths over 16 bytes. The only extra cost relies on the increase of the extra memory blocks required for these structures and also the extra comparators required for the new fundamental pattern widths. At first sight, this cost seems more preferable than the simple solution of adding the required space in the existing memories, which doubles the number of the pattern memories, but does not include the overhead of extra comparators. For exaple, adding structures for widths of 17 and 18 characters, the extra memory cost is 12 blocks for the "HashMem + Reuse" variant or 6 blocks for the two "+ Share" variants, and adding widths of 19 and 20 characters the extra cost is another 14 or 7 blocks respectively, while the doubling of the pattern memory length results in an increase of 36 blocks for the first design or 19 blocks for the second one. However, there is a tradeoff which makes more preferable the last choice, since the continuous and great increase of the intrusion patterns force us to build a design which must require the least possible adjustments in order to get updated. Adding some more memory blocks now, we double the pattern capacity of our designs with minumum changes and with no more wide comparators.

The major parameters that effect the HashMem Architecture's efficiency are the logic and memory cost but also the throughput. Our best architecture variant that implements memory sharing does not permit the use of the second memory ports in order to double the throughput in terms of two-byte input per cycle. Both Cho et al. in RDL+ROM [11] and Sourdis et al. in PHmem architecture [34] achieve a better pattern memory utilization by storing in the one memory structure patterns of variable widths. According to our HashMem architecture we believe that the use of CRC generators is a very good approach for performing hashing among the various patterns, as they are easy to find and have low logic cost. Therefore, another approach could be based on using the same scheme of hashing and memory lookup but on a different base. We could combine patterns of different widths in separate groups and then perform CRC hashing on a larger set of patterns. Since CRC generators have static input width, this is possible by selecting statically same width partitions from these patterns ensuring that each pattern partition is unique for the complete set of each group. As a result, these patterns are then stored in the same memory structures, as they are addressed by the same indirection memory. Moreover, we may retain the existing idea of partitioning the long patterns into smaller ones, performing partial matches composition in order to determine the overall match.

An important feature of this approach is that this scheme requires only a single CRC generator -> a single port of indirection memory -> a single port of pattern memory. This structure is replicated for each of the pattern groups. Therefore, such a feature would also allow the doubling of processing throughput using the second read memory ports. As in the presented HashMem architecture, this would only increase the logic cost requiring no more memory blocks.

The only possible drawback of this approach is that since the number of each group will be larger than in the clasical groups per width we used till now, it might be harder to find a CRC polynomial that complies with the same demand of unique CRC per group pattern. For that reason we would forced to increase the CRC width in order to enlarge the density of the hash space and make the CRC search easier. More bits in the CRC length may also not increase much the logic cost of the generators since this depends on the polynomial too. But it would increase the length of the indirection memories, in other words increase of the number of the required memory blocks.

Appendix A

A.1 CRC Theory - Characteristics of the Polynomials

The CRC (Cyclic Redundancy Check) is a sophisticated checksum that has long been the most commoon means of testing data for correctness. The basic idea behind CRCs is to treat the message string as a single binary word, and divide it by a key word. The remainder left after the division constitutes the ”check word” for the given message.

All CRCs are binary polynomials whose coefficients are the binary bits of the ”divisor”. Although the definition and selection of the CRC is quite complex, its use is not. One of the most common CRC polynomials is the CCITT form used by IBM’s SDLC protocol. It is of the form:

$$x^{16} + x^{12} + x^{5} + 1$$

which corresponds to the 17-bits number 0x11020. This means that the input data stream is exclusive ORed into a 16-bits shift register that has feedback terms at the bit locations with coefficients in the formula, one bit at a time. Each register bit is a function of the CRC to that point, the input data and the feedback taps. The generated CRC is a 16-bits word. In general, a polynomial with k bits leads to a (k-1)-bits CRC.

This method is obviously not foolproof because there are many different message strings that give the same CRC when input to a CRC generator. In fact, when the message string is n-bits wide and the CRC is m-bits wide, each generated CRC is the same for 2̂(n-m) differenet strings. Thus, by choosing a higher degree polynomial the number of the different message strings that result to the same CRC is decreased.

The above algorithm describes the straightforward CRC implementation and it operates at the bit- level. In our case data are input in one and two bytes per cycle and this algorithm is inefficient to execute while it has to loop once for each bit. This means that the duration of a CRC calculation is different for each input string according to its width. Thus we use separate full parallel CRC generators for each input data width, which results in concurrent CRC outputs of all generators.

A.2 Software assistance

In order to find the required polynomials that guarantee distinct addresses for each search pattern we had to automate the process. Thus, after having extracted the useful information from the Snort Ruleset we grouped them according to their widths. Then, having we implemented a software in C++ language that uses this grouped characters as input, we search for the desired polynomials. This is the most computational ppart of the whole process. In order to separate long patterns into shorter ones, we automated this procedure too.

The above software, after having completed the polynomial detection process, it generates a table with each pattern having its corresponding CRC value. It also generates a files in such format in order to be used as initialization files for both the indirection and the pattern memories.

Since we have found the proper polynomials, we had to implement the corresponding CRC generators for each one of them. In order to do so, we used an automatic VHDL code generator that implements a fully parallel unpipelined CRC generator [17]. Then, using a script we may convert this dataflow code into a fully fine-grained pipelined code.

Bibliography

[1] A. Aho and M Corasick. Fast pattern matching: an aid to bibliographic search. In *Commun. ACM*, volume 18(6), pages 333–340, June 1975.

[2] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis. E^2xB: a domain-specific string matching algorithm for intrusion detection. In *Proceedings of the 18th IFIP International Information Security Conference (SEC2003)*, May 2003.

[3] M. Attig, S. Dharmapurikar, and J. Lockwood. Implementation results of bloom filters for string matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[4] Z. K. Baker and V. K. Prasanna. Automatic synthesis of efficien intrusion detection systems on FPGAs. In *Proceedings of 14th International Conference on Field Programmable Logic and Applications*, August 2004.

[5] Z. K. Baker and V. K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[6] Z. K. Baker and V. K. Prasanna. Time and area efficient reconfigurable pattern matching on FPGAs. In *Proceedings of FPGA '04*, 2004.

[7] R. Boyer and J. Moore. A fast string match algorithm. In *Commun. ACM*, volume 20(10), pages 762–772, October 1977.

[8] Long Bu and John A. Chandy. FPGA based network intrusion detection using content addressable memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004. Napa, CA, USA.

[9] Forbes J. Burkowski. A hardware hashing scheme in the design of a multiterm string comparator. *IEEE Transactions on Computers*, 31(9):825–834, 1982.

[10] Young H. Cho and William H. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[11] Young H. Cho and William H. Mangione-Smith. 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] 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.

[13] 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.

[14] C. R. Clark and D. E. Schimmel. Scalable parallel pattern-matching on high-speed networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[15] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of snort. In *DISCEXII, DAPRA Information Survivability conference and Exposition*, June 2001. Anaheim, California, USA.

[16] Sarang Dharmapurikar, Praven Krishnamurthy, Todd Spoull, and John Lockwood. Deep Packet Inspection using Bloom Filters. In *Hot Interconnects*, August 2003. Stanford, CA.

[17] EASICS web site. http://www.easics.com.

[18] M. Fisk and G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection. In *Techical Report CS2001-0670 (updated version)*, University of California - San Diego, 2002.

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

[20] 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.

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

[22] Dan Gusfield. Algorithms on strings, trees, and sequences: Computer science and computational biology. In *University of California Press*, CA, 1997.

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

[24] J. W. Lockwood. An open platform for development of network processing modules in reconfigurable hardware. In *IEC DesignCon '01*, January 2001. Santa Clara, CA, USA.

[25] Xilinx BeNeLux Marc Defossez, FAE. Using the virtex look-up tables. Applications -Virtex, January 2005.

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

[27] G. Papadopoulos and D. Pnevmatikatos. Hashing + memory = lowcost, exact pattern matching. In *15th Int. Conference on Field Programmable Logic and Applications*, 2005.

[28] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Administration Conference*, November 7 -12 1999. Seattle Washington, USA.

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

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

[31] Sourcefire. Snort 2.0 - detection revised. In *http://www.snort.org/docs/Snort_20-v4.pdf*, October 2002.

[32] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In *Proceedings of 13th International Conference on Field Programmable Logic and Applications*, September 2003.

[33] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed nids pattern matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[34] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis. A reconfigurable perfect-hashing scheme for packet inspection. In *15th Int. Conference on Field Programmable Logic and Applications*, 2005.

[35] S. Wu and U. Mander. A fast algorithm for multi-pattern searching. In *Techical Report TR-94-17*, University of Arisona, 1994.

[36] Xilinx. Virtex-ii platform fpgas: Complete data sheet. DS031 v3.3, June 2004.

[37] Xilinx. Spartan-3 platform fpgas: Complete data sheet. DS099 v1.4, January 2005.

[38] Xilinx. Virtex-ii pro platform fpgas: Complete data sheet. DS083 v4.3, June 2005.