A Memory-Efficient Reconfigurable Aho-Corasick FSM Implementation for Intrusion Detection Systems

Vassilis Dimopoulos*, Ioannis Papaefstathiou* and Dionisios Pnevmatikatos†*

* Electronic and Computer Engineering Department,
Technical University of Crete
Chania, GR 73 100, Greece
Email: {dimopoulos, ygp, pnevmati}@mhl.tuc.gr

† Institute of Computer Science, FORTH – member of HiPEAC,
Vasilika Vouton, Heraklion, GR7110, Greece
Email: pnevmati@ics.forth.gr

***Abstract*—The Aho-Corasick (AC) algorithm is a very flexible and efficient but memory-hungry pattern matching algorithm that can scan the existence of a query string among multiple test strings looking at each character exactly once, making it one of the main options for software-base intrusion detection systems such as SNORT. We present the Split-AC algorithm, which is a reconfigurable variation of the AC algorithm that exploits domain-specific characteristics of Intrusion Detection to reduce considerably the FSM memory requirements. SplitAC achieves an overall reduction between 28-75% compared to the best proposed implementation.**

I. Introduction

The demand for high speed and always-on network access around the world is continuously increasing, creating an analogous demand for increased network security. Firewalls, i.e. security systems permitting or blocking packets based on their header information, have been a standard security solution for several years but are no longer adequate to cover the increased security needs. Network Intrusion Detection Systems (IDS) provide a powerful and versatile security tool by allowing us to specify header characteristics of "suspicious" packets as well as patterns contained in the payload that can constitute part of an attack. However, both the effectiveness and the resource requirements of a NIDS are largely dependent on the selected pattern matching algorithm.

The Aho-Corasick (AC) [1] algorithm is a very powerful but memory-hungry pattern matching algorithm whose characteristics make it an excellent choice for NIDS. The main contribution of this work is the "Split-AC" algorithm, which is a variation of the AC algorithm optimized for reconfigurable hardware implementation that requires between 28-75% less memory than the state-of-the-art AC implementation.

Software-based NIDS running on general purpose processors are very powerful and versatile but suffer from performance limitations. Proposed hardware-based NIDS can be very fast but usually ignore packet header parameters, a fact that causes them to perform searches when it is not necessary due to header mismatches. A secondary contribution is a hardware NIDS architecture which performs packet header classification, matches one pattern per rule, fits on a single FPGA chip and could cooperate with a software NIDS in order to significantly improve overall performance. While we are not the first to propose a complete system for intrusion detection we believe it is important to address the system aspects of the design in addition to the specifics of each sub-component.

This paper is organized in the following manner. Section II presents the Aho-Corasick (AC) pattern matching algorithm along with related work on memory efficient AC variations. In Section III we present Split-AC, our memory efficient variation of AC and the optimizations we used. Section IV provides a brief description of the IDS rules we use and the manner in which those rules are grouped into sets to improve memory efficiency. Section V presents the hardware architecture for Split-AC as well as for the overall IDS platform we have developed. Section VI is dedicated to the Split-AC implementation results and the comparison to related work. Finally, in section VII we present our conclusions as well as some ideas for future work.

II. The Aho-Corasick Algorithm and Related Work

The Aho-Corasick algorithm [1] was proposed in 1975 and remains, to this day, one of the most effective pattern matching algorithms when matching patterns sets. Initially, the AC algorithm combines all the patterns in a set into a syntax tree which is then converted into a non-deterministic automaton (NFA) and, finally, into a deterministic automaton (DFA). The resulting fsm is then used to process the text one character at a time, performing one state transition for every text character. Whenever the fsm reaches designated "final" states that correspond to the identification of a pattern a match

1-4244-1058-4/07/$25.00 ©2007 IEEE

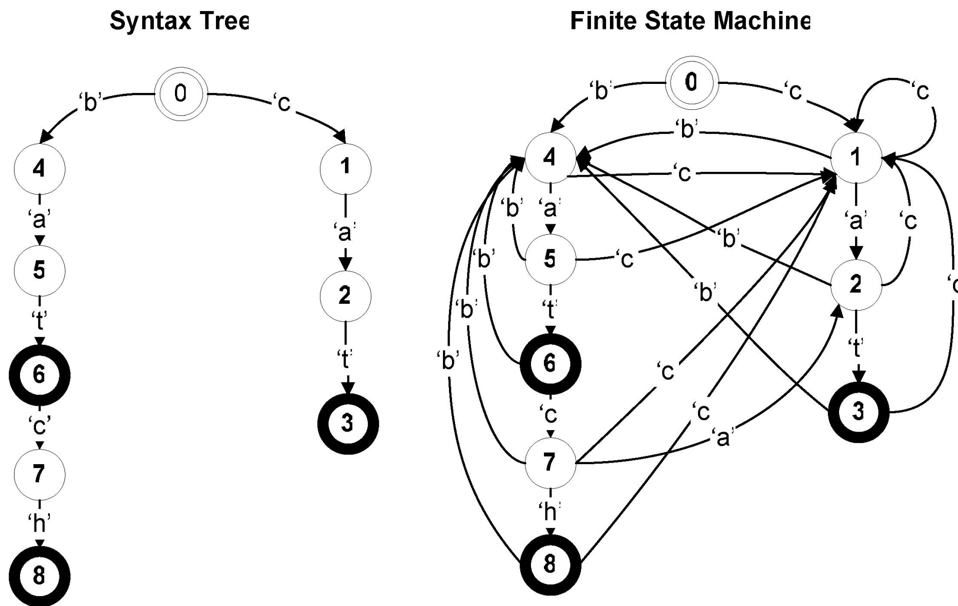

Fig. 1. An Aho-Corasick syntax tree (left) and the derived DFA (right). Numbered circles represent states (thick border for final states) while the arrows represent transitions for specific characters. For the sake of clarity, the fsm diagram omits transitions from each node leading back to the initial state, state 0, for all characters not already present in transitions from that node.

is found. A simple example of the AC fsm which corresponds to the patterns ”bat”, ”batch” and ”cat” is shown in figure 1.

The AC algorithm has the significant advantage that every text character is examined only once, i.e. the lookup cost is O(N) where N the length of the text, regardless of the number of patterns or their length. Other pattern matching algorithms, such as Boyer-Moore [2] or Wu-Manber [3], have an average lookup cost of O(N/n), where n is the length of the shortest pattern in the set. While these algorithms are significantly faster for sets with long patterns, in IDS it is quite common to have patterns which are only one or two characters in length, in which case AC is most appropriate.

The major disadvantage of AC is that it requires large amounts of memory in a straightforward implementation that keeps a lookup table of 256 pointers to next states for every fsm state. Recently, several more space-efficient variations of the AC algorithm have been proposed.

To reduce AC space requirements, Tuck et al. [4] use a bitmap and an array of transitions in each state instead of the traditional lookup table. When processing a text character c, bit c of the bitmap is checked: if the bit is set, we count the number of set bits in positions 0 to c-1 of the bitmap and the resulting number is used to address the transition array and find the next state. This reduces required space, but the popcount operation required is slow. He also proposes use path compression: for states having a dangling chain of states a single, multi-character comparison and transition is performed and the additional states in the chain are eliminated. However, this method significantly increases complexity and can only be applied to the Aho-Corasick NFA, not the DFA. The problem with the NFA is that it is more complex than the DFA and can perform up to 2N transitions for an N character text compared to the single, deterministic transition of a DFA.

Tan and Sherwood [5] propose a different approach: the initial fsm which performs 8-bit comparisons and has 256 possible transitions per state is broken down into 8 small fsms which run in parallel and each is responsible for a different character bit and has 2 possible transitions per state. The tradeoff is that additional control logic is required to check if all the small fsms have reached the same final state. Their architecture proves to be very effective, needing as little as 3.2 Mbits for the entire Snort ruleset and achieving throughput around 10 Gbps.

III. The Split-AC Algorithm

To optimize the AC implementation for IDS, we made several domain specific observations: (i) most patterns use only a small subset of the 256 possible characters, (ii) Some pattern characters are frequent and appear in a transition almost in every state while others appear infrequently, and (iii) if we partition the single, big FSM into smaller ones, the resulting FSMs have much smaller size (this actually bodes well with the IDS rules that are anyway expressed in compatible groups),

Split-AC exploits these observations: we use character translation in order to compress the 8-bit character input value into using fewer bits, thus reducing the state transition table size. To exploit the second observation, we use a combination of RAM and CAM. Frequently used transitions are kept in a regular table, while infrequent are kept in a CAM, while “default” transitions (i.e. return to the initial state) are merged into a single entry.

Split-AC starts the Aho-Corasick deterministic fsm with S states. We define as $f(c)$ the occurrence frequency of character c in transitions of this fsm. We also define the frequency

threshold Tf. Characters with $f(c) \geq Tf$ are called frequent. Accordingly, transitions made on frequent characters are called frequent transitions. Characters with $0 < f(c) < Tf$ are infrequent characters and valid transitions made for these characters are called infrequent transitions. Finally, characters with a zero occurrence frequency are non-existent characters.

Let K be the total number of frequent characters. We "compress" the input using a lookup table of 256 entries by $\lceil log_2(K+1) \rceil$ bits, rounded up. The "+1" in this formula is the additional "Not-in-This-Table" value we store for the infrequent and non-existent characters.

The number of bits needed to represent the state of the fsm is $\lceil s = log_2(S) \rceil$. For frequent character we use the traditional state lookup table to find the next state of the fsm. This table has $2^{(s+k)}$ entries and is s bits wide, and is addressed by the concatenation of the current state and the translated character bits. Infrequent transitions are stored in a content addressable memory that is searched with the same bits as the state lookup table.

An outline of the Split-AC operation when scanning text is the following:

1) The input character c is used to address the character translation table, which produces the translated character c'.
2) The state lookup table and the CAM are addressed concurrently to find the next state. If c' is a "frequent" character, the answer is offered by the state lookup table, otherwise by the CAM.
3) Check if the next state is also a "final" state. If it is "final", a pattern was located and the appropriate action must be taken.

A. *Split-AC Optimizations*

The CAM that implements infrequent transitions is an expensive resource, with cost roughly proportionate to the number of tag bits. In order to reduce the CAM size, we also compress the state bits.

This is possible by assigning sequential numbers to all fsm states that have at least one infrequent transition (we call these the "CAM states"). When the number CAM states is a small portion of the overall number, the grouped states share a common prefix of $s-s'$ bits and have a individual suffix of s' bits. When we perform this optimization, the CAM state bits are partial, and may yield matches even when there should not be any. We address this problem by comparing the remaining $s-s'$ bits in the data portion of the CAM and verifying the bits (and the corresponding transition) later. Effectively the bits are stored in the CAM but in the data instead of the tag portion, saving cost. Finally, we gain additional tag bits by compressing separately the input characters for the state lookup table and the CAM at the cost of increased RAM bits for the translation. The idea here is that usually fewer characters appear in the CAM than in the state lookup table.

IV. IDS RULE SETS AND PARTITIONING

The Snort IDS [6], [7], [8], [9] is an open-source IDS which is used as a point of reference for most IDS-related research. Typical Snort rules look like this:

```
alert tcp 192.168.1.0/24 any -> $EXTERNAL_NET
      100:200 (content: "foo"; sid:100;)
```

The first portion of the rule specifies the required packet header parameters (protocol, source and destination IP-port) as well as the action to be taken if the rule is activated (alert, pass etc.). The second half of each rule specifies the rule options, i.e. what attributes of the packet are considered suspicious. Rule options can specify suspicious payload patterns (the "content" keyword), as well as other additional parameters (flags, IP protocol number etc.).

The Snort ruleset (version 2.4) contains more than 3.000 rules, out of which a large subset can be eliminated for every packet using solely packet header information. To exploit this fact, we group together rules into different sets according to compatible packet header parameters. We use between 3 and 5 different header parameters to classify the rules, which are (depending on the packet type):

- TCP/UDP: protocol, source/destination IP address, source/destination ports.
- ICMP: protocol, source/destination IP address, ICMP type.
- IP: protocol, source/destination IP address.

Rules which have identical header parameters are placed in the same set. When rules apply to a range of protocols, they are duplicated to all applicable groups.

This rule grouping method results in approximately 400 rule sets. If we assign one Split-AC fsm to each rule set then some of the resulting fsms will still have a large number of states (up to 8K states) and still require large amounts of memory to implement. To reduce space requirements we partition large rule sets into smaller subsets which run concurrently and allocate one fsm per subset.

To understand the potential of partitioning, consider a Split-AC fsm with 512 states (9 state bits) and 128 different frequent characters (7 translated character bits). The total required memory for this fsm is $512 * 128 * 9 + 256 * 7 = 576K + 1.75K = 577.5$ Kbits. If this large fsm is divided into 8 smaller fsms with 64 states (6 state bits) and at most 64 frequent characters (6 character bits) each, the total required memory becomes $8*(64*64*6+256*6) = 8*(24K+1.5K) = 204$ Kbits or 65% less memory. The cost in this tradeoff is increased control logic, since we now have 8 fsms running concurrently instead of 1.

V. SPLIT-AC ARCHITECTURE

Figure 2 depicts the block diagram Split-AC architecture for a single fsm. The two memory blocks along with the CAM have registered outputs, and the optimized Split-AC CAM requires some additional logic to control the multiplexor signals and the CAM reset.

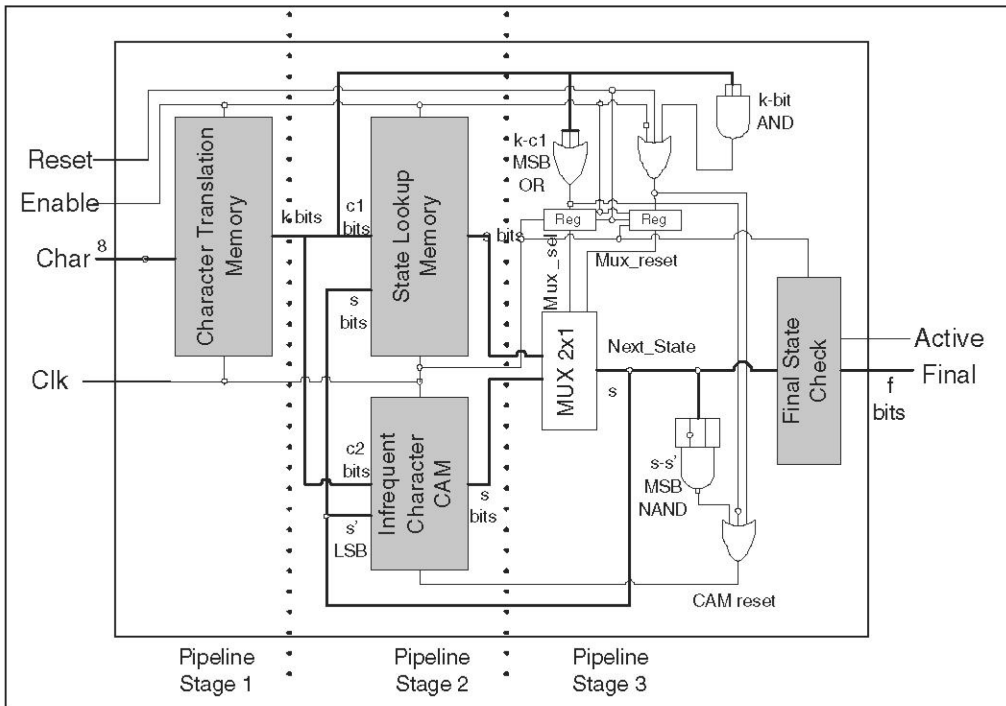

Fig. 2. Optimized Split-AC architecture with compressed CAM states and the corresponding verification.

The architecture is divided into three pipeline stages. The first stage handles the translation of the input character using the character translation memory. At the second stage we have a concurrent lookup in both the state lookup memory and the infrequent transition CAM. During the third and final stage, we decide whether the next state will be given from the state memory or the CAM results as well as check if we have reached a final stage.

The "Final" output signal is activated whenever the fsm reaches a final state and contains information about the parent rule set as well as the specific rule subset that this fsm corresponds to, as well as the id of the final state that was reached.

Figure 3 depicts the entire IDS hardware system, showing how incoming packets are first checked for their header, then categorized into groups that then activate the corresponding Split-AC fsms to scan the payload of the packet. The main components are:

- **Header Classification:** compares the header of the incoming packet to the parameters of every rule group and activates the corresponding "enable" signals.
- **Enable Encoder:** Produces the "leading" rule group, i.e. the one that will be reported on a match.
- **FSM Group x**: A collection of all subset fsms which belong to rule group x.
- **Priority Mux:** large priority multiplexer which is created using a pipelined, tree-like structure of 16-to-4 encoders and 16-to-1 multiplexers. If two or more fsms reach a final state at the same cycle, this multiplexer will forward the results of the fsm with priority. The results of the other fsms will be lost.
- **Position Counter:** keeps track of the search location to be reported when a match is determined.

VI. Implementation and Evaluation

Our work is based on the observation that we can adapt the structure of the system to exploit the regularities of the particular rule sets. Therefore, we target reconfigurable (i.e. FPGA) platforms instead of ASICs. We used the Virtex4 FPGA family by Xilinx, and described our architecture using VHDL.

To automatically generate the necessary VHDL code for our design we modified "T-Gate", a simplified but highly optimized SNORT implementation[10]. T-Gate parses the SNORT rule files, identifies groups and creates its internal structures for rule processing. Starting from that point, we modified T-Gate to produce the VHDL system description and memory initialization files according to the selected rule files and the user-defined parameters. The only hand-written VHDL was for some simple components which are independent of the parameters (for example a 16-to-4 encoder).

The two user-defined parameters are the desired maximum number of states per fsm and the character occurrence frequency threshold Tf that determines how aggressively we employ the infrequent character CAM; a large threshold value (near 1) means that we have few frequent and many infrequent characters (i.e. larger CAMs). We used the unregistered version 2.4 of the Snort ruleset and implemented hardware pattern matching only for the first pattern specified in each rule.

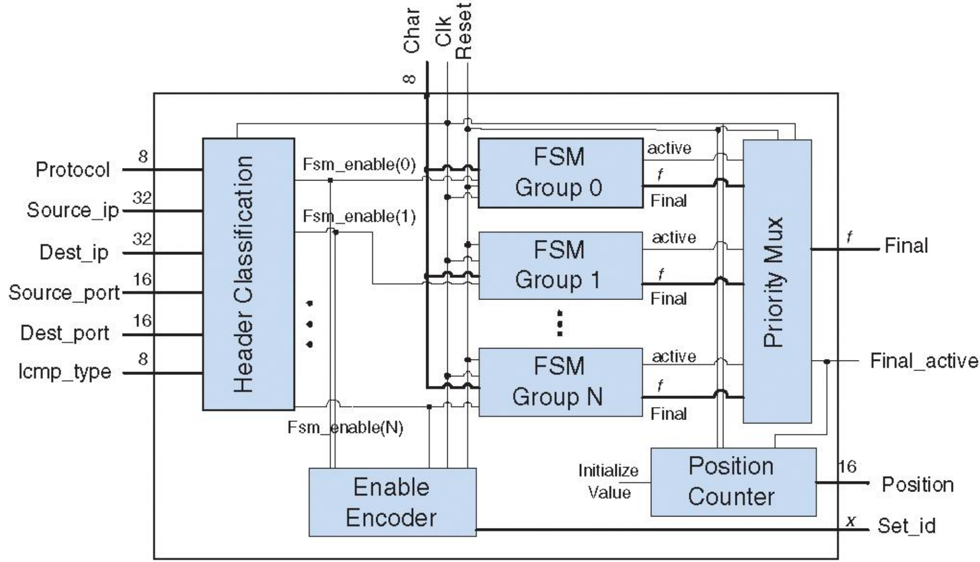

Fig. 3. Complete IDS architecture.

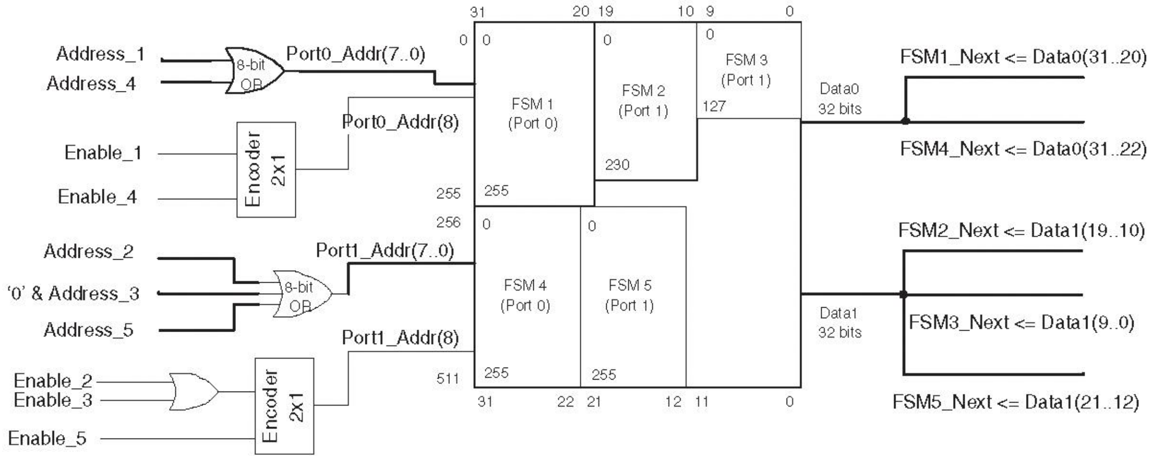

Fig. 4. Memory allocation for many small fsm memories into a single, dual-port FPGA block. By packing multiple independent state transition tables in one memory, we achieve higher memory efficiency.

A. FPGA memory allocation

Initially, we want to bring attention to a problem we faced when mapping the various fsms to an FPGA. The rule set subdivision process can generate up to nearly 900 subsets, each of which requires two different memories. Taking into account that cutting edge Virtex4 FPGAs have a maximum of 552 dual-port memory blocks, it is obvious that we cannot map every fsm memory to a different block.

The solution to this problem is to allocate data from mutually exclusive fsms into the same FPGA block. Two fsms are mutually exclusive if they can never both be active for the same packet. We also take advantage of the fact that the blocks are dual-port by allocating non-exclusive fsm data into different ports in the same FPGA block. In this manner and at the expense of some additional control logic, we achieve a memory utilization percentage between 43% and 94%, based on the selected parameters. Figure 4 shows an example of how many small fsm memories are placed in a single FPGA block.

B. FPGA implementation Results

Figure 5 shows the overall memory requirements when implementing Split-AC in FPGA. We can see that for most Tf values we require the minimum amount of memory when the number of states per fsm is 128. This is due to the fact that a small number of states leads to many subsets, which are inherently not mutually exclusive and, thus, the process of allocating fsm memories to different FPGA blocks becomes significantly more difficult. If we had the capacity to use memories of arbitrary size, the memory requirements would decrease along with the number of states per fsm.

Figure 6 shows the total required CAM tag size, i.e. the number of CAM entries times the number of tag bits. We observe that for a 0.01 threshold we require minimal CAM

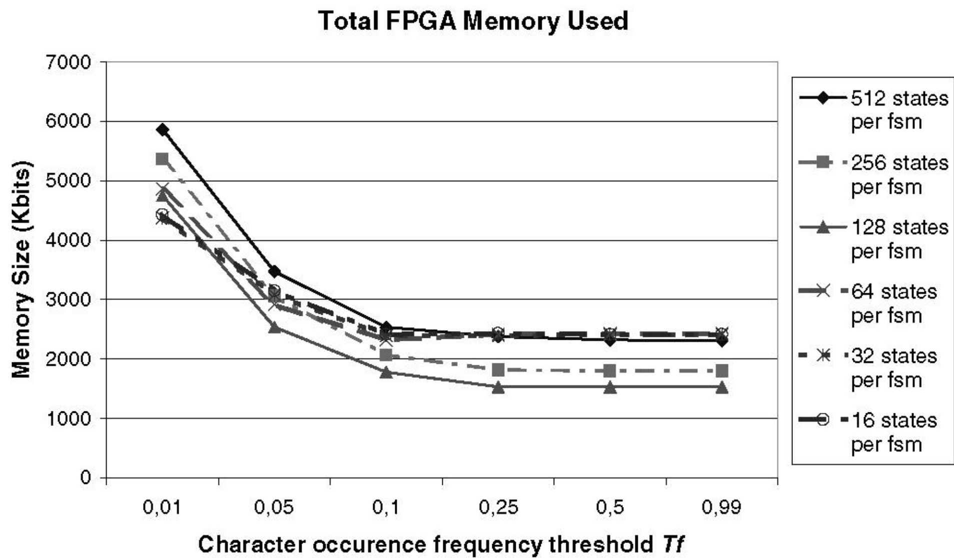

Fig. 5. Size of required FPGA memory. Creating many small FSMs reduces the size of the transition tables but increases their number, and the best results are achieved at 128 states per FSM. Larger values for the Tf threshold reduce memory size at the expense of CAM size.

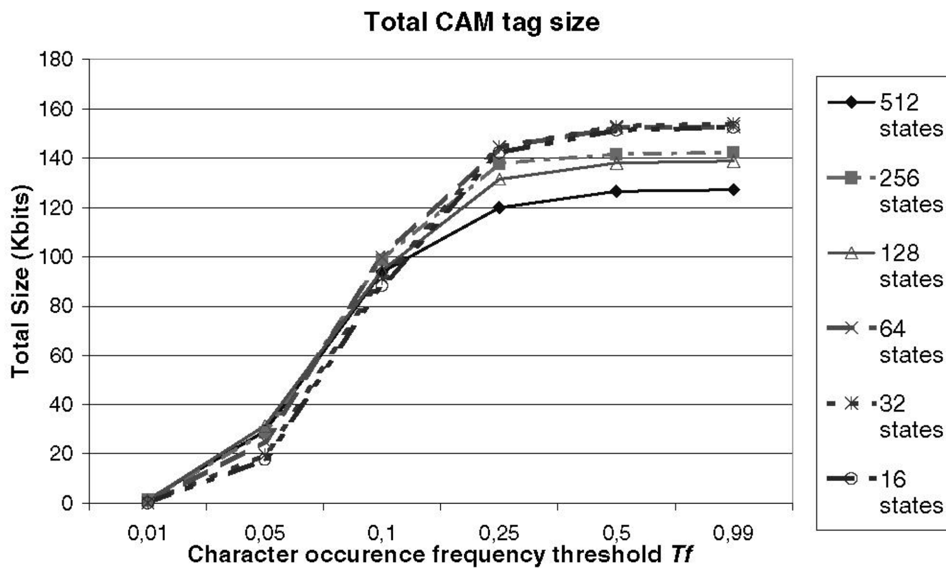

Fig. 6. Total CAM input (tag) size. Increasing the Tf threshold has a significant impact on the CAM size.

TABLE I

POST PLACE & ROUTE RESULTS FOR THE IDS ARCHITECTURE USING THE SPLIT-AC ALGORITHM.

Max States per FSM	32	128	128	128	128
Threshold Tf	0.01	0.01	0.05	0.5	0.99
Min. Period (nsec)	8.6	8.7	8.7	9.8	9.3
Max. Frequency (MHz)	116	114	114	101	107
Max. Throughput (Gbps)	0.927	0.917	0.913	0.811	0.860
# Logic Cells	16,980	12,341	23,629	59,983	60,061
# slice Flip-Flops	4,162	3,224	4,215	6,157	6,190
# Slices	8,970	6,602	12,492	31,821	31,860
# Memory Blocks (18 Kbit)	243	264	141	85	85

TABLE II

COMPARING THE MEMORY REQUIREMENTS OF SPLIT-AC AND BIT-SPLIT FSMS.

	Bit-Split	Split-AC #1 32 states 0.01 Tf	Split-AC #2 128 states 0.05 Tf	Split-AC #3 128 states 0.99 Tf
Total Memory (Mbits)	3.2	4.27	2.48	1.5
# pattern characters	12,812	24,033	24,033	24,033
Bits/character	261	186	108	65

support, less than 1.5 Kbits, which increases to roughly 150 Kbits for larger threshold values.

In Table I we present the Post Place & Route results for several configurations. Here we only included one configuration with a tight FSM size parameter (32) and concentrate on FSM size of 128 states, as it is the best tradeoff (Figures 5 and 6). We can see that the achieved frequency is not very high due to the CAM that is implemented in FPGA logic. The number of required logic cells ranges between 12,341 (for the memory-reliant configuration) and 60,061 (in the CAM-reliant case). Note that these figures include the entire header matching portion of the architecture as shown in Figure 3, while the related works usually omit this portion of the design. The operating frequency of our design is between 100-110MHz, with lower frequencies corresponding to larger CAM -and overall design- sizes. In Table I we also see that increasing the Tf threshold reduces considerably the required memory size. Even for small Tf values such as 0.05 that correspond to small CAM sizes (few tens of Kbits) we see a 40% reduction in memory size. These design points are the most promising, achieving reasonable memory sizes without exploding the CAM size.

C. Comparison to Related Work

We compare our results with those of the bit-split fsms presented in [5], which is currently the smallest AC-based algorithm, and show the results in Table II. These results show that Split-AC is significantly more compact. In the case of configuration #3, which relies heavily on CAM, Split-AC needs 75.1% fewer bits per pattern character than the bit-split fsms do. Configuration (#2 achieves a 60% memory size reduction using around 30Kbits of CAM. Even in the most memory-heavy configuration (#1) that requires minimal CAM support, Split-AC needs 28.8% fewer memory bits per pattern character, showing the effectiveness of the "exception" CAM and the rest of our compaction techniques. In terms of throughput, the work of Tan and Sherwood is considerably better approaching 10Gbps, compared to about 1 Gbps in our work. Part of this difference is due to the difference between ASIC technology used in [5], and FPGA used in our work. A limiting factor in our work is the CAM needed for the infrequent cases. Our work would greatly benefit from both fast and compact CAMs, since in our work the relatively small CAM structures are made with discrete gates.

Compared to other FPGA-based string matching approaches such as [11], [12], [13], [14], [15], [16], [17] our approach is slower in terms of throughput. The two reasons for this difference is again the use of CAM structures, but also the fact that we only process one character per cycle, while other proposed approaches process up to 4 characters per cycle. A compelling option to speed-up AC-based string search is to utilize fast *predictions* for the state transitions[18], and default in the relatively slow but compact state transitions only in the cases of mispredictions.

VII. CONCLUSIONS

We have described and evaluated Split-AC, a memory-efficient version of the Aho-Corasick algorithm. Split-AC is shown to be the smallest, in terms of required memory, variation of the Aho-Corasick algorithm compared to the state of the art. We also sketched an IDS architecture that uses the Split-AC algorithm and fits entirely on a single FPGA chip, so as to implement a complete stand-alone DS.

Split-AC is small in terms of memory footprint but is not the fastest alternative to string matching. A future course of research could be to improve the Split-AC architecture in order to make it faster or to extend it to process multiple character per cycle, thus improving throughput. Other possibilities that stem from the ideas in Split-AC would be to use character

compression with other algorithms in order to reduce their memory requirements.

Finally, intrusion detection is increasingly using regular expressions to describe in a more compact and flexible fashion the attacks ([19], [20], [21], [22]). The nature of AC FSM is similar to the automata used for regular expressions, so it is possible to combine the two approaches into a unified architecture for string as well as regular expression matching.

REFERENCES

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

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

[4] N. Tuck, , T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Mathcing for Intrusion Detection. *Proceedings of the IEEE Infocom Conference*, 2004.

[5] Lin Tan and Timothy Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. *32st International Symposium on Computer Architecture (ISCA 2005)*, 2005.

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

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

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

[9] Sourcefire. Snort rule optimizer. In *http://www.sourcefire.com/whitepapers/ sf_snort20_ruleop.pdf*, June 2002.

[10] Vassilios Dimopoulos, Giorgos Papadopoulos, and Dionisios Pnevmatikatos. On the importance of header classification in hw/sw network intrusion detection systems. In *Proceedings of the 10th Panhellenic Conference on Informatics (PCI)*, November 11-13, 2005.

[11] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Reconfigurable Pattern Matching on FPGAs. In *Proceedings of FPGA '04*, 2004.

[12] 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.

[13] 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.

[14] George Papadopoulos and Dionisios Pnevmatikatos. Hashing + Memory = Low Cost, Exact Pattern Matching. In *Proceedings of 15th International Conference on Field Programmable Logic and Applications*, 2005.

[15] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[16] Ioannis Sourdis, Dionisios Pnevmatikatos, Stephan Wong, and Stamatis Vassiliadis. A Reconfigurable Perfect-Hashing Scheme for Packet Inspection. In *Proceedings of 15th International Conference on Field Programmable Logic and Applications*, 2005.

[17] 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.

[18] Kuo-Kun Tseng, Ying-Dar Lin, Tsern-Huei Lee, and Yuan-Cheng Lai. A parallel automaton string matching with pre-hashing and root-indexing techniques for content filtering coprocessor. In *Proceedings of the 16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 113–118, 2005.

[19] J. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis. Regular Expression Matching for Reconfigurable Packet Inspection. In *Proceedings of IEEE International Conference on Field Programmable Technology*, 2006.

[20] Z. Baker, H.-J. Jung, and V. Prasanna. Regular Expression Software Deceleration for Intrusion Detection Systems. In *Proceedings of 16th International Conference on Field Programmable Logic and Applications*, 2006.

[21] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis. Packet Pre-filtering for Network Intrusion Detection. In *Proceedings of 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2006)*, December 2006.

[22] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2006)*, December 2006.