Scalable Multigigabit Pattern Matching for Packet Inspection

Ioannis Sourdis, *Student Member, IEEE*, Dionisios N. Pnevmatikatos, *Member, IEEE*, and Stamatis Vassiliadis, *Fellow, IEEE*

***Abstract*—In this paper, we consider hardware-based scanning and analyzing packets payload in order to detect hazardous contents. We present two pattern matching techniques to compare incoming packets against intrusion detection search patterns. The first approach, decoded partial CAM (DpCAM), predecodes incoming characters, aligns the decoded data, and performs logical AND on them to produce the match signal for each pattern. The second approach, perfect hashing memory (PHmem), uses perfect hashing to determine a unique memory location that contains the search pattern and a comparison between incoming data and memory output to determine the match. Both techniques are well suited for reconfigurable logic and match about 2200 intrusion detection patterns using a single Virtex2 field-programmable gate-array device. We show that DpCAM achieves a throughput between 2 and 8 Gb/s requiring 0.58–2.57 logic cells per search character. On the other hand, PHmem designs can support 2–5.7 Gb/s using a few tens of block RAMs (630–1404 kb) and only 0.28–0.65 logic cells per character. We evaluate both approaches in terms of performance and area cost and analyze their efficiency, scalability, and tradeoffs. Finally, we show that our designs achieve at least 30% higher efficiency compared to previous work, measured in throughput per area required per search character.**

***Index Terms*—Packet inspection, pattern matching, perfect hashing, reconfigurable computing.**

I. Introduction

MATCHING large sets of patterns against an incoming stream of data is a fundamental task in several fields such as network security [1]–[12] or computational biology [13], [14]. For example, high-speed network intrusion detection systems (IDS) rely on efficient pattern matching techniques to analyze the packet payload and make decisions on the significance of the packet body. However, matching the streaming payload bytes against thousands of patterns at multigigabit rates is computationally intensive. Measurements on Snort IDS [15] implemented on general-purpose processors show that up to 80% of the total processing is spent on pattern matching [16], while the overall throughput is limited to a few hundred megabits per second [16], [17]. On the other hand, hardware-based solutions can significantly increase performance and achieve higher throughput. Many hardware units have been proposed for IDS pattern matching most of them in the area of reconfigurable hardware [1]–[12], [18]. In general, field-programmable gate arrays (FPGAs) are well suited for this task, since designs can be customized for a particular set of search patterns and updates to that set can be performed via reconfiguration. Furthermore, the performance of such designs is promising and indicates that FPGAs can be used to support the increasing needs for high-speed network security.

Pattern matching is a significant issue in intrusion detection systems, but by no means the only one. For example, handling multicontent rules, reordering, and reassembling incoming packets are also significant for system performance. In this paper, we address the challenge of payload pattern matching in intrusion detection systems. We present two efficient pattern matching techniques to analyze packet payloads at multigigabit rates and detect hazardous contents. We expand on two approaches that we proposed in the past [8], [19], present and evaluate them targeting the Snort IDS ruleset. The first one is decoded CAM (DCAM) and uses predecoding to exploit pattern similarities and reduce the area cost of the designs. We improve DCAM and decrease the required logic resources by partially matching long patterns. The improved approach is denoted as decoded partial CAM (DpCAM). The second approach perfect hashing memory (PHmem), briefly described in [19], combines logic and memory for the matching. PHmem utilizes a new perfect hashing technique to hash the incoming data and determine a unique memory location of a possible matching pattern. Subsequently, we read this pattern from memory and compare it against the incoming data. We extend the perfect hashing algorithm in order to guarantee that for any given set a perfect hash function can be generated, and present a theoretical proof of its correctness. We evaluate both approaches and show that they scale well as the pattern set grows. Finally, we compare them with previous work, analyze the resources required, and discuss the cost-performance tradeoffs for each case based on a new performance efficiency metric.

The rest of this paper is organized as follows. In Section II, we discuss related work. In Sections III and IV, we describe our DpCAM and perfect hashing approaches, respectively. In Section V, we present the implementation results of both DpCAM and PHmem and compare them with related work. Finally, in Section VI, we present our conclusions.

II. Hardware-Based IDS Pattern Matching

In the past few years, numerous hardware-based pattern matching solutions have been proposed, most of them using

Manuscript received May 9, 2006; revised May 28, 2007. This work was supported by the European Commission in the context of the SARC Integrated Project #27648 (FP6). All authors are members of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

I. Sourdis is with the Department of Electrical and Computer Engineering, Delft University of Technology (TU Delft), 2628 CD Delft, The Netherlands (e-mail: sourdis@ce.et.tudelft.nl).

S. Vassiliadis, deceased, was with the Department of Electrical and Computer Engineering, Delft University of Technology (TU Delft), 2628 CD Delft, The Netherlands (e-mail: sourdis@ce.et.tudelft.nl).

D. Pnevmatikatos is with the Technical University of Crete (TUC), GR 73100 Crete, Greece, and also with the Institute of Computer Science (ICS), Foundation for Research and Technology-Hellas (FORTH), Heraklion, GR 71110 Crete, Greece.

Digital Object Identifier 10.1109/TVLSI.2007.912036

FPGAs and following CAM-like [2]–[5], [7], [8], [20], [21], finite automata [18], [1], [22], [6], or hashing approaches [19], [23], [24]. Next, we describe some significant steps forward in IDS pattern matching over the past few years.

Simple CAM or discrete comparators structures offer high performance, at high area cost [2]–[4]. Using regular expressions (NFAs and DFAs) for pattern matching slightly reduces the area requirements, however, results in significantly lower performance [1], [18], [22]. A technique to substantially increase sharing of character comparators and reduce the design cost is predecoding, applicable to both regular expression and CAM-like approaches [5]–[8], [20]. The main idea is that incoming characters are predecoded resulting in each unique character being represented by a single wire. This way, an N-character comparator is reduced to an N-input AND gate. Yusuf and Luk presented a tree-based CAM structure, representing multiple patterns as a Boolean expression in the form of a binary decision diagram (BDD) [25]. In doing so, the area cost is lower than other CAM and NFA approaches.

More recently, several hashing techniques were proposed for IDS pattern matching. Cho and Mangione-Smith proposed the use of prefix matching to read the remaining pattern from a memory and reduce the area requirements [7], [11]. However, this approach has limited performance and the restriction of patterns having short unique prefixes. Papadopoulos and Pnevmatikatos proposed a CRC-polynomial implementation to hash incoming data and determine the possible match pattern [24]. This design requires minimum logic at the cost of higher memory requirements. Another efficient and low-cost approach was presented by Attig *et al.* who used bloom filters to perform pattern matching [23]. In order to perform exact pattern matching, Attig *et al.* require external memory and the performance of the system is not guaranteed under worst case traffic (successive matching patterns or false positives).

Finally, a pattern matching approach designed for application-specific integrated circuit (ASIC) was proposed by Tan and Sherwood [12]. Instead of having a single finite-state machine (FSM) with a single incoming ASCII-character as input, they constructed eight parallel binary FSMs. Their designs support up to 10 Gb/s in 0.13-μm technology. A similar approach implemented in FPGAs was proposed by Jung *et al.* in [26].

III. Decoded CAM

Simple CAM or discrete comparators may provide high performance [2]–[4], however, they are not scalable due to their high area cost. In [4], we assumed the simple organization depicted in Fig. 1(a). The input stream is inserted in a shift register, and the individual entries are fanned out to the pattern comparators. There is one comparator for each pattern, fed from the shift register. This design is simple and regular, and with proper use of pipelining, the circuit can be fast. Its drawback, however, is the high area cost. To remedy this cost, we suggested *sharing* the character comparators exploiting similarities between patterns as shown in Fig. 1(b).

The Decoded CAM architecture illustrated in Fig. 2, builds on this idea extending it further by the following observation: instead of keeping a window of input characters in the shift register each of which is compared against multiple search patterns, we can first test for equality of the input for the desired characters, and then delay the partial matching signals. This approach

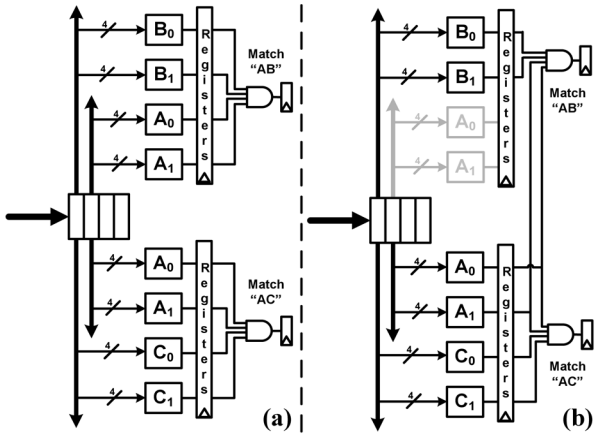

Fig. 1. Basic discrete comparator structure and its optimized version which shares common character comparators.

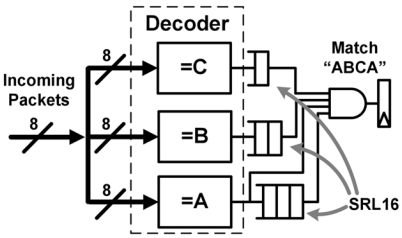

Fig. 2. Decoded CAM: Three comparators provide the equality signals for characters A, B, and C ("A" is shared). To match pattern "ABCA" we have to remember (using shift registers) the matching of character A, B, C, for 3, 2, and 1 cycles, respectively, until the final character is matched.

both shares the equality logic for character comparators and replaces the 8-bit wide shift registers used in our initial approach with single bit shift registers for the equality result(s). If we exploit this advantage, the potential for area savings is significant. In practice, about 5 $\times$ less area resources are required compared to simple CAM and discrete comparators designs [8].

One of the possible shortcomings of our approach is that the number of the single bit shift registers is proportional to the length of the patterns. Fig. 2 illustrates this point: to match a four-character long pattern, we need to test equality for each character (in the dashed "decoder" block), and to delay the matching of the first character by three cycles, the matching of the second character by two cycles, and so on, for the width of the search pattern. In total, the number of storage elements required in this approach is $(L * (L-1)/2)$ for a string of length L. For many long patterns this number can exceed the number of bits in the character shift register used in the original CAM design. To our advantage, however, is that these shift registers are true first-input–first-outputs (FIFOs) with one input and one output, as opposed to the shift registers in the simple design in which each entry in the shift register is fan-out to comparators.

To tackle this possible obstacle, we use two techniques. First, we reduce the number of shift registers by sharing their outputs whenever the same character is used in the same position in multiple search patterns. Second, we use the SRL16 optimized implementation of shift register that is available in Xilinx devices and uses a single logic cell for a shift register of any width up to 17 [27]. Together these two optimizations lead to significant area savings. To further reduce the area cost of our designs, we

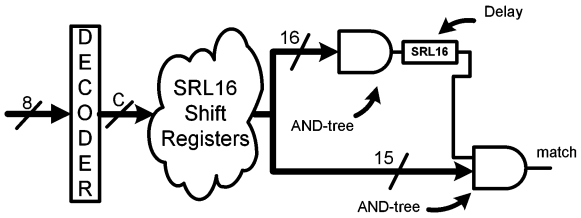

Fig. 3. DpCAM: Partial matching of long patterns. In this example, a 31-byte pattern is matched. The first 16 bytes are partially matched and the result is properly delayed to feed the second substring comparator. Both substring comparators are fed from the same pool of shifted decoded characters (SRL16s) and therefore sharing of decoded characters is higher.

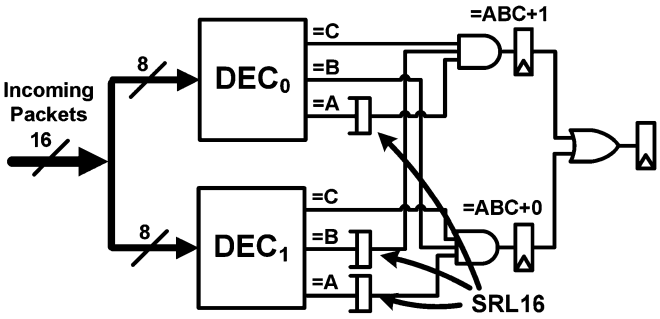

Fig. 4. DpCAM processing two characters per cycle.

split long patterns in smaller substrings and match each substring separately. This improved version of DCAM is denoted as *DpCAM* (decoded partial CAM). In doing so, instead of delaying the decoded data for a large number of cycles, we only need to delay the partial match signal. Fig. 3 depicts the block diagram of matching patterns longer than 16 characters. Long patterns are partially matched in substrings of maximally 16 characters long. The reason is that the AND-tree of a 16 character substring needs only five LUTs, while only a single SRL16 shift register is required to delay each decoded input character. Consequently, a pattern longer than 16 characters is partitioned in smaller substrings which are matched separately. The partial match of each substring is properly delayed and provides input to the AND-tree of the next substring. This way all the substring comparators need decoded characters delayed for no more than 15 cycles.

In order to achieve better performance, we use techniques to improve the operating frequency, as well as the throughput of our DpCAM implementation. To increase the processing throughput, we use parallelism. We widen the distribution paths by a factor of P providing P copies of comparators (decoders) and the corresponding matching gates. Fig. 4 illustrates this point for $P = 2$. To achieve high operating frequency, we use extensive fine-grain pipelining. The latency of the pipeline depends on the pattern length and in practice is a few tens of cycles, which translates to a few hundreds of nanoseconds and is acceptable for such systems.

In the DpCAM implementation, we also utilize a partitioning technique to achieve better performance and area density. In terms of performance, a limiting factor to the scaling of an implementation to a large number of search patterns is the fan-out and the length of the interconnections. If we partition the entire set of search patterns in smaller groups, we can implement the entire fan-out-decode-match logic for each of these groups in a significantly smaller area, reducing the average length of the wires. This reduction in the wire length though comes at the cost of multiple decoders. Each character must be decoded once in each of the groups it appears, increasing the cost. On the other hand, smaller groups may require smaller decoders, if the number of distinct characters in the group is low. Hence, if we group together search patterns with more similarities, we can reclaim some of the multidecoder overhead. We have implemented a simple, greedy algorithm that partitions iteratively the set of search patterns [8]. Additionally, partitioning reduces the implementation time of the design (synthesis, place and route) compared to a "flat" implementation flow. In the case of incremental design changes (e.g., new rules are added), the generation of the updated design bitstream is substantially faster when following an incremental design flow (using prespecified guide-files) compared to performing a new implementation of the entire design. Finally, partial reconfiguration is also feasible, when only a few blocks of the design need to be updated.

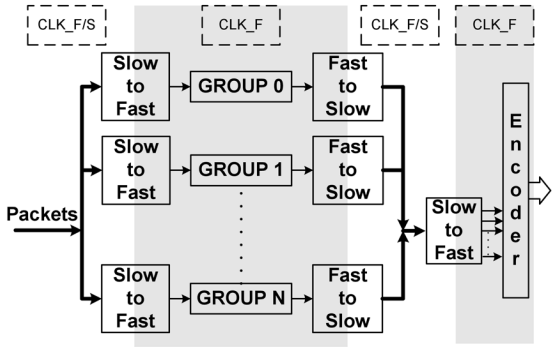

Fig. 5. DpCAM with multiple clock domains. Long, wide but slower busses (depicted with thick lines) distribute input data over large distances to the multiple search matching groups. These groups operate at higher clock rates to produce results faster.

In the partitioned design, the multiple groups will be fed data through a fan-out tree and all the individual matching results will be combined to produce the final matching output. Each partition is relatively small and hence can operate at a high frequency. However, for large designs, the fan-out of the input stream must traverse long distances. In our designs, we have found that these long wires limit the frequency for the entire design. To tackle this bottleneck, as depicted in Fig. 5, we use multiple clocks: one slow clock to distribute the data across long distances over wide busses and a fast clock for the smaller and faster partitioned matching function.

IV. Perfect Hashing Memory (PHmem)

The alternative pattern matching approach proposed in this paper is the PHmem. 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 known for decades. We introduce a perfect hashing algorithm and extend previous hardware hashing solutions for pattern matching based on two approaches proposed in the early 1980s. The first one used unique pattern prefixes matching to access a memory and retrieve the remaining search patterns [28] (also later used by Cho *et al.* in [7] and [11]), while the second showed

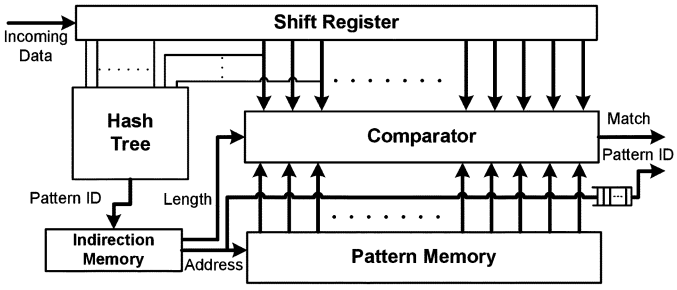

Fig. 6. PHmem block diagram.

that a hash function for a set of items can be composed by several subhashes of its subsets [29]. Fig. 6 depicts our 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. Selected bit positions of the shifted incoming data are used as input to a hash module, which outputs the ID of the "possible match" pattern. For memory utilization reasons (see Section IV-C), 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 [24]. The indirection memory outputs the actual location of the pattern in the pattern memory and its length that is used to determine which bytes of the pattern memory and the incoming data are needed to be compared. In our case, the indirection memory performs a *1-to-1* instead of the *N-to-1* mapping in [24], since the output address has the same width (number of bits) as the pattern ID. Finally, it is worth noting that the implementation of the hash tree and the memories are pipelined. Consequently, the incoming bitstream must be buffered by the same amount of pipeline stages in order to correctly align it for comparison with the chosen pattern from the pattern memory. This alignment is implicitly performed by the shift register and in this manner we can perform one comparison in each cycle.

A. Perfect Hashing Tree

The proposed scheme requires the hash function to generate a different address for each pattern, in other words, requires a *perfect* hash function which has no collisions for a given set of patterns. Furthermore, the address space would preferably be *minimal* and equal to the number of patterns. Instead of matching unique pattern prefixes as in [28], we hash unique substrings in order to distinguish the patterns. To do so, we introduce a perfect hashing method to guarantee that no collisions will occur for a given set.

Generating such a perfect hash function may be difficult and time consuming. In our approach, instead of searching for a single hash function, we search for multiple simpler subhashes that when put together in a tree-like structure will construct a perfect hash function. The perfect hash tree, is created based on the idea of "divide and conquer." Let A be a set of unique\ substrings $= \{a_1, a_2, \ldots, a_n\}$ and $\mathbf{H}(A)$ a perfect hash function of A, then the perfect hash tree is created according to the following equations:

$$\mathbf{H}(A) = \mathbf{h_0}(\mathbf{H_1}(\text{1st half of } A), \mathbf{H_2}(\text{2nd half of } A)) \tag{1}$$

$$\mathbf{H_1}(\text{1st half of} A) = \mathbf{h_1}(\mathbf{H_{1.1}}(\text{1st quarter of } A), \mathbf{H_{1.2}}(\text{2nd quarter of } A)) \tag{2}$$

and so on for the smaller subsets of the set A (until each subset contains a single element). The $\mathbf{h_0}, \mathbf{h_1}$, etc., are functions that combine subhashes. The $\mathbf{H_1}, \mathbf{H_2}, \mathbf{H_{1.1}}, \mathbf{H_{1.2}}$, etc., are perfect hashes of subsets (subhashes).

Following the previously discussed methodology, we create a binary hash tree. For a given set of n patterns that have unique substrings, we consider the set of substrings as an $n \times m$ matrix A. Each row of the matrix A (m bits long) represents a substring, which differs at least in one bit from all the other rows. Each column of the matrix A (n bits long) represents a different bit position of the substrings. The perfect hash tree should have $\log_2(n)$ output bits in order to be minimal. We construct the tree by recursively partitioning the given matrix as follows.

- Search for a function (e.g., $\mathbf{h_0}$) that separates the matrix A in two parts (e.g., A_0, A_1), which can be encoded in $\log_2(n) - 1$ bits each (using the `SUB_HASH` described in Section IV-B).
- Recursively repeat the procedure for each part of the matrix, in order to separate them again in smaller parts (performed by `HASH_TREE` of Table I).
- The process terminates when all parts contain one row.

Table I depicts the formal description of the `HASH_TREE`. In order to generate the hash tree, the `HASH_TREE` process recursively splits the given set of items in two subsets. The number of items that such two subsets may contain is an integer value that belongs to `SubsetSize`. `SubsetSize` is calculated by the `HASH_TREE` so that each subset can be encoded in $\log_2(n) - 1$ bits. To split a given set in two, a basic function $\mathbf{h_i}$ is used, generated by the `SUB_HASH` as described in Section IV-B.

Fig. 7(a) depicts the hardware implementation of the binary hash tree using 2-to-1 multiplexers for each tree node. In general, a tree node splits a given n-element set S in two parts and is represented by a 2-to-1 multiplexer ($\log_2(n) - 1$ bits wide). The select bit of the multiplexer is the function $\mathbf{h_i}$ generated by `SUB_HASH` and selects one of the two encoded parts of the set $\mathbf{H_1}(S_1), \mathbf{H_2}(S_0)$. The node output $\mathbf{H}(S)$ ($\log_2(n)$ bits wide) consists of the multiplexer output and the select bit of the multiplexer (MSbit). A leaf node of the hash tree separates three or four elements and consists of a 1-bit 2-to-1 multiplexer and its select bit. Each input of a leaf multiplexer is a single bit that separates two elements. To exemplify the hardware representation of the algorithm consider the following: the hash function $\mathbf{H}(A)$ of (1) is the output of the entire binary tree of Fig. 7(a) (k-bits) created by the concatenation of $\mathbf{h_0}$ and the output of the root multiplexer. The $\mathbf{h_0}$ is also used to choose between the inputs of the root multiplexer $(\mathbf{H_1}, \mathbf{H_2})$ which encode the two halves of A. Similarly, we can associate (2) with the second node of the binary tree, and so on.

The binary perfect hash tree can separate a set of patterns; however, we can optimize it and further reduce its area. In a single search for a select bit, we can find more than one select bits (in practice 2–5 bits) that can be used together to divide the set into more than two parts (4 to 32). The block diagram of our optimized hash tree is illustrated in Fig. 7(b). Each node of the tree can have more than two branches and, therefore, the tree is more compact and area efficient.

TABLE I
PERFECT HASHING ALGORITHM

constant Threshold $\in [1, m]$

H = HASH_TREE (A){
  SubsetSize=$\{x \in \mathbb{N}, |A| - 2^{\lceil \log_2(\frac{|A|}{2}) \rceil} \leq x \leq 2^{\lceil \log_2(\frac{|A|}{2}) \rceil}\}$

  h = SUB_HASH (A, SubsetSize)
  // $\mathbb{h}_{|A_1} : A_1 \mapsto \{1\} \subsetneq \{0,1\}$
  // $\mathbb{h}_{|A_0} : A_0 \mapsto \{0\} \subsetneq \{0,1\}$ where $A_1 \cup A_0 = A$ and $A_1 \cap A_0 = \emptyset$
  // $|A_0|, |A_1| \in$ SubsetSize

  IF($|A_1| > 1$)
    $\mathbf{H_1}$=**HASH_TREE** (A_1)

  IF($|A_0| > 1$)
    $\mathbf{H_2}$=**HASH_TREE** (A_0)

  RETURN($\mathbf{h} \circ (\mathbf{h} * \mathbf{H_1} + \overline{\mathbf{h}} * \mathbf{H_2})$)
}

h = SUB_HASH (A, SubsetSize){
  distance=$|A|$

  FOR k= 1 **to** Threshold {
    Find any k-**input XOR** function f of the bit columns {1,..,m},
    of A, such that $f : A \rightarrow \{0,1\}$, where $f_{|A_1} : A_1 \mapsto \{1\} \subsetneq \{0,1\}$,
    $f_{|A_0} : A_0 \mapsto \{0\} \subsetneq \{0,1\}$, with $A_1 \cup A_0 = A$, $A_1 \cap A_0 = \emptyset$

    IF($|A_1| \in$SubsetSize){
      $F = f$
      break
      // Instead of **break**, the algorithm can be modified to continue and
      // afterwards choose the simpler function among the suitable ones.
    }
    ELSE IF($\forall x \in$ SubsetSize, $min(|x - |A_1||) \leq$ distance){
      distance = $min(|x - |A_1||)$
      $F = f$
    }
  }

  IF($|A_1| \in$ SubsetSize){
    RETURN(F)
  }
  ELSE IF($|A_1| > max($SubsetSize$)$){
    new_A=A_1, where $F_{|A_1} : A_1 \mapsto \{1\} \subsetneq \{0,1\}$
    RETURN(F*SUB_HASH (new_A,SubsetSize))
  }
  ELSE IF($|A_1| < min($SubsetSize$)$){
    new_A=A_0, where $F_{|A_0} : A_0 \mapsto \{0\} \subsetneq \{0,1\}$
    newSubsetSize= $\{y \in \mathbb{N}, \forall x \in$SubsetSize$, y = x - |A_1|\}$
    where A_1 is $F_{|A_1} : A_1 \mapsto \{1\} \subsetneq \{0,1\}$
    RETURN(F+SUB_HASH (new_A,newSubsetSize))
  }
}

To prove that our method generates perfect hash functions, we need to prove the following.

1) For any given set A of n items that can be encoded in $\lceil \log_2(n) \rceil$ bits, our method generates a function $\mathbf{h} : A \rightarrow \{0,1\}$ to split the set in two subsets that can be encoded in $\lceil \log_2(n/2) \rceil$ bits (that is $\log_2(n) - 1$ bits).
2) Based one the first proof, the proposed scheme outputs a perfect hash function for the initial set of patterns.

In Section IV-B, we prove the first point and show that our algorithm guarantees the generation of a function $\mathbf{h} : A \rightarrow \{0,1\}$ for any given set A. We prove next that such functions when used in the tree-like structure will construct a perfect hash function.

Proof: By definition, a hash function $\mathbf{H}_{|A}$ of set $A = \{a_1, a_2, \ldots, a_x\}$ which outputs a different value for each element a_i is *perfect*

$$\mathbf{H}_{|a_1} \neq \mathbf{H}_{|a_2} \neq \cdots \neq \mathbf{H}_{|a_x}. \tag{3}$$

Also, if $\mathbf{h}_{|S}$, where $S = A \cup B \cup \cdots \cup N$ and $A \cap B \cap \cdots \cap N = \vee$ is a hash function that separates the n subsets $A, B, \ldots, N$ having a different output for elements of different subsets is also *perfect*, that is

$$\mathbf{h}_{|A} \neq \mathbf{h}_{|B} \neq \cdots \neq \mathbf{h}_{|N}. \tag{4}$$

We construct our hash trees based on two facts. First, the "selects" of the multiplexers $\mathbf{h}$ separate perfectly the subsets of the node; Section IV-B shows that our method generates a function $\mathbf{h}$ for any given set. Second, that the inputs of the leaf nodes are perfect hash functions; this is given by the fact that each element differs to any other element at least one bit, therefore, there exists a single bit that separates (perfectly) any pair of elements in the set. Consequently, it must be proven that a node which combines the outputs of perfect hash functions $\mathbf{H}_A, \mathbf{H}_B, \ldots, \mathbf{H}_N$ of the subsets $A, B, \ldots, N$ using a perfect hash function $\mathbf{h}_{|S}$ which separates these subsets, outputs also a perfect hash function $\mathbf{H_{node}}$ for the entire set S.[1]

The output $\mathbf{H_{node}}$ of the node is the following:[2]

$$\begin{aligned}\mathbf{H_{node}} = \mathbf{h}_{|S} \circ^2 \mathbf{IF}(\mathbf{h}_{|S} = \mathbf{h}_{|A})\ \mathbf{THEN}\ \mathbf{H}_A\ \mathbf{ELSE}\\ \mathbf{IF}(\mathbf{h}_{|S} = \mathbf{h}_{|B})\ \mathbf{THEN}\ \mathbf{H}_B\ \mathbf{ELSE}\\ \ldots\\ \mathbf{IF}(\mathbf{h}_{|S} = \mathbf{h}_{|N})\ \mathbf{THEN}\ \mathbf{H}_N.\end{aligned}$$

Consequently, the $\mathbf{H_{node}}$ outputs different values for either two entries of the same subset $\mathbf{H_{node}}_{|a_i} \neq \mathbf{H_{node}}_{|a_j}$ based on (3), or for two entries of different subsets $\mathbf{H_{node}}_{|a_i} \neq \mathbf{H_{node}}_{|b_j}$ based on (4). Therefore, each tree node and also the entire hash tree output perfect hash functions. ■

B. PHmem Basic Building Function

There is more than one function $\mathbf{h} : A \rightarrow \{0,1\}$ that can split a given set A of n items (m bits long) in two parts A_0 and A_1 which can be encoded in $\lceil \log_2(n/2) \rceil$ bits (where $A_1 \cup A_0 = A, A_1 \cap A_0 = \oslash$). The number of such functions is equal to the combination $\binom{n}{t}$ of selecting t items out of n, where n is the number of items and $t \in$ SubsetSize. That is due to the fact that any function $f : A \rightarrow \{0,1\}$ which selects any t items out of the n satisfies the previous condition.[3] For instance, when the number of items n is a power of two, there exist $\binom{n}{\frac{n}{2}} = {}^nC_{\frac{n}{2}} = \frac{n!}{\frac{n}{2}!\frac{n}{2}!}$ functions (e.g., ${}^{128}C_{64} \simeq 2.4 \times 10^{37}$). Moreover, all the possible input values (m bits long) that do not belong to the set A are "don't care" terms since they do not change the effectiveness of the function. This wide range of functions suitable for our algorithm and the large number

[1]Assuming n subsets $A, B, \ldots, N$ of x elements each, then $\mathbf{H}_A, \mathbf{H}_B, \ldots, \mathbf{H}_N$ output $\lceil \log_2(x) \rceil$ bits each, $\mathbf{h}_{|S}$ outputs $\lceil \log_2(n) \rceil$ bits, and $\mathbf{H_{node}}$ outputs $\lceil \log_2(n) \rceil + \lceil \log_2(x) \rceil$ bits.

[2]Where "$\circ$" is the concatenation operator.

[3]The condition for a basic building function is the number of items of each subset to belong to the SubsetSize, $|A_0|, |A_1| \in$ SubsetSize.

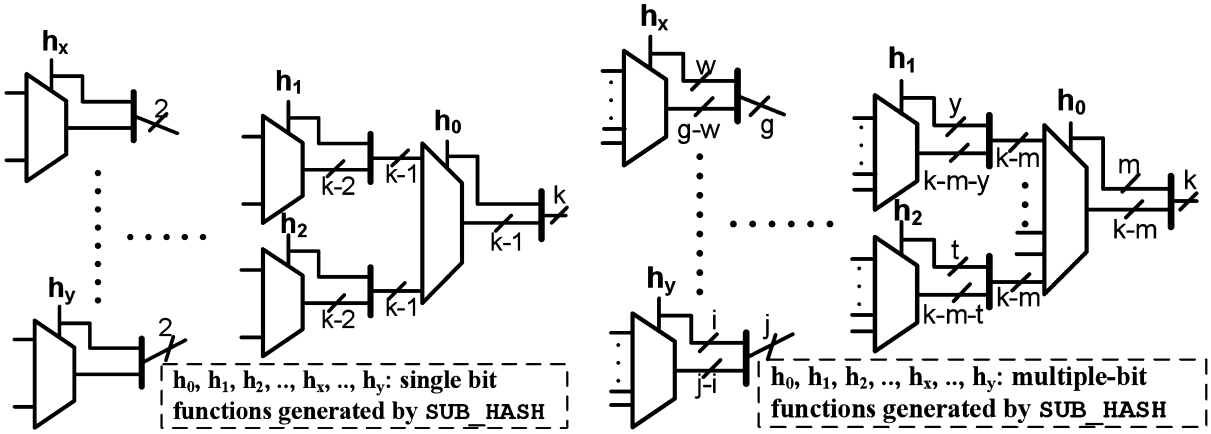

Fig. 7. Perfect hash trees: the select of each multiplexer is a function generated by the SUB_HASH. (a) Binary hash tree. (b) Optimized hash tree.

of "don't care" terms leave room to find a simple function that meets our requirements.

There are several algorithms to minimize the cost of a logic function such as the ones using exclusive-or sum of products (ESOP) representations [30], [31], that are especially effective in incompletely specified functions (having "don't care" terms). Although these algorithms can be used in our case, they require to explicitly specify the output of the function for the input terms of interest, limiting the alternative solutions to only a single function out of ${}^nC_t$. In our case, it is required to split the given set of items using *any* of the ${}^nC_t$ functions rather than specifying a single one.

We propose a new method (SUB_HASH) described in Table I, to find a function $\mathbf{h} : A \rightarrow \{0,1\}$ to separate a given set of items A ($n \times m$ matrix, $|A| = n$) into two subsets A_1 and A_0 which can be encoded in $\lceil \log_2(n/2) \rceil$ bits each. For simplicity, we assume for any function $f : A \rightarrow \{0,1\}$ that $|A_1| \geq |A_0|$, where $f_{|A_1} : A_1 \mapsto \{1\} \subsetneq \{0,1\}$ and $f_{|A_0} : A_0 \mapsto \{0\} \subsetneq \{0,1\}$. Otherwise we can use the inverted f, $\bar{f}$.

Starting from a given set A and the SubsetSize specified by HASH_TREE, the SUB_HASH exhaustively searches whether any k-input XOR function satisfies the condition.[4] Variable k is assigned values between "1" and Threshold, where Threshold can be a value between "1" and the length of the items m (bits). The greater the Threshold, the more computationally intensive the for-loop and on the other hand the more XOR functions are checked. In case a function satisfies the condition then the process is terminated and the identified function is returned. Otherwise, the function that produces subsets closer to the SubsetSize is picked and a new SUB_HASH iteration starts. In case the found function F outputs "1" for more than SubsetSize items of the specified set, then the following is performed: the new set is the subset of items for which the F outputs "1", the SubsetSize remains the same and the returned function is the product of F and the result of the new SUB_HASH call. When F outputs "1" for less than SubsetSize items, the new set is the subset for which F outputs "0", while the new SubsetSize consists of the elements in SubsetSize each one subtracted by $|A_1|$.[4] In this case, the returned value is the sum of F and the function returned by the new SUB_HASH call. Fig. 8 depicts an example of a set split in two using SUB_HASH. The process requires four iterations before it meets the condition, while the last intermediate function needs to be inverted in order to meet the condition. In summary, when the condition is not met then the same process is repeated to a subset of the set as specified before. The subset is smaller than the set of the previous iteration by at least one item. That is due to the definition that every item differs in at least one bit position compared to any other item, and consequently, there exists a single-bit input function which outputs for at least one item of the set a different value $(\bar{v})$ compared to the rest of the items (v). This way it is guaranteed that the recursive process will terminate with a solution in a finite $\left(n - 2^{\lceil \log_2(n/2) \rceil}\right)$ number of steps. In practice, all the results obtained in this paper required a single iteration of SUB_HASH having a Threshold $= 4$.

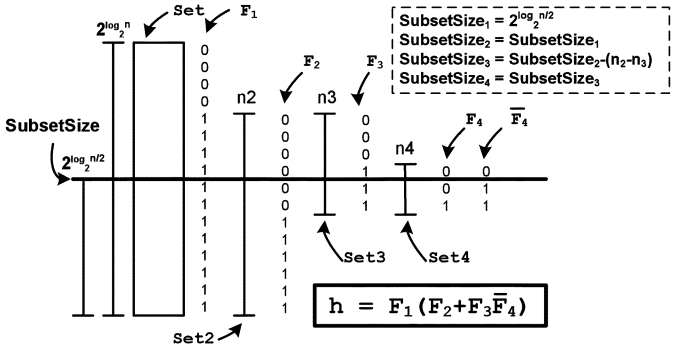

Fig. 8. Example of using SUB_HASH to split a Set in two subsets which require one bit less to be encoded compared to the set. Note that each F function (e.g., F_1, F_2, F_3, and F_4) is the result of a SUB_HASH call and consists of either a single column selection or an XOR of multiple columns. For presentation reasons, the items of the set are sorted so that F outputs "0" for the upper items of the set and "1" for the rest, however, any function that outputs equal number of "0's" (and "1's") with the F of the example would have the same functionality.

C. Pattern Preprocessing and Implementation Details

To generate a PHmem design for a given set of IDS patterns, we first extract the patterns from the Snort ruleset and group them so that patterns of each group have a unique substring. We then reduce the length of the substrings, keeping only the bit-positions that are necessary to distinguish the patterns (in practice

[4] $|A_1|$ is the number of A items for which F outputs "1".

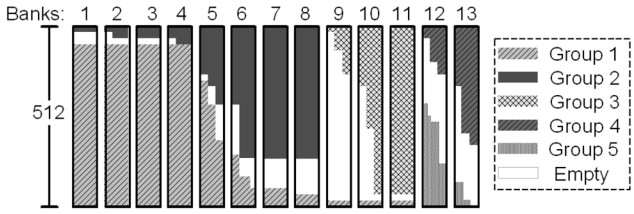

Fig. 9. Example of storing patterns in the pattern memory. There are five groups of patterns, distributed in the pattern memory such that each memory bank (block RAM) contains patterns of one or two groups.

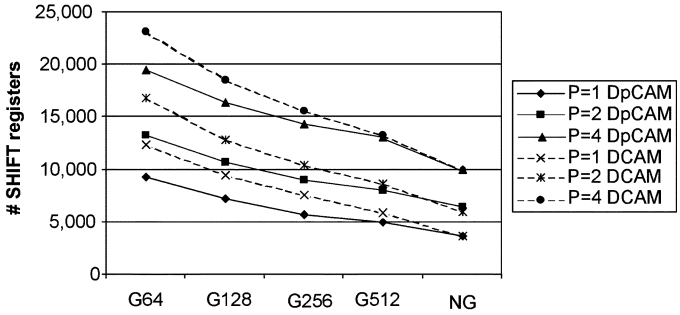

Fig. 10. Required number of shift registers (SRL16) in DpCAM and DCAM designs, for different partitioning and degree of parallelism. Fewer shift registers indicate higher sharing.

11–26 bits). Finally, we generate the hash trees for every reduced substring file.

We store the search patterns in the widest Xilinx dual-port block RAM configuration (512 entries $\times$ 36 bits), a choice that limits group size to a maximum of 512 patterns. Patterns of the same group should have unique substrings (for simplicity prefixes or suffixes) in order to distinguish them using hashing. The grouping algorithm takes into account the length of the patterns, so that 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. Fig. 9 depicts an example of the way patterns are stored in the pattern memory. In our designs, the memory utilization is about 60%–70%. We use an indirection memory to decouple the patterns ordering of a perfect hashing function from their actual placement in the memory; this flexibility allows us to store patterns sorted by length and increase memory utilization.

In order to achieve better performance, we use several techniques to improve the operating frequency and throughput of our designs. We increase the operating frequency of the logic using extensively fine-grain pipelining in the hash trees and the comparator. The memory blocks are also limiting the operating frequency so we generate designs that duplicate the memory and allow it to operate at half the frequency of the rest of the design. To increase the throughput of our designs, we exploit parallelism. We can widen the distribution paths by a factor of 2 by providing two copies of comparators and adapting the procedure of hash tree generation. More precisely, in order to process two incoming bytes per cycle, we first replicate the comparators such that each one of them compares memory patterns against incoming data in two different offsets (0 and 1 byte offsets), and their match signals are ORed. Furthermore, each substring file should contain two substrings of every pattern in offsets 0 and 1 byte. These substrings can be identical, since they point out the same pattern, but they should be different compared to the rest of the substrings that exist in the file. This restriction makes grouping patterns more difficult resulting in a potentially larger number of groups (in practice, 9–11 groups instead of 8). The main advantage of this approach is that using parallelism does not increase the size of the pattern memory. Each pattern is stored only once in the memory, and it is compared against incoming data in two different offsets.

V. Evaluation and Comparison

In this section, we first present our implementation results of the DpCAM and PHmem structures. We then evaluate the efficiency of our designs, investigate the effectiveness of utilizing memory blocks and/or DpCAM for pattern matching, and finally, compare our designs against related works.

A. Evaluation

We implemented both DpCAM and PHmem using the rules of the Snort open-source intrusion detection system [15]. Snort v2.3.2 has 2188 unique patterns of 1–122 characters long, and 33 618 characters in total. We implemented our designs using Virtex2 devices with -6 speed grade, except DpCAM designs that process 4 bytes/cycle, which were implemented in a Virtex2-8000-5, the only available speed grade for the largest Virtex2 device. We measure our pattern matching designs performance in terms of processing throughput (gigabits per second), and their area cost in terms of number of logic cells required for each matching character. For designs that require memory, we measure the memory area cost based on the observation that 12 bytes of memory occupy area similar to a logic cell [32]. In order to evaluate our schemes and compare them with the related research, we introduce a new performance efficiency metric (nPEM) which takes into account both performance and area cost described by the following equation:

$$\mathrm{nPEM} = \frac{\text{Performance}}{\text{Area}} = \frac{\text{Throughput}}{\frac{\text{Logic Cells} + \frac{\text{MEMbytes}}{12}}{\text{Characters}}}. \quad (5)$$

1) DpCAM Evaluation: To evaluate DpCAM and compare it to DCAM, we implemented designs that process 1, 2, and 4 bytes/cycle (P = 1, 2, and 4) with different partition sizes: partitions of 64, 128, 256, 512 patterns (G64, G128, G256, G512) and designs without partitioning (NG).

DpCAM Versus DCAM: DpCAM targets the increased sharing of decoded characters, while the rest of the design is similar to DCAM. We can estimate the number of distinct shifted characters needed for each design by counting the number of SRL16 shift registers, with fewer SRL16s indicating better character sharing. Fig. 10 plots the number of SRL16s used for DCAM and DpCAM designs of several different datapath widths and partition sizes. DpCAM needs up to 25% less SRL16s and up to 15% less total area, while the improvement is higher in the case of small partition sizes where sharing

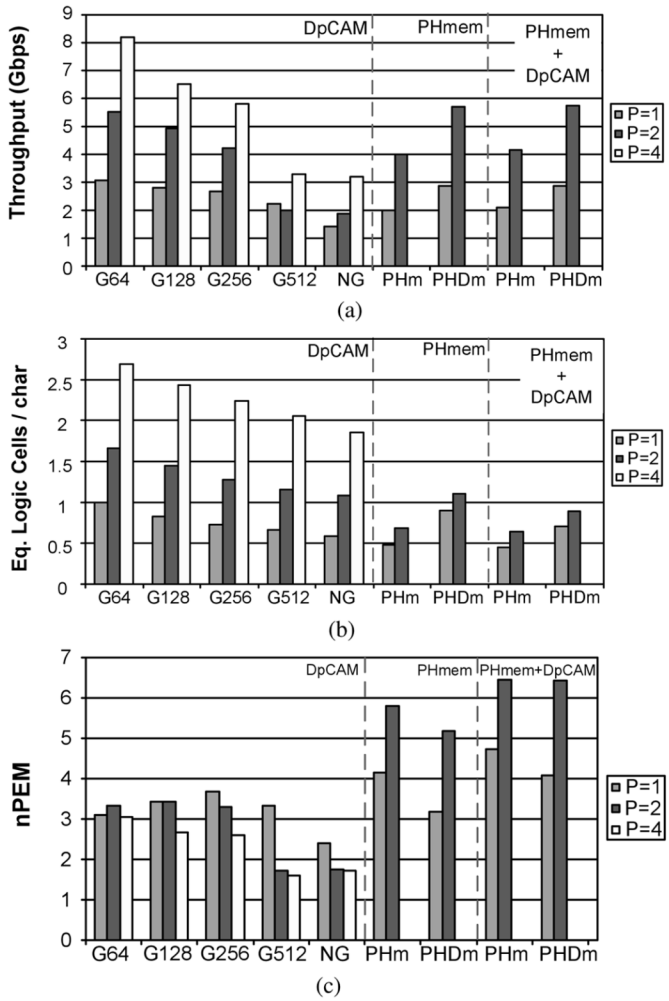

Fig. 11. PHmem and DpCAM performance, area cost, and efficiency. Memory area is calculated based on the following equation: $12 \times \mathrm{MEMbytes} =$ Logic Cell. (a) DpCAM and PHmem performance. (b) DpCAM and PHmem area cost. (c) DpCAM and PHmem efficiency.

possibilities are limited. The frequency in both cases is similar and consequently DpCAM is more efficient than DCAM.

DpCAM Results: Fig. 11(a) illustrates the performance in terms of processing throughput for the five partition sizes and datapath width of 1, 2, and 4 bytes/cycle. Designs that process 1 byte/cycle achieve 1.4 to 3 Gb/s throughput, while designs that process 2 bytes/cycle can support 1.9 to 5.5 Gb/s. Furthermore, designs with 4 bytes datapaths have a processing throughput between 3.2 to 8.2, implemented on a -5 speed grade Virtex2 device as there are no -6 grade devices large enough to fit them. From our results, we can draw two general trends for group size. The first is that smaller group sizes lead to higher throughput. The second is that when the group size approaches 512 the performance deteriorates, indicating that optimal group sizes will be in the 64-256 range.

We measured area cost and plot the number of logic cells needed for each pattern character in Fig. 11(b). Unlike performance, the effect of group size on the area cost is more pronounced. As expected, larger group sizes result in smaller area cost due to the smaller replication of comparators in the different groups. In all, the area cost for the entire Snort rule set is 0.58–0.99, 1.1–1.6, and 1.8–2.7 logic cells/character for designs that process 1, 2, and 4 bytes/cycle, respectively.

While smaller group sizes offer the best performance, it appears that if we also take into account the area cost, the medium group sizes (128 or 256) become also attractive. This conclusion is more clear in Fig. 11(c) where we evaluate the efficiency of our designs (*Performance/Area Cost*). For $P = 1$ the most efficient design is *G256*, for *P=2* is *G64*, *G128*, and *G256* groupings have similar efficiency, while for $P = 4$ where the designs are larger and thus more complicated the best *performance/area* tradeoff is in *G64*.

2) PHmem Evaluation: Fig. 11(a)–(c) illustrates the performance, area cost, and efficiency of perfect hashing designs. We implemented designs that process 1 and 2 incoming bytes/cycle ($\mathrm{P} = 1$ and 2). Apart from the designs that operate in a single clock domain (denoted as PHm), there are designs with double memory size (denoted as PHDm) that operates in half the operating frequency relative to the rest of the circuit. Our perfect hashing design that processes one byte/clock cycle achieves 2 Gb/s of throughput, using 35 block RAMs (630 kb), and requiring 0.48 *equivalent* logic cells (ELC) per matching character (counting also the area due to the memory blocks). A design that utilizes double memory to increase the overall performance of the pattern matching module achieves about 2.9 Gb/s, requiring 0.9 ELCs per matching character. The design that processes 2 bytes/cycle achieves 4 Gb/s, while needing 0.69 ELCs per character. When using double size of memory and process 2 bytes/cycle, PHmem can support more than 5.7 Gb/s of throughput, requiring 1.1 ELCs per matching character. It is noteworthy that about 30%–50% of the required logic is due to the registered memory inputs and outputs and the shift registers of incoming data. Matching subpatterns of constant length and then merging the partial results as implemented in [24], would possibly decrease this area cost.

3) PHmem + DpCAM: PHmem designs present a significant disadvantage when the pattern set includes very long patterns. The pattern memory in this case should be wide enough to fit these long patterns, resulting in low memory utilization. Consequently, we implemented designs that use PHmem for matching patterns up to 50 characters long and DpCAM for longer patterns. These designs, as Fig. 11 illustrates, have similar performance with the original PHmem designs and lower area cost, leading to an increase of the performance efficiency metric by 10%–25%.

B. Memory-Logic Tradeoff

DpCAM and PHmem offer a different balance in the resources used for pattern matching. The decision of following one of the two approaches, or the combination of both, is related to the available resources of a specific device. In general, using a few tens of block RAMs is relatively inexpensive in recent FPGA devices, while running out of logic cells can be more critical for a system. Counting ELCs that include memory area gives an estimate of the designs area cost. By using this metric to measure area, we evaluate and compare the efficiency of PHmem and DpCAM designs. Fig. 11(c) illustrates the performance efficiency metric of DpCAM and PHmem. It is clear that the perfect hashing designs outperform DpCAM, since they require less area and maintain similar performance. PHmem designs with DpCAM for matching long patterns are

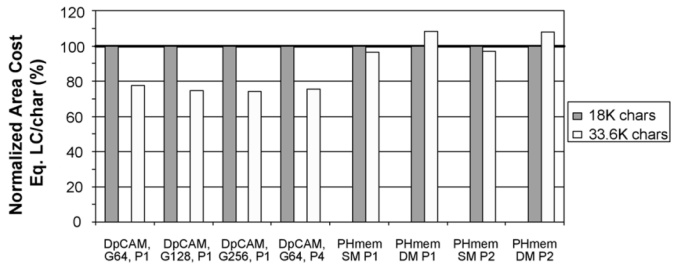

Fig. 12. Normalized area cost per matching character of different PHmem, DpCAM designs that match 18 and 33.6 K pattern characters. The values are normalized to the designs that match 18 K characters.

even more efficient. The reason is that they have a better pattern memory utilization (over 70%) and therefore require fewer resources. Designs that require more logic usually lead to more complicated implementation (synthesis, place and route, and wire distances) and require longer cycle times. Consequently, PHmem is simpler to synthesize compared to DpCAM, while the cycle time variation from one design to another is negligible. In summary, even though the nPEM (5) gives an estimate of which approach is more efficient, it is difficult to make a decision in advance, since the pattern matching module will be part of a larger system.

C. Scalability

An IDS pattern matching module needs to scale in terms of performance and area cost as the number of search patterns increases. That is essential since the IDS rulesets and the number of search patterns constantly grow. Two different pattern sets were used to evaluate the scalability of our designs. Apart from the one used in our previous results which contains about 33 K characters, an older Snort pattern set (18 K characters) was employed for this analysis. Both DpCAM and PHmem designs do not have significant performance variations as the pattern set grows from 18 to 33 K characters. Fig. 12 depicts how the area cost scales in terms of ELCs per character. Generally speaking, DpCAM area cost scales well as the pattern set almost doubles since character sharing is more efficient. DpCAM designs that match over 33 K characters require about 75% of the LC/char compared to the designs that match 18 K characters. On the other hand, PHmem shows some small variations in area cost primarily due to variations in the pattern memory utilization, however, in general the area cost is stable. Finally, both DCAM and PHmem have only up to 5% variations in throughput with the PHmem designs being more scalable due to their reduced area cost. In summary, both DCAM and PHmem scale well in terms of performance and resource utilization as the number of patterns increases, which is promising since the IDS pattern set grows rapidly.

D. Comparison

In Table II, we attempt a fair comparison with previously reported research on FPGA-based pattern matching designs that can store a full IDS ruleset. Table II contains our results as well as the results of the most efficient recent related approaches for exact pattern matching. Here the reader should be cautioned that some related works were implemented and evaluated on different FPGA families. Based on previous experience of implementing a single design in different device families [8], [19] and the Xilinx datasheets [27], we estimate that compared to Virtex2, Spartan3 is about 10%–20% slower, while Virtex2Pro is about 25%–30% faster. Fig. 13 illustrates the normalized nPEM of our designs and related work, taking into account the device used for the implementation. Note that the previous results intend to give a better estimate of different pattern matching designs since different device families achieve different performance results. Compared to related works, PHmem $(\mathrm{P} = 2)$ has at least 20% better efficiency. DpCAM has slightly lower or higher efficiency compared to the most of the related works, while PHmem+DCAM is at least 30% better.

Compared to Attig *et al.* Bloom Filters design [23], PHmem has better efficiency [19]. Bloom filters perform *approximate* pattern matching, since they allow false positives. Attig *et al.* proposed the elimination of false positives using external SDRAM which needs 20 cycles to verify a match. Since the operation of this SDRAM is not pipelined, the design's performance is not guaranteed under worst case traffic.

It is difficult to compare any FPGA-based approach against the "Bit-Split FSM" of Tan and Sherwood [12], which was implemented in ASIC 0.13-μm technology. Tan and Sherwood attempted to normalize the area cost of FPGA designs in order to compare them against their ASIC designs. Based on this normalization, our best PHmem design has similar and up to 5$\times$ lower efficiency compared to "bit-split FSM" designs, that is: 492 and 540 Gb/s/(char/mm^2) for PHmem and PHmem+DpCAM, compared to 556–2 699 Gb/s/(char/mm^2) for Bit-split. Despite the fact that our approach is less efficient than the previous ASIC implementation, there are several advantages to oppose. The implementation and fabrication of an ASIC is substantially more expensive than an FPGA-based solution. In addition, as part of a larger system, a pattern matching module should provide the flexibility to adapt on new specifications and requirements on demand. Such flexibility can be provided more effectively by reconfigurable hardware instead of an ASIC. Therefore, reconfigurable hardware is an attractive solution for IDS pattern matching providing flexibility, fast time to market, and low cost.

VI. Conclusion

We described and compared two reconfigurable pattern matching approaches, suitable for intrusion detection. The first one (DpCAM) uses only logic and the predecoding technique to share resources. The second one (PHmem) requires both memory and logic, employing an in practice simple and compact hash function to access the pattern memory. The proposed PHmem algorithm guarantees the generation of a perfect hash function for any given set of patterns. Both techniques were implemented in reconfigurable hardware and evaluated in terms of area and performance. We analyzed their tradeoffs and discussed their efficiency compared to related work. Utilizing memory turns out to be more efficient than using only logic, while the combination of PHmem and DpCAM produces the most efficient designs. PHmem and DpCAM are able to support up to 5.7 and 8.2 Gb/s throughput, respectively, in a Xilinx Virtex2 device. Our perfect hashing technique achieves about 20% better efficiency compared to other FPGA-based exact pattern matching approaches and when combined with the DpCAM for matching long patterns can be up to 30% better. Even compared to ASIC designs our approach has comparable

TABLE II
COMPARISON OF FPGA-BASED PATTERN MATCHING APPROACHES

Description	In bits /cycle	Device	Throughput (Gbps)	Logic Cells[5]	ELC/ char	MEM Kbits	#chars	nPEM
PHmem	8	Virtex2	2.000	9,466	0.48	630	33,618	4.15
			2.857[6]	16,852	0.90	1,260[6]		3.17
	16		4.000	15,672	0.68	702		5.81
			5.714[6]	22,114	1.10	1,404[6]		5.18
DpCAM	8		2.667	24,470	0.72	0		3.66
	16		4.943	48,678	1.44	0		3.41
	32		8.205	17,538	2.69	0		3.04
PHmem + DpCAM for long patterns	8		2.108	6,272	0.45	288	20,911	4.72
			2.886[6]	9,052	0.71	576[6]		4.06
	16		4.167	10,224	0.64	306		6.46
			5.734[6]	12,106	0.89	612[6]		6.44
CRC Hash [24]	8	Virtex2	2.000	2,570	0.50	630	18,636	4.00
	16		3.712	5,230	0.96	1,188		3.87
BDDs [25]	8	Virtex2	2.500	?	∼0.60	0	19,715	∼4.17
	48		∼12-14	?	∼3.60	0		3.33
NFAs [6]	32	Virtex2	7.004	54,890	3.10	0	17,537	2.26
RDL w/reuse [7]	8	Spartan3	2.000	?	0.81	0	20,800	2.46
ROM-based [11]			1.900	>8,000[7]	>0.47[7]	162		<4.06[7]
Unary [5]	8	Virtex2P	1.488	8,056	0.41	0	19,584	3.63
	32		4.507	30,020	1.53	0		2.94
Tree-based [20]	8		1.896	6,340	0.32	0		5.86
bit-split [26]		Virtex4	1.600	4,513	0.27	6,000	16,715	0.39

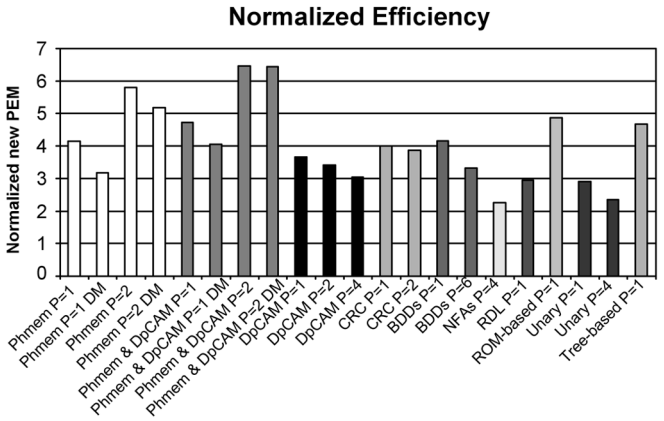

Fig. 13. Normalized nPEM of PHmem, DpCAM and related work. The performace of designs implemented in devices other than Virtex2 are normalized as follows. Spartan3: $\times$ 1.2; Virtex2Pro: $\div 1.25$[8], [19], [24], [27].

results. Both DpCAM and PHmem scale well in terms of performance and area cost as the IDS ruleset grows. Consequently, perfect hashing provides a high throughput and low area IDS pattern matching which can keep up with the increasing size of IDS rulesets, while DpCAM minimizes the cost of matching long patterns.

ACKNOWLEDGMENT

The authors would like to thank G. N. Gaydadjiev for his valuable comments on PHmem algorithm, C. Galuzzi for his help on putting in mathematical forms several statements, and C. Strydis for his comments which helped to improve the quality of this material.

REFERENCES

[1] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2002, pp. 111–120.

[2] Y. H. Cho, S. Navab, and W. Mangione-Smith, "Specialized hardware for deep network packet filtering," in *Proc. 12th Int. Conf. Field Program. Logic Appl.*, 2002, pp. 452–461.

[3] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards gigabit rate network intrusion detection technology," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2002, pp. 404–413.

[4] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2003, pp. 880–889.

[5] Z. K. Baker and V. K. Prasanna, "A methodology for synthesis of efficient intrusion detection systems on FPGAs," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 135–144.

[6] C. R. Clark and D. E. Schimmel, "Scalable parallel pattern-matching on high-speed networks," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 249–257.

[7] Y. H. Cho and W. H. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 125–134.

[8] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 258–267.

[9] Z. K. Baker and V. K. Prasanna, "Automatic synthesis of efficien intrusion detection systems on FPGAs," in *Proc. 14th Int. Conf. Field Program. Logic Appl.*, 2004, pp. 311–321.

[10] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of bloom filters for string matching," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 322–323.

[11] Y. H. Cho and W. H. Mangione-Smith, "Programmable hardware for deep packet filtering on a large signature set," in *Proc. Conf. Interaction Between Arch., Circuits, Compilers*$(\mathrm{P} = \mathrm{ac}2)$, 2004.

[12] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. 32nd Int. Symp. Comput. Arch. (ISCA)*, 2005, pp. 112–122.

[13] P. Krishnamurthy, J. Buhler, R. D. Chamberlain, M. A. Franklin, K. Gyang, and J. Lancaster, "Biosequence similarity search on the mercury system," in *Proc. 15th IEEE Int. Conf. Appl.-Specific Syst., Arch., Processors (ASAP)*, 2004, pp. 365–375.

[14] E. Sotiriadis, C. Kozanitis, and A. Dollas, "FPGA based architecture for DNA sequence comparison and database search," presented at the 13th Reconfigurable Arch. Workshop (RAW), Rodos, Greece, 2006.

[15] SNORT, "SNORT official website," 2007. [Online]. Available: http://www.snort.org

[16] M. Fisk and G. Varghese, "An analysis of fast string matching applied to content-based forwarding and intrusion detection," Univ. California, San Diego, Tech. Rep. CS2001-0670, 2002.

[17] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, and M. Polychronakis, "Performance analysis of content matching intrusion detection systems," in *Proc. Int. Symp. Appl. Internet*, 2004, pp. 208–218.
[18] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 227–238.
[19] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *Proc. 15th Int. Conf. Field Program. Logic Appl.*, 2005, pp. 644–647.
[20] Z. K. Baker and V. K. Prasanna, "Automatic synthesis of efficient intrusion detection systems on FPGAs," *IEEE Trans. Dependable Sec. Comput.*, vol. 3, no. 4, pp. 289–300, Oct. 2006.
[21] J. Singaraju, L. Bu, and J. A. Chandy, "A signature match processor architecture for network intrusion detection," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2005, pp. 235–242.
[22] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2003, pp. 31–38.
[23] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, Jan. 2004.
[24] G. Papadopoulos and D. Pnevmatikatos, "$\mathrm{Hashing} + \mathrm{Memory} = \mathrm{Low\ Cost}$, exact pattern matching," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2005, pp. 39–44.
[25] S. Yusuf and W. Luk, "Bitwise optimized CAM for network intrusion detection systems," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2005, pp. 444–449.
[26] H.-J. Jung, Z. K. Baker, and V. K. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," presented at the Reconfigurable Arch. Workshop IPDPS (RAW), Rodos, Greece, 2006.
[27] Xilinx, San Jose, CA, "VirtexE, Virtex2, Virtex2Pro, and Spartan3 datasheets," 2006. [Online]. Available: http://www.xilinx.com
[28] F. J. Burkowski, "A hardware hashing scheme in the design of a multiterm string comparator," *IEEE Trans. Comput.*, vol. 31, no. 9, pp. 825–834, Sep. 1982.
[29] R. C. Merkle, "Protocols for public key cryptosystems," in *Proc. IEEE Symp. Security Privacy*, 1980, pp. 122–134.
[30] N. Song and M. A. Perkowski, "Minimization of exclusive sum-of-products expressions for multiple-valued input, incompletely specified functions," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 15, no. 4, pp. 385–395, Apr. 1996.
[31] T. Kozlowski, E. L. Dagless, and J. Saul, "An enhanced algorithm for the minimization of exclusive-OR sum-of-products for incompletely specified functions," in *Proc. Int. Conf. Comput. Des.*, 1995, pp. 244–249.
[32] T. Sproull, G. Brebner, and C. Neely, "Mutable codesign for embedded protocol processing," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2005, pp. 51–56.

Ioannis Sourdis (S'06) was born in Corfu, Greece, in 1979. He received his Diploma and the M.S. degree in electronic and computer engineering from Technical University of Crete, Crete, Greece, in 2002 and 2004, respectively. He is currently pursuing the Ph.D. degree in computer engineering from the Delft University of Technology, Delft, The Netherlands.

His research interests include architecture and design of computer systems, multiprocessor parallel systems, interconnection networks, reconfigurable computing, network security, and networking systems.

Dionisios N. Pnevmatikatos received the B.S. degree in computer science from the University of Crete, Crete, Greece, and the M.Sc. and Ph.D. degrees in computer science from the University of Wisconsin-Madison, Madison.

He is an Associate Professor with the Department of Electrical and Computer Engineering, Technical University of Crete, Crete, Greece, and a Research Associate with the Institute of Computer Science, Foundation for Research and Technology-Hellas, Heraklion, Crete, Greece. His research interests include the architecture and design of computer and networking systems, and reconfigurable computing.

Prof. Pnevmatikatos is a member of the ACM.

Stamatis Vassiliadis (M'86–SM'92–F'97) was born in Manolates, Samos, Greece, in 1951. Regrettably, Prof. Vassiliadis deceased in April, 2007. He was a Chair Professor with the Electrical Engineering Department, Delft University of Technology (TU Delft), Delft, The Netherlands. He had also served with the Electrical Engineering faculties of Cornell University, Ithaca, NY, and the State University of New York (S.U.N.Y.), Binghamton, NY. He worked for a decade with IBM, where he had been involved in a number of advanced research and development projects.

Prof. Vassiliadis was a recipient of numerous awards including 24 publication awards, 15 invention awards, an Outstanding Innovation Award For Engineering/Scientific Hardware Design, an Honorable Mention Best Paper Award from the ACM/IEEE MICRO25 in 1992, Best Paper Awards in the IEEE CAS (1998, 2001), IEEE ICCD (2001), PDCS (2002), and the Best Poster Award from the IEEE NANO (2005). His 72 U.S. patents rank him as the top all time IBM inventor. He was an ACM fellow and a member of the Royal Dutch Academy of Science.