Traffic-aware Design of a High Speed FPGA Network Intrusion Detection System

Salvatore Pontarelli, Giuseppe Bianchi, Simone Teofili
Consorzio Nazionale InterUniversitario per le Telecomunicazioni (CNIT)
University of Rome "Tor Vergata"
Via del Politecnico 1, 00133, Rome, ITALY

***Abstract*—Security of today's networks heavily rely on Network Intrusion Detection Systems (NIDSs). The ability to promptly update the supported rule sets and detect new emerging attacks makes Field Programmable Gate Arrays (FPGAs) a very appealing technology. An important issue is how to scale FPGA-based NIDS implementations to ever faster network links. Whereas a trivial approach is to balance traffic over multiple, but functionally equivalent, hardware blocks, each implementing the whole rule set (several thousands rules), the obvious cons is the linear increase in the resource occupation. In this work, we promote a different, *traffic-aware*, modular approach in the design of FPGA-based NIDS. Instead of purely splitting traffic across equivalent modules, we classify and group homogeneous traffic, and dispatch it to *differently capable* hardware blocks, each supporting a (smaller) rule set tailored to the specific traffic category. We implement and validate our approach using the rule set of the well known Snort NIDS, and we experimentally investigate the emerging trade-offs and advantages, showing resource savings up to 80% based on real world traffic statistics gathered from an operator's backbone.**

***Index Terms*—Deep Packet Inspection, FPGA, Intrusion Detection System, Snort, String matching, Traffic awareness**

I. Introduction

The demand for network security and protection against threats and attacks is ever increasing, due to the widespread diffusion of network connectivity and the higher risks brought about by a new generation of Internet threats. Network Intrusion Detection Systems (NIDS) play a key role in such a scenario. A NIDS is a system that analyzes the traffic crossing the network, classifies packets according to header, content, or pattern matching, and further inspects payload information with respect to content/regular-expression matching rules for detecting the occurrence of anomalies or attacks.

Software based NIDS, such as the widely employed software implementation of the Snort NIDS [1], cannot sustain the multi Gbits/sec traffic rates typical of network backbones, and thus are confined to be used in relatively small scale (edge) networks. For high speed network links, hardware-based NIDS solutions appear to be a more realistic choice, but the hardware implementation needs to permit the frequent update of the supported rule set, so as to cope with the continuous emergence of new different types of network intrusion threats and attacks.

Field Programmable Gate Arrays are thus appealing candidates. Indeed, an FPGA-based NIDS can be easily and dynamically reprogrammed when the content-matching rules change. Moreover, current FPGA devices are capable to provide a very high processing capability, and support high speed interfaces (FPGA for 100 Gbits/sec processing are available and for 400 Gbits/sec are forthcoming [2]). However, such an increase in the traffic collection ability is not matched with a comparable scaling of the device frequency. Indeed, logic resources still operate with frequencies in the order of "just" hundreds of MHz; for instance a frequency of 500 MHz, that is achievable only by last generation FPGA devices, can process 8-bit characters at "only" 4 Gbits/sec.

This issue is showcased by Figure 1 which reports the historical evolution of a commercial product (Xilinx FPGAs) from 2003 to the time of writing. The y-axis values are normalized with respect to the corresponding parameters of the Xilinx Virtex-II (V2) family. Whereas the number of logic resources (# LUT) has increased 10 times, and the I/O capabilities (bandwidth) has raised up to 400 Gbits/sec[1], achievable in the Xilinx Virtex-7 by 16 transceivers working at 28 Gbits/sec each, the maximum operating frequency (speed) has increased from 200 MHz of a Virtex-II in 2003 to slightly over 600 MHz of the latest Virtex-7 product. In a nutshell, the plot in Figure 1 appears to follow Gilder's law for the evolution of the bandwidth, whereas it appears to follow Moore's law for what concern logic resources.

It is thus straightforward to conclude that, similar to the multi-core parallelization trend in microprocessors, parallelization in FPGA-based NIDS traffic analysis appears to be a mandatory approach to sustain the increased network throughput.

In this work, we address the question of whether there are better ways to parallelize a NIDS architecture, other than the obvious approach of balancing the collected traffic across multiple (equivalent) hardware modules devised to inspect packets using the *same* set of rules. We specifically propose a *traffic-aware* approach. The idea is to first process packets via *Dispatcher* which i) uses elementary header information (Protocol, port, etc) to classify traffic flows into different categories, and ii) accordingly routes packets towards distinct *content matching* engines (hereafter also referred as String Matching Engines), namely hardware modules in charge of supporting the subset of rules devised for the specific category at hand.

As discussed in details later on, even if the basic idea is very simple, turning it into practice is not nearly straightforward

[1]the plot conventionally starts from 1 Gbits/sec, as the Xilinx Virtex-II family was not equipped with transceivers.

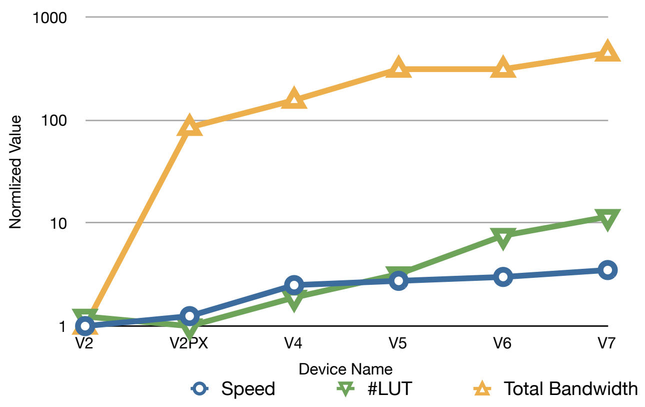

Fig. 1. Evolution of the main FPGA characteristics: speed, number of resources, bandwidth

for several reasons. First, traffic classification rules used by the dispatcher must be extremely simple, and in any case they must be purely based on header information. This restricts the type of classification that can be enforced. Second, such classification approaches yield categories of uneven size in terms of traffic volume, so that dimensioning of the content matching modules cannot be anymore based on the nominal link speed, but must rely on the *actual* per-category traffic load. Third, and most important, the type of classification enforced should attempt to group traffic so that the actual NIDS rules to be enforced in the dedicated hardware modules are as most disjoint as possible (see dedicated discussion of the specific Snort rule set in Section IV), thus minimizing the usage of logic resources. The application of the traffic-aware approach to the hardware domain therefore require a detailed analysis of aspect that are not be covered by previous works.

The specific contributions of this work can be summarized as follows.

Snort rules analysis and relevant classification policies - We analyze the whole rule set of Snort, in order to organize such set into disjoint subsets, identified by suitable combinations of packet header fields. For instance, the rule subset in charge of detecting possible exploits against http servers (protocol=TCP, destination port = 80) obviously differs from the set of rules to be employed by another protocol such as FTP or SMTP; but, perhaps less obviously, also differs from the subset dedicated to analyze threats still for the http protocol, but against web clients (protocol=TCP, source port = 80). Such an analysis yields the classification policies exploited in the dispatching of traffic towards the hardware modules, each supporting one or more subsets[2]. Also in the software implementation of Snort [3] rules are grouped by port/protocol and, for each packet, only the group of rules corresponding to the port/protocol of the packet are checked. This rule partitioning help to reduce memory consumption and CPU usage of Snort. Differently from the software implementation, in our case this partitioned is the first step for applying traffic-awareness.

Real world traffic analysis for HW module sizing - We offline analyze real world traffic, provided by an Internet Service Provider, to quantitatively assess how traffic splits according to the envisioned classification policies, determine the expected worst case per-class throughput, and thus set forth the relevant input rate requirements for dimensioning each content matching HW module. In essence, we apply, for an HW-based development, a methodology similar to the one proposed in [3], where an adaptive algorithm depending on the traffic mix is used to optimize a software-based IDS. We stress that the goal of such an analysis is not to provide an once-for-all system dimensioning, but, rather, to suggest a methodological approach. Indeed, variations in the traffic mix do occur during the operating lifetime of the NIDS, and may also depend on the specific operator's deployment. This does not appear to be a practical concern, as in any case the synthesis of the content matching engine must be rerun at every rule set update (order of once per week) whereas variations in the traffic mix are shown to be much slower, in the order of several weeks [4], [5]. And when significant variations in the traffic mix are detected, the resulting system re-design can be conveniently accounted for, while performing the synthesis associated to the periodic rule update.

HW module implementation and relevant trade-offs - We perform several constrained syntheses (with respect to speed and area) of the different string matching engines, to gather insights in the emerging area/speed trade-offs for the specific NIDS rule set synthesis scenario. We show that if multiple copies of the same string matching engine are used to achieve a higher throughput, the choice between area or speed optimization of the engine is not unique, but strictly depends on the circuits to be implemented and, in some cases, the emerging area-delay trade-offs are quite unexpected (see details in Section V). The use of multiple copies of low-speed string matching engines allows to make an optimization between the area of the single engine, its maximum operating frequency and the overall throughput.

The rest of the paper is organized as follows: Section II presents the architecture of the basic string matching module and discusses the problems related to the implementation of string matching systems. Section III presents the overall system architecture. Dispatching policies and repartition of NIDS rule subsets across the deployed string matching engines is discussed in Section IV, whereas their optimized synthesis is addressed in Section V. Section VI presents the analysis of the traffic taken into account, derives the requested throughput of each subsystem and presents the performance and characteristics of the overall system. Finally, conclusions are drawn in section VII.

II. Implementation of String Matching Circuits

In this section, we first introduce the reference HW string matching circuit design architecture considered throughout this paper, meanwhile addressing related work and extensions. Then, we discuss basic scalability issues which appear to affect

[2]Note that traffic can be tunneled and therefore the traffic classification based only on header information may be evaded. Although this is a serious issue, we consider this to be out of the scope of the present work, as this issue applies as well to *any* rule-based NIDS, including Snort, and it is typically addressed via dedicated traffic classification and anti-evasion techniques.

most design alternatives, and which motivate our proposed packet multiplexing approach.

A. Basic architecture and related extensions

String matching algorithms have been widely studied, for both Software [6] and FPGA Hardware implementation, and software tools for automatic translation from the NIDS rules to the corresponding hardware circuits have been proposed [7], [8]. Moscola [9] proposed a Deterministic Finite Automata (DFA), supporting multi bytes comparisons and partial matches. Sidhu and Prasanna [10], [11] mapped a Non-deterministic Finite Automata (NFA) on an FPGA. These solutions can be applied to pattern matching and extended to matching regular expression such as PCRE (Perl Compatible Regular Expression). Extension to regular expression has been proposed in [8], [11], [12], [13]. These extensions can be based on the use of DFA or NFA, as described in [8], [12], [13], or on the shift-and-compare architecture presented in [14]. Prefix sharing rule saving FPGA's area [14], [18] have been proposed too. Moreover, [19] proposes the use of decoded CAM of hashing functions to achieve data rate up to 8 Gibs/sec on a Virtex-II FPGA, while in [10], [13] a multi character NFA, achieving 8 Gbits/sec is presented. [20] proposes an implementation based on a scalable, highly fine-grain pipelined architecture, able to process up to 11 Gbits/sec. Finally, approaches based on hashing [21], [22] or Bloom filters [23] have been also considered in literature. Even if approaches based on hash can be appealing, they suffer of different drawbacks, related to the number of hash to perform and to the memory requirements. But the main limitation of an hash-based approach is that it hardly complies with the implementation of SNORT modifiers (which other approaches instead easily support).

The basic architecture considered in this paper follows the shift-and-compare architecture presented in [14]. The relevant design is reported in Fig. 2. The main input of the circuit is an 8 bit signal, that transports the payload under inspection one character each clock cycle. The only output of the circuit is the "Match" signal, set to high only if a string is matched. The input is fed into an 8 bits register chain storing the last M packet's characters. The outputs of the register chain are provided as input to a combinatorial network. This latter detects which characters are stored, and performs the AND operation of the detected characters. This signal indicates that a rule has been matched without specifying which rule. If, as suggested in [15], [16], [17], our system is deployed as a snort offloader, devised to forward the malicious packets to a software IDS implementation, a matching signal is all needed to drive a simple pass/drop packet logic. Indeed, note that the deployment of a full-fledged hardware IDS requires supplementary features ($e.g.$ alert generation, packet logging and so on), that can be better performed in software. Besides, we remark that if the goal is to further detect which rule has been matched, a quite straightforward implementation consists in substituting the OR gate with a priority encoding circuit that takes as input the output of each AND gate and provides as output the binary representation of the highest input with high value. A rough estimation of the resource occupation of the encoder is around 15K LUTs[3], that is similar to the value reported in [28]. Fig. 2 further illustrates a toy example which consists in the search for two strings: "**abc**" or "**def**". If the character **d** is stored in the register labeled $x(n-3)$, **e** is stored in $x(n-2)$ and **f** is stored in $x(n-1)$, all the inputs of the uppermost AND gate are equal to 1 and the circuits signal the matching by the "Match" output.

Modifier	Description
offset: N	the search for the content begin after N characters
depth: N	the search for the content ends after N characters
distance: N	the distance between to contents is at least N characters
within: N	the distance between to contents is less than N characters

TABLE I
DESCRIPTION OF KEYWORDS MODIFIERS

A string matching circuit can be implemented using character comparators (realized with a combinatorial network) and shift registers storing the most recent characters.

For example, in [20], [30] a decoded structure is proposed, which allows sharing of the comparators in the combinatorial network. While increasing the number of registers, this structure permits to minimize the combinatorial network, if the number of string to be search is large enough.

Starting from the basic string matching circuit (Fig. 2), we extended it with counters and comparators (following [8]) to support the more specific and complex rules specified by Snort. Specifically, we deployed a global counter and a number of dedicated registers tracking partial matches. This extension allows an easy hardware implementation of the typical Snort rules [1], which usually are expressed in the form of content + modifiers, where

- Content: fixed pattern to be searched in the packets payload of a flow. If the pattern is contained anywhere within the packets payload, the test is successful and the other rule options tests are performed. A content keyword pattern may be composed of a mix of text and binary data. Most of the rules have multiple contents.
- Modifiers: they identify a location in which a content is searched inside the payload. This location can be absolute (defined with respect to the start of the flow) or relative to the matching of a previous content. Table II-A summarizes the most used modifiers.

Fig. 3 shows an example of a rule matched exploiting the extended content matching approach. The rule to be matched is composed of two contents "**ab**" and "**cde**" that must be at a distance less than 10 bytes. The first part of this rule, $i.e.$ the match of the content "**ab**" is performed by the two inputs AND gate. When the content is matched, the value of the global counter is stored in a register. Now, when the second content is matched, the system also checks if the difference between the global counter and the value stored in the register is less

[3]The encoder output requires $log_2(n)$, where n is the number of rules. In our cases of 5700 rules, 13 bits are sufficient. For each output a logarithmic tree of 6-input LUTs can be used. The number of LUTs in the logarithmic tree is $n \cdot (1/6 + 1/36 + 1/126 + \ldots) \approx 0,2 \cdot n$. The total number of estimated LUT is therefore $13 \cdot 0,2 \cdot n \approx 14820$.

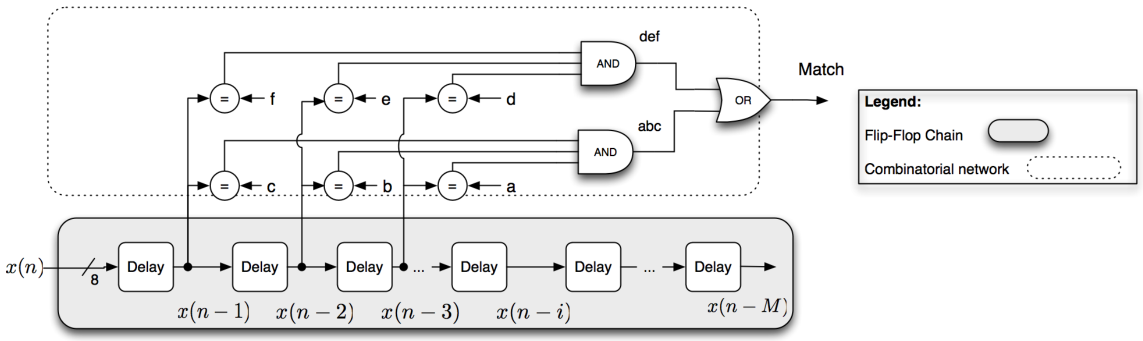

Fig. 2. Basic implementation of the string matching circuit

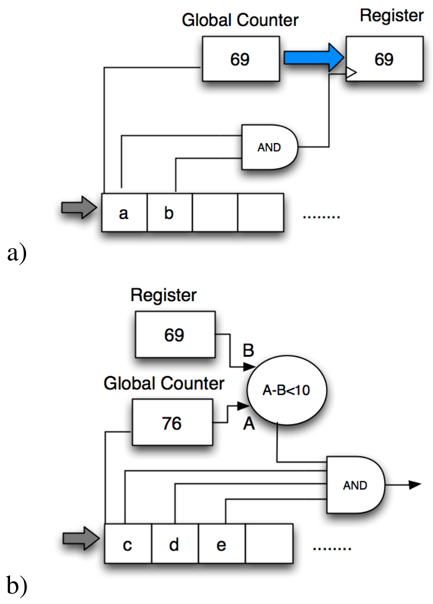

Fig. 3. implementation of a rule composed of two contents: a) first content matching b) second content matching and distance check

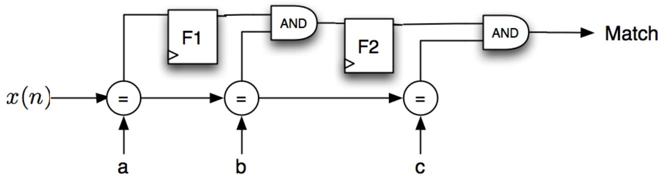

Fig. 4. DFA implementation of the string matching circuit

than ten bytes. This extension is resource consuming because a register and a comparator must be instantiated for each part in which the rule is decomposed. The resource occupation of this block depends on the number of rules implemented.

An alternative implementation of a string matching engine is to rely on a DFA based structure, like the one presented in Fig. 4. This structure shares, with the previous one, the use of a character comparator, followed by some AND gates. The difference is that here, registers (and therefore the state) do not store the last transmitted characters (like in Fig. 2), or the result of their decoding (like in [20], [30]), but an intermediate result that tracks the partial matching of the transmitted characters. In particular, in the example in Fig. 4 the flip-flop labeled F1 stores the matching of **a**, while F2 stores the matching of "**ab**". Note that the number of register elements grows with the number of strings to be searched for, and with the number of characters of each string.

In the rest of the paper we focus our attention on the basic shift-and-compare architecture. We remark that our results can be (at least qualitatively) extended to other alternative constructions, including the DFA architecture above described. Indeed, these architectures ultimately are composed of the same basic blocks, differ only on how and where the matching state is stored, and may be transformed one into the other (or obtain intermediate hybrid architectures depending on the area/speed trade-off) by using suitable retiming algorithms [31], [32].

B. *Scalability issues*

Realistic applications employ several thousand content matching rules (e.g. the Snort basic ruleset includes more than 5700 rules), and strings to be matched can be as long as one hundred characters. In such practical deployment conditions, the combinatorial network can become huge, and the number of registers grows linearly with the number of character of the longest string. The parallel search for different strings corresponds to an increase in the fan-out of the registers and of the character comparators. Also, the fan-in of the AND gates increases with the strings size.

Moreover, the area of the resulting circuit is not only directly affected by the dimension of the combinatorial network, but it is also compromised by the limitations of the logic optimization algorithms, especially when long strings are required by the application. In our synthesis experiments, performed on a last generation workstation equipped with 4 GBytes memory, the logic optimization of a circuit searching for one thousand strings has required 20 hours to be carried out by using the Xilinx XST synthesis software [33]. The results obtained when the number of overall character grows are worse that expected also in the amount of logic reduction.

The other implementation issue that must be faced when the number of overall character grows, is the timing closure problem. This problem is amplified by the high fan-out and fan-in nodes in the circuit. These fan-out nodes are related both to the architecture of the circuit and to the logic optimization performed to increase the resource sharing in the combinatorial network.

As expected, timing closure and area occupation are conflicting issues which require a huge design effort to be solved. The techniques to face these problems are widely known, and span from limiting the fan-out and replicate the logic, to increasing the pipeline and performing speed retiming. In all these cases, when the requested throughput increases to the Gbits/sec rate, these approaches become unfeasible.

Therefore scalability issues becomes very important for multi-Gbps networks and this concern has been already mentioned in some previous works [19], [20].

A solution proposed at architectural level consists in using a wide data bus that operates on multiple characters each clock cycle [19], [13]. With this approach, the data processing rate can be improved without increasing the operating frequency. For example, an 1-character string matching circuit operating at 125 MHz provides a throughput of 1Gbit/sec, while a 4-character based circuit is able to sustain a rate of 4 Gbits/sec. The area required for implementing a multi character circuit increases at least linearly with the number of characters. This approach has several drawbacks that limits the usability of such method. First of all it is not easily scalable since the modification of the number of characters that a content matching engine checks in a clock cycle requires the complete redesign of the content matching engine itself. Moreover, the extension of this method to regular expression can be difficult. Basically the problems related to a multi-character approach derive from the nature of the NIDS rules (and from the nature of the data that are inspected) that are strongly character dependent. Since the rules often are based on matching strings, counting the number of characters and so on, it is not trivial to extend these analyses to a multi-character implementation. Finally, the effort of the logic optimization algorithms for these architectures is greater than one of a single character architecture and also the quality of results is a such case is in doubt. Similar considerations about the limitation of the multi-character approach have been presented in [8]. Moreover, [13] shows that, when number of characters becomes greater than 4, the performances of this approach, computed as throughput/area (Gbps/#LUTs), decreases.

The approach proposed in what follows alleviate these issues, since the area/speed optimization constraints are applied to smaller hardware blocks running at lower frequency.

III. Overall system architecture

As anticipated in the introduction, our proposed system comprises multiple string matching modules. These are further organized into clusters, suitably sized so as to sustain the expected per-cluster traffic load. Packets are balanced across clusters on the basis of policies implemented in a block called *dispatcher*. The overall system architecture is shown in Fig. 5. The main blocks of the system are:

- *network interface* - it collects packets from the network link under monitoring;

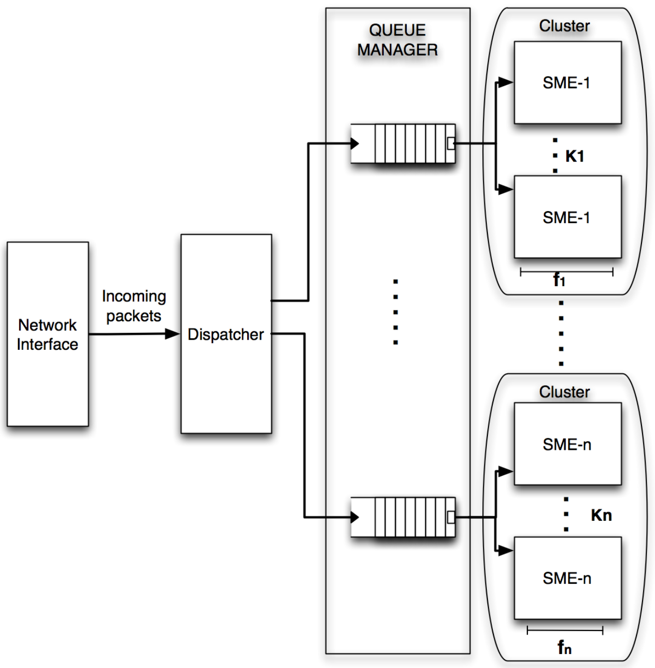

Fig. 5. Implementation of the overall string matching system

- *dispatcher* - it provides a header-based packet classification, whose result is used to determine to which specific string matching cluster the packet is transmitted;
- *string matching engines* - blocks performing string matching; their design is identical (as described in the previous section), but the content searching rules synthesized in string matching engines belonging to different clusters differ and specifically depend on the type of traffic routed to the considered cluster (see Section IV); A generic string matching system is composed of n cluster, each one clocked at a specific frequency f_i and composed of K_i identical SMEs.
- *queue manager* - this block provides a queue for each SME cluster. The queue provides the buffering of packets to cope with packet bursts. The queues can be realized by using external memories to provide enough space. The memory can be partitioned as a set of circular buffers, each one controlled by two pointers. A control FSM, realizing a round-robin policy allows using the memory as a set of independent queues. Since the SME cluster may be clocked with a different frequency, with respect to each other and to the queue manager, asynchronous FIFOs for clock decoupling are deployed between the queue and the SMEs.

Since, as discussed in the previous sections, multi-byte string matching engines do complicate the internal design, the queue output uses 8 bits. Conversely, the interfaces between the remaining modules can be implemented using multiple characters at a time. For example, if the network interface is the 10 Gigabit ethernet core of Xilinx [34], that provides a 64 bits interface working at f_0=156.25 MHz, the data width will be 64 (**N**=64 bits), and the operating frequency of the dispatcher will be f_0.

The architecture shown in Fig. 5 is very flexible and general. The resulting operation in fact depends on a configuration setting which includes the following decisions and parameters:

- *Dispatcher classification policy*;
- *string matching rules loaded over each cluster of engines*;
- *operating frequency of each cluster*;
- *number of string matching engines deployed in every cluster*.

For instance, a basic configuration which we call *agnostic* (indeed used as benchmark in Section VI) consists in *not* using any packet classification policy (or, in other words, deploy a single cluster handling all the offered traffic), but route each received packet to the queue with most available space. Consistently, all the string matching engines are identical in the fact that they must support the *whole* ruleset. Furthermore, their operational frequency f_1 is the same, and it is designed to satisfy the obvious inequality $N \cdot f_0 < 8Kf_1$ where K is the number of deployed engines. The dispatching of packets to different subsequent blocks implementing only a subset of Snort rules has been already proposed in literature (e.g. in [28], [35]). In particular, [28] uses a two-stage approach in which a first pre-filtering stage selects which rules of the Snort rule-set must be further analyzed by the second stage engine. The first stage inspect the incoming packet with respect to the first part of each rule. This first stage is more complex with respect to our approach, that only requires to inspect the header to detect ports and protocol. The dispatching policy of [35] is similar to the one we proposed, since it is based on port/protocol classification. This allows optimizing the memory requirements of the FSMs implementing the Aho-Corasik [6] algorithm. The overall Snort rule-set is split in different FSMs , saving memory with respect to an unique FSM containing all the rules. The work in [35] try to optimize memory occupation, while our work focuses on logic optimization, since we implemented the string matching engines by means of comparators and shift-registers. However, the main difference of these paper from our approach is that both the works [28], [35] do not take into account any information about the traffic rates for dimensioning the second stage engines. As will be shown in Section VI, this difference allows an impressive resource saving with respect to the above mentioned approaches.

Conversely, we are interested in exploring the performance gains that may be achieved by *dispatching different traffic types to different clusters, consistently distributing different content matching rules over different engines*, *independently optimize the area-frequency tradeoff for each deployed engine*, and *dimensioning each engine depending on the traffic-load conditions*. For simplicity, we take a practical three-steps design, organized as follows.

Step 1: ruleset distribution and relevant packet dispatching policy. The first, necessarily heuristic, step, is to distribute different content matching rules across multiple engines. Such distribution is driven by two practical requirements: i) permit an elementary dispatching policy, based on simple protocol header information, meanwhile ii) attempt to obtain (as much as possible disjoint) subsets of size smaller than the whole rule set. Our proposed classification, described in the next section IV, indeed relies on trivial protocol/port information, thus permitting a straightforward implementation of the dispatcher. It is worth noting that per-protocol grouping of string matching rules is the most natural direction, as in practical NIDS such as Snort, rules defined for a same protocol not rarely share common sub-strings (for instance, the string "HTTP" requires to be matched by most rules applied to protocol:TCP and destination port:80), and hence may yield savings in the subsequent HW circuit design.

Step 2: per-engine optimized HW design. For each specific engine (and its subset of different rules) we performed in Section V a large amount of syntheses in order to identify the best tradeoffs in terms of throughput/area. Quite surprisingly, such trade-off significantly depends on the specific ruleset considered. The output of this second stage design is the frequency at which each engine is implemented.

Step 3: Traffic-load-based system dimensioning. Finally, in Section VI we perform an experimental analysis of real-world traffic devised to provide information about the per-cluster load, and consequently determine how many copies of each synthesized engine are needed to sustain the resulting load.

Obviously, the outlined approach is open to improvements, by using information here exploited for individual steps in a more holistic design procedure (e.g. use traffic information for determining how to distribute rules across engines), although it does not appear simple to move from heuristics to a more formal design methodology. Finally, even if here we refer always to the string matching engines described in the previous section, we outline that this method can be generically applied to many of the string matching systems proposed in literature. For example, also [21] could benefit of a partitioned traffic-aware implementation, since the implementation techniques used to improve performances ($i.e.$ pipelining, parallelism, and memory replication) suffer of the same scalability issues already mentioned in previous section for other techniques ([13] [19]). Instead, packet level parallelization should be able to better exploit area/delay trade-off than the classical parallel/pipelined implementation, and the traffic-awareness could be easily reduce the memory replication.

IV. Snort ruleset subdivision

We started from the analysis of the Snort [1] ruleset. The basic ruleset for Internet Traffic analysis comprises 5567 rules. These are already pre-classified on a per-protocol basis (TCP, UDP, IP including ICMP etc). Note that a non overlapping (disjoint) partitioning of the Snort rules is not feasible, as some rules, for instance those related to the IP protocol, must be obviously applied to all the incoming network traffic.

The quantitative results of our analysis are summarized in Table II. This table splits the Snort rules on the basis of the type of considered traffic, specified in terms of protocol (IP, TCP, UDP) and port (HTTP, IANA assigned non HTTP, i.e., smaller than 1024, and ephemeral i.e., greater or equal than 1024). The direction of the traffic (i.e. source versus destination port numbers) is further accounted in our proposed

TABLE II
SNORT RULES BREAKDOWN

Type of traffic	Number of rules	Set
IP rules	34	A
TCP rules without port specification	491	B
UDP rules	384	C
TCP rules with DST port 80 (uplink: HTTP towards servers)	1869	D
TCP rules with SRC port 80 (downlink: HTTP towards clients)	812	E
TCP rules with DST port not 80	1977	F+G
TCP rules with DST port $<$ 1024 and not 80	1157	F
TCP rules with DST port $\geq$ 1024	820	G
total number of rules	5567	-

TABLE III
MAPPING OF RULE SUBSETS OVER STRING MATCHING ENGINES

String Match Engine (SME) Name	Sets	Number of rules
SME-UDP	A+C	418
SME-WEB_UPLINK	A+B+D	2394
SME-WEB_DOWNLINK	A+B+E	1337
SME-NONWEB_PORTLOW	A+B+F	1682
SME-NONWEB_PORTHIGH	A+B+G	1345

classification. Such split has the advantage that a classification policy is trivially supported over the dispatcher.

Table II's rule breakdown is based on the following considerations:

- Set A (IP rules): used for the IP protocol, and hence applied to all TCP and UDP packets and to be supported on all the string matching engines; note that the number of rules in this set is very small (only 34).
- Set B (generic TCP rules): applied to all TCP traffic, irrespective of the specific application supported (i.e. port number), and hence to be supported on all the TCP-related string matching engines; although greater than the previous IP case, the size of such set is still relatively small (491 rules) especially if compared with the following sets.
- Set C (UDP rules): dedicated to UDP traffic; this set is fully disjoint from the previous set B (TCP traffic) and hence can be supported over a different string matching cluster dedicated to UDP traffic analysis.
- Set D (HTTP server rules) and set E (HTTP client rules): these two set of rules are dedicated to the analysis of Web (HTTP) traffic. The rules devised to inspect the traffic addressed towards web servers (namely, set D) is fully disjoint from the rules devised to inspect traffic in the opposite direction (set E); this suggesting the deployment of two independent string matching clusters for the two cases.
- Set F and G (non web TCP traffic): these rules, in total 1977, are devised to analyze non web traffic. They can be conveniently partitioned into two disjoint subsets: set F comprising 1157 rules, dedicated to inspect non web traffic generated by IANA-registered applications (port number lower than 1024), and set G comprising the 820 rules devised to analyze the remaining traffic.

These considerations suggest to deploy 5 different string matching engine designs, summarized in Table III, along with the relevant number of supported rules. Note that, in the worst case of the engine to be included in the cluster dedicated to the analysis of traffic addressed to a web server, only 2394 rules out of the initial set of 5567 shall be implemented, i.e. about 43%. Finally, we stress that the number of rules supported by each engine is just a very rough indicator of the expected circuital complexity, as this latter depends on the *specific* rules to be deployed, indeed largely differing in terms of size of strings to be matched, rule modifiers to be accounted, and so on. As a matter of fact, Table IV presented in the next section highlight that despite the large amount of rules, SME-WEB_UPLINK is very conveniently implemented (thanks to the recurrent sub-string patterns to be matched and the limited usage of Snort modifiers), especially when compared with SME-NONWEB_PORTLOW, despite this latter contains a significantly lower number of rules ("only" 1682 versus the 2394 of SME-WEB_UPLINK).

V. STRING MATCHING ENGINES SYNTHESIS

The five String Matching Engines (SMEs) identified in the previous section are implemented as independent modules; hence their synthesis can be independently optimized. We carried out, per each SME, a number (between 5 and 10 each) of syntheses with different speed constrains, so as to identify the one(s) which achieve the best area/speed trade-off.

Specifically, in order to maximize the throughput by using the fewer number of logic resources, we evaluated the area-delay product of each implementation, computed as the ratio between number of LUTs and the maximum operating frequency, and selected the one with the lowest area-delay product.

Interestingly, depending on the complexity of the circuit to be synthesized, qualitatively different results have been obtained. For large circuits, such as the case of SME-NONWEB_PORTLOW, optimization criteria are not straightforward as, the best implementation is at an intermediate, hardly predictable, operating frequency. Indeed, the plot in Fig. 6 a) shows the number of used FPGA logic elements (y-axis) versus the maximum achievable frequency, for different implementations of SME-NONWEB_PORTLOW.

The two most resource consuming implementations require almost 48000 LUTs and run at 275 and 295 MHz, while the least consuming one requires 32408 LUT and runs at 221 MHz. The solution that maximizes the throughput can be easily identified as the one with the lowest Area-Delay

Product. To this purpose, Figure 6.b) plots the ratio between number of LUTs and Maximum Frequency. It can be seen that the best choice is the circuit running at 221 MHz, whereas circuits with less area or higher frequency achieve lower performance.

Conversely, for small circuits the implementation with the lowest area-delay product is typically the fastest one. As an example, Fig. 7 documents results obtained for the SME-WEB_UPLINK case; the remaining SMEs are qualitatively similar to this and not reported to save space.

Overall, our experiments appear to further confirm, for the specific scenario of string matching here tackled, the fact that when circuit complexity increases, limitations in the synthesis process largely impact results. Indeed, with small SMEs, speed optimization is carried out at cost of a limited increase in logic resource consumptions (the number of LUTs for the various implementations differs of just about 5%), thus making the implementation with the highest speed also the one that provides the best throughput/area trade-off. Instead, when the circuit size grows, not only the best implementation in terms of throughput/area trade-off is for an intermediate speed, but the variability in terms of number of LUTs is up to as much as 50%, being 32408 the smallest one and 47516 the fastest one. We believe that this is caused by limitations in the heuristics exploited by the synthesis algorithms. All data presented in this section are referred to result obtained without enabling the retiming available by the Xilinx synthesis tools. Enabling retiming we achieve results that are quantitatively different, but very similar to those presented here.

Table IV summarizes the best throughput/area results achieved for all the 5 SMEs synthesized. For sake of comparison, the table further reports results obtained by the synthesis of a single SME supporting all the rules. Note that such a single synthesis uses more than 8% extra LUTs with respect to the multiple SME case[4], and that in all cases but one the resulting single SME implementation has a lower speed. This is quite remarkable, since, as discussed in section IV, a disjoint partition of rules was not technically achievable (rules belonging to sets A and B had to be reimplemented in most of the SMEs - see table III and the total number of rules implemented is thus 7176).

TABLE IV
SYNTHESIS RESULTS FOR THE BEST IMPLEMENTATIONS OF THE FIVE SME

SME Name	Frequency (MHz)	# LUT
SME-UDP	337	5220
SME-WEB_UPLINK	303	6055
SME-WEB_DOWNLINK	288	8273
SME-NONWEB_PORTLOW	221	32408
SME-NONWEB_PORTHIGH	340	7810
Unified set	241	64701

VI. TRAFFIC-AWARE SYSTEM DIMENSIONING

Up to now, the discussion has been limited to the design of single modules. We now address our initial goal, i.e., how to dimension the entire NIDS, and which savings our approach may accomplish. We recall that our approach promotes a *traffic-aware* dimensioning. Our dispatcher routes different traffic types to different string matching clusters on the basis of the enforced classification policy. As a result, a string matching cluster is *not* required to support the entire traffic, but shall sustain only the amount of traffic belonging to the considered type. It follows that the number K_i of SME replicas to be deployed within each cluster i (refer back to Fig. 5) can be adapted to the actual per-cluster traffic load, in turns lower than the network link throughput.

First, let's compute, as benchmark, the system sizing in the assumption of a *traffic agnostic* system where a single string matching engine is designed including all rules, and it is replicated to sustain a total peak load of 10 gbps. From table IV, we see that the best throughput/area implementation for the case of single SME supporting all the IDS ruleset operates at 241 MHz, i.e. it can support up to about 1.9 gbps. Therefore a cluster of 6 replicated engines is needed to sustain a 10 gbps peak load, which in turns implies that a total number of 388206 LUTs are required.

In order to dimension a traffic-aware system, information about the actual traffic composition is needed. Of course, the overall sizing depends on such a traffic mix, and hence on the specific deployment case, so that no universally valid dimensioning rules are possible. Nevertheless, some insights on the extent of such a resource saving may be gathered by analyzing specific (and realistic) use case deployments. In details, in what follows we base our quantitative considerations on the analysis of 68GB of traffic made available by a regional ISP during the experimental demonstration of the european research program FP7-PRISM [36]. Unfortunately, we have no access to a tier one network operator running 10 gbps links, hence we need to rely on the assumption that the traffic hereafter analyzed is representative also for a core network backbone (assumption which appears reasonable as such traffic is the result of the multiplexing of several traffic aggregates collected by smaller ISPs).

traffic dataset analysis: average values

Our first step consisted in the analysis of the traffic dataset, and the extraction of statistics which can be directly related to the classification promoted in section IV. Table V reports statistics based on number of packets and amount of delivered bytes. The table further reports the SME cluster in charge of handling the relevant traffic.

In terms of dimensioning, our main concern relates to the traffic rate measured in each category. The table shows that three traffic classes, namely IP/UDP, web downloads, and non

[4]Note that these considerations are purely based on synthesis results; when we move to the entire system dimensioning and include traffic awareness in the system sizing, the overall resource saving is much greater - see next section VI.

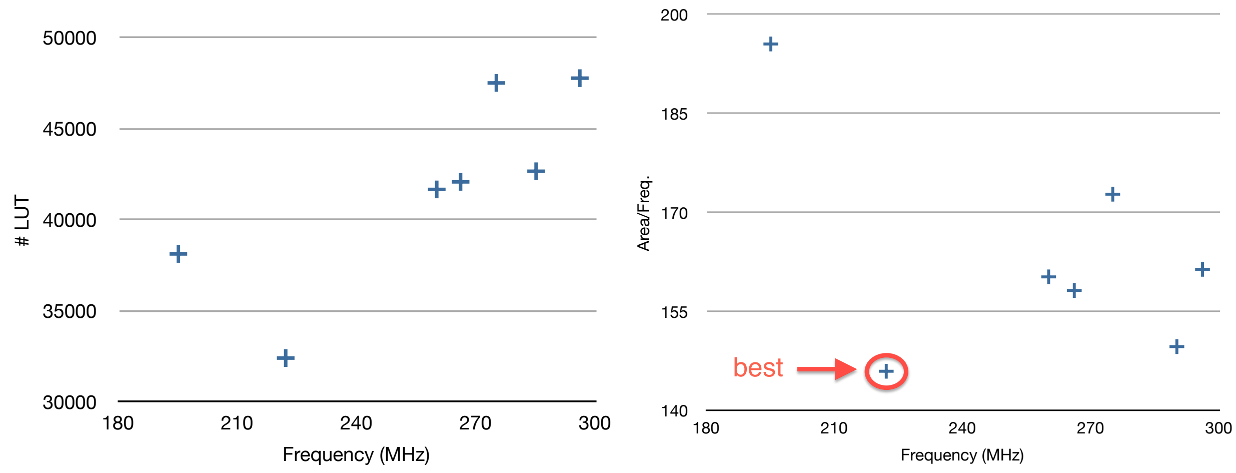

Fig. 6. Area and Area/frequency synthesis results for the SME-NONWEB_PORTLOW

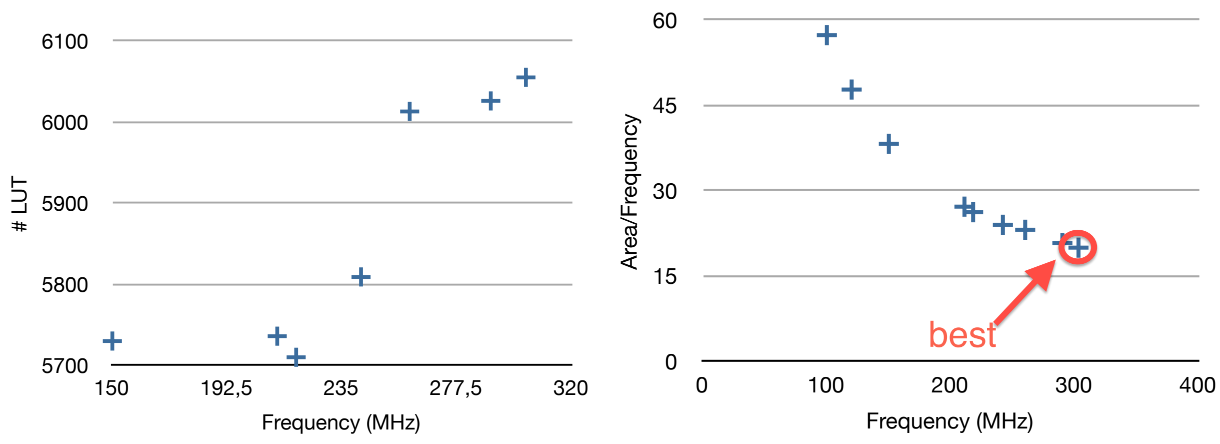

Fig. 7. Area and Area/frequency synthesis results for the SME-WEB_UPLINK

TABLE V
TRAFFIC STATISTICS BREAKDOWN

Type of traffic	% of packets	% of bytes	Handling SME cluster
IP+UDP	31.1%	14.3%	SME-UDP
TCP, DST port 80	9.0%	1.7%	SME-WEB_UPLINK
TCP, SRC port 80	14.2%	33.8%	SME-WEB_DOWNLINK
TCP, non HTTP, port $<$ 1024	0.2%	0.1%	SME-NONWEB_PORTLOW
TCP, non HTTP, port $\geq$ 1024	45.5%	50.1%	SME-NONWEB_PORTHIGH

web traffic using high ports (hence including several peer-to-peer applications) largely dominate. The average rate for the two remaining traffic classes is very low, to the extent that, in the assumption of an overall load of 10 Gbits/sec, the traffic belonging to the SME-NONWEB_PORTLOW accounts for as little as 10 Mbits/sec[5], and the traffic addressed to web servers (i.e. HTTP requests or POSTs) remains well below 200 Mbit/sec.

[5]We remark that such a very small traffic load may be trivially handled via a software implementation of the relevant string matching engine. In what follows, for consistency of presentation, we assume that all SMEs are implemented in hardware, but this finding suggests that a further significant saving may occur if hybrid HW/SW implementations are considered.

traffic dataset analysis: fluctuations

Clearly, the IDS system dimensioning should not be performed on just average traffic loads, but must be robust to traffic fluctuations. Indeed, figure 8 graphically plots the per-traffic-type fluctuations experimented versus time. The plot in the figure focuses on a 4 GB trace collecting 8 million packets: results obtained for other traces available do not significantly differ. The result presented in table VI shows the average and plot the peak rate for the different types of traffic.

Queue dimensioning

Note that the choice of the throughput at which each SME shall be operate must be strictly greater than the average

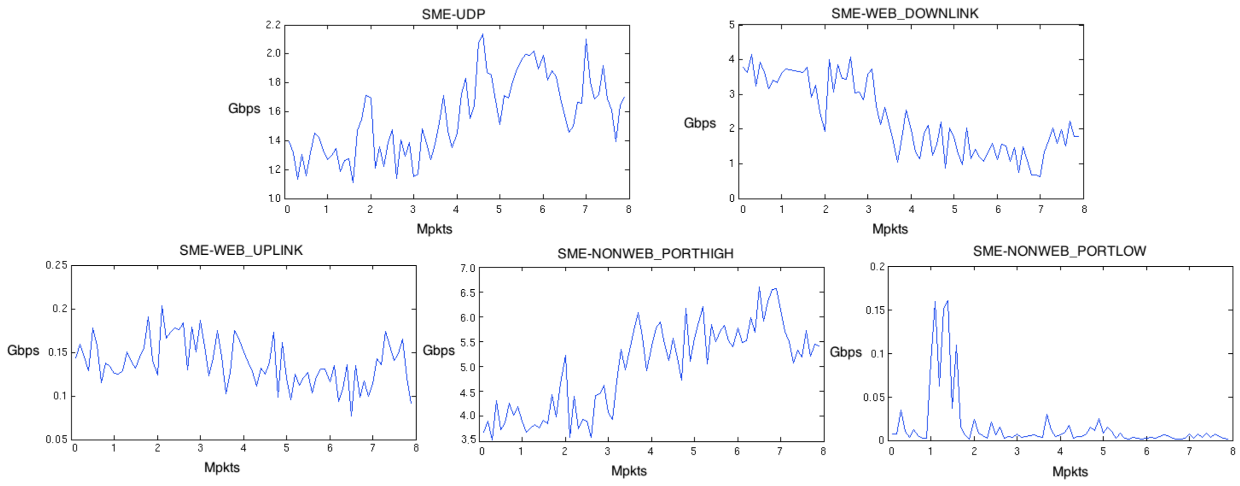

Fig. 8. Time dependency of traffic rate of the analyzed trace - x-axis unit = 1 Million of packets

TABLE VI
PEAK RATE OF THE DIFFERENT TYPES OF TRAFFIC

SME Cluster	Average values (Gbits/sec)	Peak Rate (Gbits/sec)
SME-UDP	1.43	2.2
SME-WEB_UPLINK	0.17	0.2
SME-WEB_DOWNLINK	3.38	4.1
SME-NONWEB_PORTLOW	0.01	0.2
SME-NONWEB_PORTHIGH	5.01	6.5

throughput, otherwise loss at the queues would be unavoidable. Moreover, the larger the SME throughput, the lower the requirements on the queue buffer sizing. A thorough dimensioning of the queue buffer requires an in depth queueing analysis of the traffic variations (long term behavior, correlation, etc). Such level of analysis is out of the scopes of the present paper for at least two reasons: first, it would require a much larger dataset captured at different day/week times; second and most significant, such an analysis is worth only when tailoring the design to an *actual* real world deployment. And, even in in this case, measurement-based approaches consisting in appropriately resizing the system at every periodic IDS ruleset update (and related HW module re-synthesis) on the basis of historical measurements appear more practical. However, to provide some quantitative insights in the behavior of the system with respect to this issue, we have performed a set of simulations devised to quantify the queues occupancy at different service rates (the SME throughput). The simulations have been fed with the real traffic traces. From the data of table VI we can see that the SME-WEB UPLINK and SME-NONWEB PORTLOW have extremely low throughput requirements with respect to the one achievable with an SME (a SME clocked at 100 MHz provides 0.8 Gbits/sec). Using the service rate of 0.8 Gbits/sec for this traffic type the maximum queue occupancy remains always lower than 25 KB for all the analyzed traces in the case of SME-WEB UPLINK (see Fig. 9.a) and 20KB (see Fig. 9.b) for the SME-NONWEB PORTLOW traffic.

Instead, for the other three clusters which exhibit a more significative amount of traffic, we analyze the queue occupancy when the service rate is set to the average rate incremented by 5 %, 10% or 20%. The data has been reported in Fig. 9.c 9.d and 9.e for SME-UDP, SME-WEB DOWNLINK and SME-NONWEB PORTHIGH respectively.

By using as service rate the average rate incremented by 5%, the maximum occupancy of the queues was 150KB for the UDP traffic and 1.8 MB for SME-DOWNLINK and 1 MB for SME-PORTHIGH traffic. Instead, by oversizing the service rate up to 20%, the queue occupancy decreases to 100KB for UDP traffic, 600 KB for SME-DOWNLINK and 250 KB for SME-PORTHIGH traffic. These results provide some suggestion on the practical sizing of the queues used in our system. Worth to mention is the fact that even a limited oversizing of the SME throughput allows to achieve a relatively small queue size.

A. Implementation details

The target board for our implementation is the INVEA COMBO-LXT [37], an express PCI x8 mother card equipped with the XILINX Virtex5 XC5VLX155T [38], two QDR RAM memories and up to 4 GB of DDR2 memory. The 2 QDR II SRAM chips provides high bandwidth dual port memory for routing tables, flow memory, low latency data buffers. The total capacity is of the QDR is 9 MB with a throughput of 17166 Mbps for read operation and 17166 Mbps for write operation. The amount of memory provided by the QDR memory fully satisfies the memory requirements for the queues, as estimated in the previous subsection. An alternative solution is the use of the DRAM that is extremely abundant with respect to the queues size requirements. This memory can be used when the problem of out of order times, due to FPGA reconfiguration must be taken into account. FPGA reconfiguration occurs when a new ruleset is loaded into the system, or as a consequence of the detection of a sudden change in the traffic-mix rates due to a malicious event (*e.g.* a Denial of Service attack). As discussed in [39] long queues can be used to face the out of order time needed for FPGA reconfiguration. The reconfiguration time of the entire XC5VLX155T FPGA is around 300 ms [40], corresponding

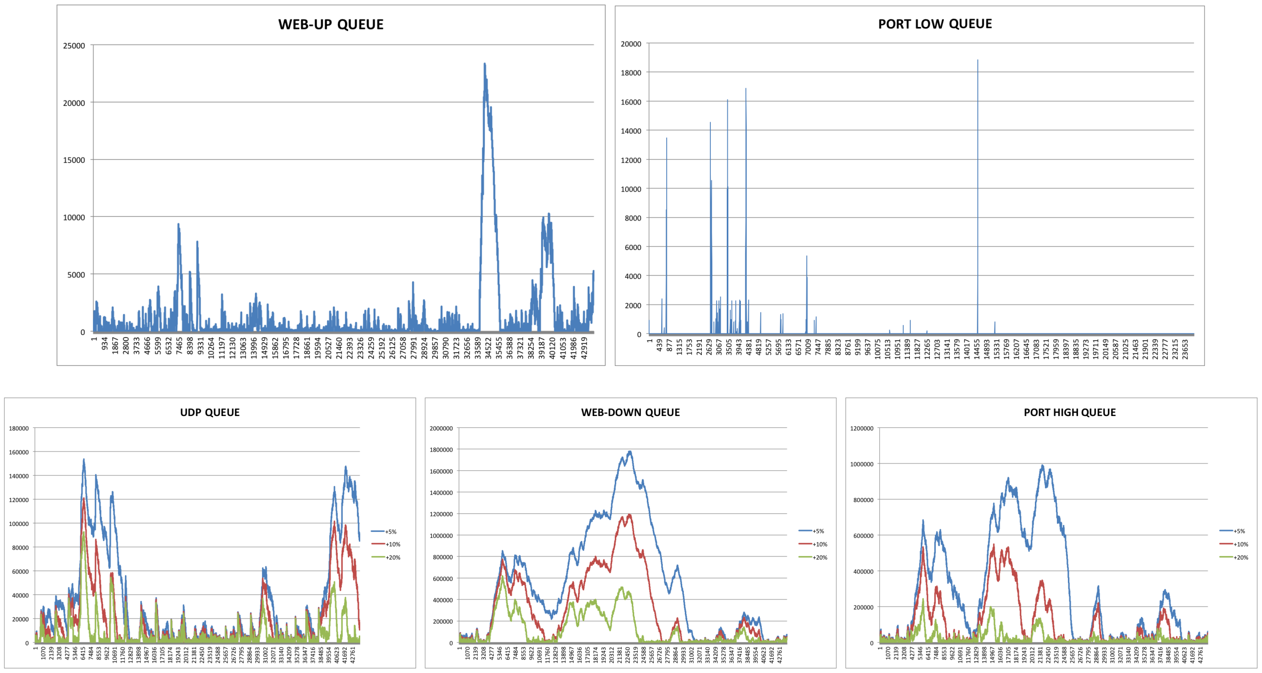

Fig. 9. Queue occupancy for the different clusters

to 3 Gb (less than 512 MB) for a 10 Gbits/sec peak rate. This queue cannot be implemented in the QDR memory, but can be easily implemented in the DRAM. The board is equipped with a daughter board providing two 10 GbE interfaces, and is hosted in a PC workstation. The operating system sees the board as a traditional network card, and therefore the filtered packets coming from the FPGA can be directly provided to a software IDS. In order to simplify the design we restrict the system to work on four clock frequencies: 156.25 MHz for the dispatcher and network interface, and 100, 280 and 300 MHz for the SME clusters[6]. The Xilinx Digital Clock Manager Modules (DCM) [38] have been used to provide the different clock source to the different SME clusters implemented on the COMBO-LXT card.

Table VII shows that three clusters, namely SME-UDP, SME-WEB_UPLINK and SME-NONWEB_PORTLOW, can be implemented with just a single SME, thanks to the low relevant traffic load. SME replication is needed only in the case of SME-WEB_DOWNLINK, where two SMEs permit to achieve a throughput of 4.48 Gbps, sufficient to sustain the expected load, and in the case of SME-NONWEB_PORTHIGH, where three SME replicas are needed. The total number of SME is 8. In order to exploit the QDR available in the target board to realize the queues for packet buffering, we allocated for 2 MB for the queues corresponding to the SME clusters with high traffic (SME-UDP, SME-WEB DOWNLINK and SME-NONWEB PORTHIGH) and a generous 1MB for the other queues. Following the discussion presented in the previous sub-section, this amount of memory is sufficient to avoid that the queue is going full when burst of packet of the same types arrives. The most remarkable result shown in Table VII is the *total* amount of LUTs needed to implement the system, only 77664. When compared with the size of a traffic agnostic system implementation (388206 LUTs), *the resource saving is as much as 80%*. Besides, note that almost half of them are required to implement the SME-NONWEB_PORTLOW SME which could alternatively implemented via SW given the marginal rate requirements (see discussion in footnote 5). The last row of the table reports also the area and throughput of an agnostic system with only one instance of each SME. This system has an area occupation similar to the traffic-aware system but provides a throughput that is less than 2 Gbps. Even if it is really difficult to carry-out a fair comparison between our proposed system and other ones proposed in literature, (different FPGA technologies, different versions of the ruleset ecc.) we outline that our agnostic system has comparable performance and resource occupation with other system proposed in literature (e.g. [28] reports 180K LUTs and 14 Gbps for a Virtex4 implementation). Instead, the traffic-aware implementation allows a significant savings in terms of area with respect to the approaches previously presented in literature.

VII. Conclusions

This paper shows that the exploitation of traffic classification and load statistics may bring significant savings in the design of HW Network Intrusion Detection Systems (NIDS). Specifically, we have presented a traffic-aware NIDS architecture, where a dispatcher forward different traffic types to string matching engines supporting different IDS rulesets. This basic idea has been developed in three steps. First, considering the well known Snort NIDS as reference use-case, we have analyzed the relevant ruleset and subdivided rules into subsets handling different traffic types. Second,

[6]actually the COMBO-LXT also another clock source for the PCI interface, that we do no take into account

TABLE VII
OPTIMIZATION RESULTS FOR THE IMPLEMENTATIONS OF THE FIVE SME. 100%

SME name	Number of copies	Frequency (MHz)	120% of average rate (Gbits/sec)	Achieved throughput (Gbits/sec)	Total number of LUTs
SME-UDP	1	300	1.71	2.4	5220
SME-WEB_UPLINK	1	100	0.012	0.8	6055
SME-WEB_DOWNLINK	2	280	4.05	4.48	16546
SME-NONWEB_PORTLOW	1	100	0.2	0.8	32408
SME-NONWEB_PORTHIGH	3	300	6	7.2	23430
Total	N/A	N/A	N/A	10(*)	77664
Agnostic system	6	241	N/A	11.5	388206
Single instance of the agnostic system	1	241	-	1.91	64701

(*) 10 Gbits/sec under the traffic conditions of Table VI

we have optimized the HW implementation of each specific String Matching Engine supporting such derived rule subsets. Finally, we have dimensioned the system based on traffic statistics experimentally gathered from a real world operator's deployment.

Numerical results show that our proposed design methodology yields significant advantages in terms of resource savings. In the specific experimental scenario considered in this paper, an 80% reduction in the number of LUTs was accomplished. Such resource savings stem from the following main reasons. First, in most generality, traffic awareness permits to fine tune the amount of HW resources dedicated to each traffic category. More specifically, replication of string matching engines is restricted to the case of highly loaded traffic categories. Second, each string matching engine supports only a subset of IDS rules, and thus optimization of its HW implementation appears more effective. Finally, our results point out a very interesting NIDS-specific consideration: the IDS rules which are by far more complex (i.e. resource consuming) in terms of HW implementation are associated to traffic classes whose load is marginal. Such finding questions the effectiveness of a full-HW IDS implementation, and rather suggest that further significant resource savings appear in principle possible when extending the work presented in this paper to traffic-aware *hybrid HW/SW* design.

REFERENCES

[1] Sourcefire, "Snort: The Open Source Network Intrusion Detection System", available at http://www.snort.org.

[2] Xilinx Website, available at http://www.xilinx.com/

[3] S. Sinha, F. Jahanian, J. Patel, "Wind: Workload-aware intrusion detection", Recent Advances in Intrusion Detection, Springer, pp. 290–310, 2006.

[4] K. Papagiannaki, N. Taft, Z.-L. Zhang, C. Diot, "Long-term forecasting of Internet backbone traffic: observations and initial models," in Proc. of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications, INFOCOM 2003, pp. 1178-1188.

[5] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, K. Cho, "Seven years and one day: Sketching the evolution of internet traffic", in Proc. of the Twenty-Eight Annual Joint Conference of the IEEE Computer and Communications, INFOCOM 2009, pp. 711-719.

[6] A.V. Aho, M.J. Corasick,"Efficient String Matching: An Aid to Bibliographic Search", Communications of ACM, Vol. 18 n. 6, June 1975.

[7] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in Proc. of IEEE Symp. on Field-Programmable Custom Computing Machine, FCCM 2002, pp. 111-120.

[8] J. Bispo, I. Sourdis, J. Cardoso, S. Vassiliadis, "Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues", in Proc. of the 3rd international conference on Reconfigurable computing: architectures, tools and applications, ARC 2007, Springer-Verlag.

[9] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," Proc. of 11th IEEE Symp. on Field-Programmable Custom Computing Machines, FCCM 2003, pp. 31-38.

[10] C. R. Clark and D. E. Schimmel, "Scalable parallel pattern-matching on high-speed networks," in Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2004, pp. 249-257.

[11] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in Proc. of the 9th IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2001, pp. 227 - 238.

[12] Y.H.E. Yang, W Jiang, V.K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA", in Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2008, pp. 30–39.

[13] N. Yamagaki, R. Sidhu, S. Kamiya "High-speed regular expression matching engine using multi-character NFA", in Proc. of International Conference on Field Programmable Logic and Applications, FPL 2008, pp. 131-136.

[14] Z. K. Baker, V. K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs", IEEE Transactions on Dependable and Secure Computing, vol. 3, no. 4, pp. 289-300, Oct.-Dec. 2006.

[15] M. Attig, J. Lockwood, "SIFT: Snort Intrusion Filter for TCP", HOTI, Proceedings of the 13th Symposium on High Performance Interconnects.

[16] Haoyu Song, T. Sproull, M. Attig, J. Lockwood, "Snort offloader: a reconfigurable hardware NIDS filter," International Conference on Field Programmable Logic and Applications, FPL 2005, pp.493–498.

[17] S. Teofili, E. Nobile, S. Pontarelli, G. Bianchi, "IDS Rules Adaptation for Packets Pre-filtering in Gbps Line Rates", in Trustworthy Internet, pp. 303–316, Springer, 2011.

[18] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering", in Proc. of the 12th Conference on Field Programmable Logic and Applications, Springer-Verlag, 2002, pp. 452–461.

[19] I. Sourdis, D. N. Pnevmatikatos, S. Vassiliadis, "Scalable Multigigabit Pattern Matching for Packet Inspection", IEEE Transaction on VLSI Systems Vol. 16, no. 2, pp. 156-166, 2008.

[20] I. Sourdis, D. Pnevmatikatos, "Fast, large-scale string match for a 10gbps FPGA-based network intrusion detection system", in Proc. of International Conference on Field Programmable Logic and Applications, FPL 2003, pp. 880-889.

[21] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection" in Proc. of the 15th Int. Conf. Field Programmable Logic Application, FPL 2005, pp. 644-647.

[22] F. J. Burkowski, "A hardware hashing scheme in the design of a multiterm string comparator" IEEE Transaction on Computers, vol. 31, no. 9, pp. 825-834, Sep. 1982.

[23] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of bloom filters for string matching" in Proc. of IEEE Symp. on Field-Programmable Custom Computing Machine, FCCM 2004, pp. 322-323.

[24] D. Markovic, B. Nikolic, R.W. Brodersen, "Power and area efficient VLSI architectures for communication signal processing", in Proc. of the IEEE International Conference on Communications, ICC 2006, pp. 3223 - 3228.

[25] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low-power CMOS digital design," IEEE J. Solid-State Circuits, vol. 27, no. 4, pp. 473-484, 1992.

[26] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection," in Proc. of International Conference on Field Programmable Logic and Applications, FPL 2003, pp. 404-413.

[27] I. Sourdis and D. Pnevmatikatos, “Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching,” in Proc. of IEEE Symposium on Field-Programmable Custom Computing Machine, FCCM 2004, pp. 258-267.

[28] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos and S. Vassiliadis, “Packet Pre-filtering for Network Intrusion Detection”, in 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2006, pp. 183-192.

[29] C. Lin, C. Huang, C. Jiang and S. Chang, “Optimization of Pattern Matching Circuits for Regular Expression on FPGA”, IEEE Transactions on Very Large Scale Integration Systems, vol 15, no 12, pp. 1303-1310, 2007.

[30] C. Greco, E. Nobile, S. Pontarelli, S. Teofili “An FPGA based architecture for complex rule matching with stateful inspection of multiple TCP connections” in Proc. of the VI IEEE Southern Programmable Logic Conference, SPL 2010, pp.119-124.

[31] N. Shenoy, R. Rudell, “Efficient implementation of retiming”, in Proc. of the 1994 IEEE/ACM international conference on Computer-aided design, ICCAD 1994, pp. 226-233.

[32] Baumgartner, J. and Kuehlmann, A., “Min-area retiming on flexible circuit structures”, in Proc. of the 2001 IEEE/ACM international conference on Computer-aided design, ICCAD 2001, pp. 176-182.

[33] XST User Guide, available at http://www.xilinx.com/

[34] LogiCORE IP 10-Gigabit Ethernet MAC v10.1, available at http://www.xilinx.com/

[35] V. Dimopoulos , I. Papaefstathiou , D. Pnevmatikatos ,“A Memory-Efficient Reconfigurable Aho-Corasick FSM Implementation for Intrusion Detection Systems”, Proceedings of 2007 International Conference on Embedded Computer Systems: Architectures, Modelling and Simulation (IC-SAMOS 2007), Samos, Greece, July 16-19, 2007.

[36] PRIvacy-aware Secure Monitoring, available at http://www.fp7-prism.eu/

[37] COMBO Product Brief, available at http://www.invea-tech.com/data/combo/combo_pb_en.pdf

[38] Virtex-5 Family Overview, available at http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[39] S. Pontarelli, C. Greco, E. Nobile, S. Teofili, G. Bianchi, “Exploiting Dynamic Reconfiguration for FPGA based Network Intrusion Detection Systems,” IEEE International Conference on Field Programmable Logic and Applications (FPL), 2010, pp. 10-14.

[40] K. Papadimitriou, A. Dollas, S. Hauck, “Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model,” ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 4, no. 4, December 2011.

Salvatore Pontarelli: Dr. Salvatore Pontarelli received the Master Degree in Electronic Engineering at University of Bologna in 2000. In 2003 he received the PhD Degree in Microelectronics and Telecommunications from the University of Rome Tor Vergata. Currently, he is with the Department of Electronic Engineering at the same university. In the past Dr. Pontarelli has worked with the National Research Council (CNR), the Italian Space Agency (ASI), the National Inter-University Consortium for Telecommunications (CNIT) and has been consultant for various Italian and European companies for projects related to digital design and to fault tolerance in digital systems. Since 2009 he has been working on the development and design of FPGA based methods for high speed network intrusion detection systems. His research activities are mainly focused on : development of highly reliable systems for space applications; Error detection and correction codes; Fault detection and recovery for arithmetic circuits; Use of post-CMOS technologies (in particular Quantum-Dot Cellular Automata) for the implementation of digital circuits at subnanometric integration scale.

Giuseppe Bianchi: Giuseppe Bianchi is Full Professor of Telecommunications at the School of Engineering of the University of Roma Tor Vergata since January 2007. He was formerly employed at Politecnico di Milano and University of Palermo. He spent 1992 as visitor researcher at the Washington University of St. Louis, Missouri, USA, and 1997 as visitor researcher at the Columbia University of New York. His research activity (published on about 170 papers in peer-refereed international journals and conferences) spans several areas, current active topics being wireless networks, network security, network measurement and monitoring, privacy. G. Bianchi has been involved in coordination roles (general, technical or scientific coordinator) for three European STREP projects, one European IP project, and two national PRIN projects, and has participated as unit coordinator to many other projects. He’s area editor of IEEE Transactions on Wireless Communication, editor for IEEE/ACM Transactions on Networking, and area editor for Elsevier Computer Communications. He has chaired more than 10 international IEEE/ACM conferences or workshops.

Simone Teofili: Dr. Simone Teofili received the Master Degree in Electronic Engineering at University of Rome Tor Vergata in 2006. In 2009 he received the PhD Degree in Microelectronics and Telecommunications from the same university. His research interests are focused on Statistical Traffic Analysis countermeasures, Privacy-aware Network Monitoring and FPGA-based Network Processing Applications.