A RECONFIGURABLE PERFECT-HASHING SCHEME FOR PACKET INSPECTION

Ioannis Sourdis†, Dionisios Pnevmatikatos‡, Stephan Wong†, Stamatis Vassiliadis†*

†Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
The Netherlands
{Sourdis,Stephan,Stamatis}@CE.ET.TUDelft.NL

‡Microprocessor and Hardware Laboratory,
Electronic and Computer Engineering Dept.,
Technical University of Crete,
Chania, Greece
Pnevmati@MHL.TUC.GR

ABSTRACT

In this paper, we consider scanning and analyzing packets in order to detect hazardous contents using pattern matching. We introduce a hardware perfect-hashing technique to access the memory that contains the matching patterns. A subsequent simple comparison between incoming data and memory output determines the match. We implement our scheme in reconfigurable hardware and show that we can achieve a throughput between 1.7 and 5.7 Gbps requiring only a few tens of FPGA memory blocks and 0.30 to 0.57 logic cells per matching character. We also show that our designs achieve at least 30% better efficiency compared to previous work, measured in throughput per area required per matching character.

1. INTRODUCTION

The proliferation of Internet and networking applications, coupled with the wide-spread availability of system hacks and viruses have increased the need for network security. Deep packet inspection is performed by firewalls and intrusion detection/prevention systems (IDS/IPS) to provide sufficient protection from attacks. Such systems check the packet header, rely on pattern matching techniques to analyze the packet payload, and make decisions on the significance of the packet body. Matching every incoming byte, though, against thousands of pattern characters at wire rates is a computationally intensive task. Measurements on Snort IDS show that 80% of total processing is spent on string matching in the case of Web-intensive traffic [1]. IDS based on general-purpose processors can only achieve a throughput up to a few hundred Mbps. On the other hand, hardware-based solutions can significantly increase performance and achieve much higher throughput.

In the past, several hardware units have been proposed for FPGA-based IDS pattern matching [2, 3, 4, 5, 6, 7, 8, 9]. Generally speaking, the performance of FPGA-based systems is promising and shows that FPGAs can support the increasing needs for network security. In this paper we extend previous pattern matching techniques [5, 8, 9, 10, 11]:

- We propose a perfect-hashing technique to determine a single possible match based on the incoming data.
- We use a centralized, banked pattern-memory to store the entire Snort IDS set of patterns, and optimize the rule placement to increase memory utilization.
- We exploit pipelining, parallelism, and memory replication to increase performance.

In doing so, we save a significant amount of resources. Our designs can support a throughput ranging from 1.7 to 5.7 Gbps, while requiring only a few tens of FPGA memory blocks and 0.30 to 0.57 logic cells per matching character. Since recent FPGA devices include hundreds of memory blocks and tens of thousand logic cells, our approach can be considered as a low cost pattern matching solution.

The remainder of the paper is organized as follows: In Section 2 we describe our perfect-hashing system. In Section 3, we present the implementation results and compare with related work. Finally, in Section 4 we present our conclusions.

2. PATTERN MATCHING SYSTEM UTILIZING PERFECT-HASHING

The detection engine of an IDS consists of header matching and payload matching. In this section we describe a technique for payload pattern matching. We consider scanning the payload of every incoming packet and therefore the obtained throughput will remain constant even in worst case scenarios (targeted attacks etc.). Instead of matching each pattern separately, it is more efficient to utilize a hash module to determine which pattern is a possible match, read this pattern from a memory and compare it against the incoming data. Hardware hashing for pattern matching is a technique widely used for decades. Figure 1 depicts our Perfect-Hashing Memory (PHmem) scheme. The incoming packet data are shifted into a serial-in parallel-out shift register. The parallel-out lines of the shift register provide input to the comparator, which is also fed by the memory that stores the patterns. A selected subset of the incoming data bits are used

*Also with the Institute of Computer Science (ICS), Foundation for Research and Technology-Hellas (FORTH).

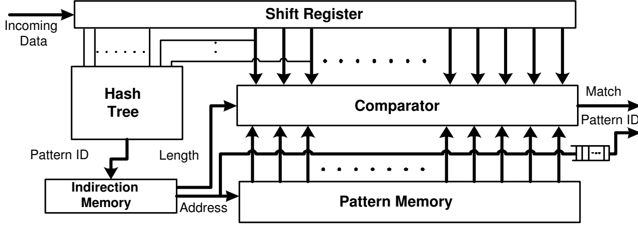

Fig. 1. Block diagram of our pattern matching approach.

as inputs to a hash module, which outputs the ID of the "possible match" pattern. For memory utilization reasons, we do not use this pattern ID to directly read the search pattern from the pattern memory. We utilize instead an *indirection* memory, similar to [11], that outputs (i) the actual address of the possible match pattern and (ii) its length. However, in our case the indirection memory performs a *1-to-1* and not an *N-to-1* mapping, since the output address has the same width as the pattern ID. This address is utilized to read the pattern, while the pattern length is used to determine how many bytes of the pattern memory and the incoming data are needed to be compared.

2.1. Perfect Hashing Tree

Our approach is based on Burkowski's multiterm string comparator [10] and Merkle's hash tree [12]. Burkowski matches substrings in order to detect which pattern would possibly match; a similar approach was presented by Cho et al. [5, 9]. Burkowski selects a unique substring for each pattern and, subsequently, uses an associative memory and encoder to match them and produce the pattern address. On the contrary, we hash the substrings in order to distinguish the given set of patterns. For this reason, we introduce a perfect-hashing method meaning that our hash modules will guarantee that no collisions will occur for a specific set of substring entries. We verified that our hash trees do not have any collisions for the given set of entries by exhaustively simulating their VHDL representation. First, we select a unique substring for each pattern (for simplicity: either prefix or suffix) and we reduce the length of the set of substrings by deleting all the columns (bit-positions) that are not necessary to distinguish the substrings. Subsequently, the remaining bit-positions provide input to our Merkle-like hash tree. In our perfect hashing scheme, the size of the hash tree depends only on the number of the substrings, and not on their length, which is an advantage compared to complete substring match approaches. Merlke's hash tree, created for public key cryptosystems and authentication, is constructed based on the idea of "divide and conquer". If we define Y as an element file $= \{Y_1, Y_2, \ldots, Y_n\}$, such that the i_{th} element is Y_i, and $H(Y)$ as the hash function of Y, then Merkle created his hash tree according to:

$$H(Y) = H_1(1st\ half\ of\ Y), H_2(2nd\ half\ of\ Y) \quad (1)$$

$$H_1(1st\ half\ of\ Y) = H_{1.1}(1st\ quarter\ of\ Y), H_{1.2}(2nd\ quarter\ of\ Y) \quad (2)$$

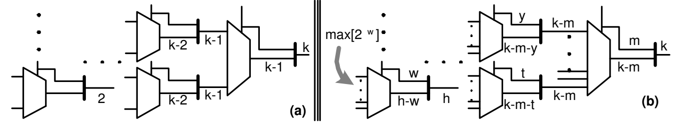

Fig. 2. (a) Binary Hash Tree. (b) Optimized Hash Tree.

and so on for the smaller parts of the element file Y.

Generating a single perfect hash function to distinguish a given set of patterns is difficult and time consuming. Merkle's method is very suitable and simplifies the hash function generation. Consequently, instead of searching for a single complex hash function, we construct a hash tree that consists of several simpler sub-hashes.

Following Merkle's methodology, we created a binary hash tree. For a given set of patterns that have unique substrings, we consider the set of substrings as a 2-D $m \times n$ matrix. Each row of the matrix (m bits long) represents a substring, which differs at least in one bit from all the other rows. Each column of the matrix (n bits long) represents a bit position of the substrings. The binary tree should have $\log_2(n)$ output bits. We construct our binary hash tree by recursively partitioning the given matrix as follows:

- Search for a hash function that separates the matrix in 2 parts, which can be encoded in $\log_2(n/2)$ bits each. We search for either a single column or a XOR combination of several columns.
- We recursively repeat the same procedure for each part of the matrix, in order to separate them again in smaller parts.
- The process terminates when all parts contain one row.

Figure 2(a) depicts the hardware implementation of the binary hash tree using 2-to-1 multiplexers for each tree node. From Equation (1) each hash-function *H(Y)* is considered as the select bit of a multiplexer and the encoded bits of 1st and 2nd halves of element file Y as the inputs of the multiplexer. The multiplexer's output combined with its select bit are considered as the encoded bits of Y. For example, a node that divides into two parts an n-element (sub-)file, which needs $\log_2(n) = k$ bits to be encoded, is represented by a $(k - 1)$-bit 2-to-1 multiplexer. The select bit of the multiplexer is either a single bit-position or a XOR function of several bit-positions. The k-bit address of Y element file consists of the $(k - 1)$ bits of the multiplexer output and the select bit of the multiplexer. Each leaf node of the hash tree is a 1-bit 2-to-1 multiplexer that separates 3 or 4 elements and each input of the multiplexer is a single bit that separates 2 elements.

The hardware implementation of a binary tree is an efficient solution to separate a set of patterns; however, we can optimize it and further reduce its area. During the generation of hashing functions we noticed that in a single search for a select bit, we could find more than one select bit (actually 2-5 bits) that can be used together to divide the set into more than two parts (4 to 32). This approach results in smaller,

in terms of area, hash trees using larger multiplexers. The block diagram of our optimized hash tree is illustrated in Figure 2(b). Each node of the tree can have more than two branches and, therefore, the size of the tree is smaller despite the use of bigger multiplexers.

The construction of our perfect hashing modules is fully automated (including VHDL generation) and takes a few minutes in case of the binary trees and a few minutes to a few hours in case of N-ary trees (depends on maximal N). Given that the place & route of such a design takes a couple of hours and that the required area is relatively small, our approach is suitable for frequent (partial) reconfiguration.

2.2. Implementation Details

To generate our PHmem design for a given set of Snort rules we first extract the pattern-matching portion of the rules, and we group them, so that each pattern in a group has a unique substring. Subsequently, we reduce the length of substrings, keeping only the bit-positions necessary to distinguish the patterns and, finally, we generate the hash trees for every reduced substring file.

We use the wider Xilinx dual-port block RAM configuration (512 entries $\times$ 36 bits), to store patterns. Therefore, we group patterns in groups of maximally 512 patterns. Patterns in the same group should have unique substrings in order to distinguish them using hashing. The grouping algorithm takes into account the length of the patterns, i.e., longer patterns are grouped first. Patterns of all groups are stored in the same memory, which is constructed by several memory banks. Each bank is dual-ported, therefore, our grouping algorithm ensures that in each bank are stored patterns (or parts of patterns) of maximally two different groups. This restriction is necessary to guarantee that one pattern of each group can be read at every clock cycle.

3. EVALUATION & COMPARISON WITH PREVIOUS WORK

In this section, we evaluate the efficiency of our overall pattern matching modules utilizing two main metrics: performance in terms of operating frequency and processing throughput (post place & route results), and area cost in terms of required FPGA logic cells. Our implementation targets Xilinx Virtex2 and Spartan3 devices. For these devices, ISE Xilinx tool has relatively accurate timing information. Although Virtex4 devices can achieve about double performance compared to Virtex2, we decided not to implement our designs in Virtex4 because ISE tool outputs only preliminary timing results for these devices. The use of block RAMs is a limiting factor for our operating frequency, therefore, we also implemented designs with double memory size that operates in half the operating frequency in relation to the rest of the circuit. However, this technique only had significant performance improvement for Virtex2 devices. Finally, we also considered the use of parallelism to increase throughput and we implemented designs that process 2 bytes per cycle.

Table 1. Comparison of PHmem and other FPGA-based string matching approaches.

Description	Input bits /cycle	Device	Throughput (Gbps)	LUTs /FFs	Logic Cells[1]	Logic Cells /char	MEM Kbits	#chars	PEM
Our Proposed Scheme (PHmem)	8	Virtex2 -1000	2.108	3,451 / 5,805	6,272	0.30	288	20,911	7.03
			2.886[2]	4,410 / 8,115	9,052	0.41	576[2]		6.93
		Spartan3 -1000	1.724	3,451 / 5,805	6,688	0.32	288		5.39
	16	Vitex2 -1500	4.167	6,675 / 9,459	10,224	0.49	306		8.52
			5.734[2]	7,659 / 11,685	12,106	0.57	612[2]		9.91
		Spartan3 -1000	3.317	6,675 / 9,459	10,868	0.52	306		6.38
[13] DCAM no grouping	8	Virtex2 -1500	2.254	8,095 / 9,125	10,016	0.55	0	18,036	4.05
		Spartan3 -1000	1.703	8,095 / 9,125	10,170	0.56	0		3.04
[6] DCAM	32	Virtex2 -6000	9.708	55,026 / 57,723	64,268	3.56	0		2.73
	8	Virtex2 -3000	2.678	13,946 / 15,677	17,538	0.97	0		2.76
[11] CRC Hash + MEM	8	Virtex2 -1000	2.000	?	2,570	0.14	630	18,636	14.5
	16	Virtex2 -3000	3.712	?	5,230	0.28	1,188		13.8
[4] NFAs	32	Virtex2 -8000	7.004	?	54,890	3.1	0	17,537[3]	2.26
[5] RDL w/Reuse	8	Spartan3 -1500	2.000	16,930 / ?	?	0.81[4]	0	20,800	2.46[4]
[9] ROM-based		Spartan3 -400	1.600	4,415 / ?	~8,000[5]	~0.38[5]	162		~4.16[5]
		Spartan3 -1000	1.900	4,415 / ?	>8,000[5]	>0.38[5]	162		<5.00[5]
[3] Unary	8	Virtex2Pro -100	1.488	?	8,056	0.41	0	19,584	3.63
	32		4.507	?	30,020	1.53	0		2.94
[7] tree-based	8		1.896	?	6,340	0.32	0		5.86
[8] Bloom filters[6]	8	VirtexE -2000	0.502	23,328 / ?	36,720	0.09	629	420k[7]	5.74

In order to evaluate our proposed scheme and compare it with the related research, we utilize the Performance Efficiency Metric (PEM), which takes into account both performance and area cost: $PEM = Throughput/(LC/Chars)$.

Our designs that process one byte per clock cycle can achieve 1.7 and 2.1 Gbps of throughput, requiring 0.32 and 0.30 logic cells per matching character in Spartan3 and Virtex2 devices respectively. On both devices, 16 block RAMs (288 Kbits) were used. A design that utilizes double mem-

[1]Two *Logic Cells* form one *Slice*. We calculate the number of logic cells required for a design according to the next equation: $Logic\ Cells = 2 \times Slices$, where slices is the reported number of used slices of the Xilinx ISE tool.

[2]Designs that have double size memory to increase performance.

[3]Over 1,500 patterns that contain 17,537 characters.

[4]Cho et al. report LUTs instead of slices. If they had used slices, their designs would have a higher area cost and a lower PEM.

[5]The design uses 4,415 LUTs and 99% of available slices, or 8,000 Logic Cells for a Spartan3-400 device (according to Xilinx datasheet). For a Spartan3-1000 the design would probably use even more logic cells, since the ISE tool uses more resources (when available) to increase performance.

[6]Bloom Filters perform approximate matching and allow false positives.

[7]25 bloom filters match 2-26 character patterns and can store 35,475 patterns each, about 420,000 characters in total. However, about 1,400 patterns were actually stored.

ory to increase the operating throughput of the pattern matching module can support about 2.9 Gbps in Virtex2, requiring slightly larger area. Designs that process 2 bytes per clock cycle achieve 3.3 and 4.1 Gbps in Spartan3 and Virtex2 devices, while needing 0.52 and 0.48 logic cells per character, respectively. Finally, the Virtex2 design that uses double size of memory and processes 2 bytes per cycle can operate at 5.7 Gbps. It is noteworthy that about 30-40% of the area required is because of the registered memory inputs and outputs and the shift registers of incoming data.

In Table 1, we attempt a fair comparison with previously reported research on *pattern matching* designs that can store a full IDS ruleset. Apart from our new results, Table 1 contains some results of our previous work and Sourdis master thesis [6, 13]. There are also results of related approaches of exact and approximate matching. Our designs have better PEM values (at least 30%) compared to the best *previous* related schemes using similar or equivalent devices. Compared to the ROM-based solution of Cho et al.[5, 9], whose approach is closer to our work, our 1-byte per cycle designs have similar performance and slightly lower area cost, but we need about 75% more memory mainly due to the indirection memories. Our 2-byte per cycle designs require almost twice the memory, but achieve double throughput with slightly higher area cost. Finally, compared to recent CRC-hashing approach of Papadopoulos and Pnevmatikatos our designs require about two times more area and half memory blocks, while our performance is 5-10% higher for our single clock domain configuration, and up to 55% higher for our pipelined memory designs.

4. CONCLUSIONS

We proposed a perfect-hashing method to determine which pattern could possibly match and perform a simple comparison afterwards between the pattern and the incoming data. We utilized fine-grain pipelining to achieve high operating frequencies, multiple clock domains to overcome memory latency, and parallelism to increase the processing throughput. This combination achieves a throughput of 1.7 to 5.7 Gbps for the entire Snort ruleset, requiring 0.30-0.57 logic cells per matching character and 16 to 34 memory blocks. Considering that most FPGAs have tens or even hundreds of memory blocks, our designs can be easily implemented in most FPGA devices. Based on PEM, our designs at least 30% better efficiency compared to the best previous published approaches. Additionally, in our approach in utilizing parallelism the memory size was not affected. This is very encouraging since it shows that we can increase manifold our throughput without any memory overhead.

5. REFERENCES

[1] M. Fisk and G. Varghese, "An Analysis of Fast String Matching Applied to Content-based Forwarding and Intrusion Detection," in *Techical Report CS2001-0670*, University of California - San Diego, 2002.

[2] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *13th Int. Conf. FPL*, 2003.

[3] Z. K. Baker and V. K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection systems on FPGAs," in *IEEE Symposium on FCCM*, 2004.

[4] C. R. Clark and D. E. Schimmel, "Scalable Parallel Pattern-Matching on High-Speed Networks," in *IEEE Symposium on FCCM*, 2004.

[5] Y. H. Cho and W. H. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," in *IEEE Symposium on FCCM*, 2004.

[6] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *IEEE Symposium on FCCM*, 2004.

[7] Z. K. Baker and V. K. Prasanna, "Automatic Synthesis of Efficien Intrusion Detection Systems on FPGAs," in *14th Int. Conf. FPL*, 2004.

[8] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation Results of Bloom Filters for String Matching," in *IEEE Symposium on FCCM*, 2004.

[9] Y. H. Cho and W. 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.

[10] F. J. Burkowski, "A Hardware Hashing Scheme in the Design of a Multiterm String Comparator." *IEEE Transactions on Computers*, vol. 31, no. 9, pp. 825–834, 1982.

[11] G. Papadopoulos and D. Pnevmatikatos, "Hashing + Memory = Low Cost, Exact Pattern Matching," in *15th Int. Conf. FPL*, 2005.

[12] R. C. Merkle, "Protocols for public key cryptosystems." in *IEEE Symposium on Security and Privacy*, 1980, pp. 122–134.

[13] I. Sourdis, "Efficient and High-Speed FPGA-based String Matching for Packet Inspection," Master's thesis, ECE Dept. Technical University of Crete (TUC), Chania, Greece, July 2004, http://www.mhl.tuc.gr/personal_pages/Sourdis/MS_thesis.pdf.