Microprocessors and Microsystems 37 (2013) 1200–1207

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Multi-character cost-effective and high throughput architecture for content scanning ☆

José M. Bande [a,*], José Hernández Palancar [a], René Cumplido [b]

[a] Advanced Technologies Application Center, 7ma A #21406 e/ 214 y 216, CP: 12200 Havana, Cuba
[b] Instituto Nacional de Astrofísica Optica y Electrónica, Luis E. Erro 1, Sta. Ma. Tonanzintla, Puebla 72840, Mexico

ARTICLE INFO

Article history:
Available online 23 August 2013

Keywords:
NIDS
String matching
Content scanning
FPGA
Unique substrings

ABSTRACT

String matching is a time and resource consuming operation that lies at the core of Network Intrusion Detection Systems. In this paper a method and corresponding hardware architecture for string matching is presented. The proposed method is composed of two main steps. The first step performs a pre-detection of signatures alignment, and in the second step the alignment is corrected and the signatures are detected by a matcher. The compact and efficient architecture is designed to share resources among several modules that perform the detection and correction step needed for the string matching. Implementation results in a FPGA Virtex5 device show that the proposed architecture can perform string matching with a database with more than 400 K characters. And is also capable of achieving speeds of more than 30Gbps, which is much higher that previous works reported in the literature.

1. Introduction

Current data transmission technologies are capable of achieving multi-gigabit rates. For instance, recent standard for optical wire technologies OC-768 [1] present a data transmission rate of 40Gbps. In Network Intrusion Detections Systems (NIDS) one of the main tasks is the scanning of the packets content. Thousands of strings declared as signatures of malicious content must be detected in order to identify imminent attacks. And the speed of the data flow on which this string matching operation should be performed is extremely high. In addition, lowering such speed is not a valid option since this would affect the quality of the service provided by the network.

In the most demanding environments, current sequential machine technologies are unable to meet these requirements [2]. Consider the hypothetical situation where a General Purpose Processors (GPP) could process a single character per clock cycle. It would need to be working at 5 GHz in order to achieve 40 Gbps. Obviously, this is not possible with current technologies. This technological barrier dominated by the operating frequency of the sequential devices can be outperformed with an intensive use of parallelism. Here is where the Field Programmable Gates Arrays devices (FPGAs), have played an important role for NIDS in the last decades. The most important advantages of these devices over traditional GPPs are the reconfigurability and the fine grained parallelism they can provide. Reconfigurability, allows updating the signatures set any number of times, and with the use of parallelism, the signatures can be concurrently detected, while one or more than one character is processed at each clock cycle.

Hardware cost reduction and throughput increment, are key issues in string matching through hardware; being the hardware cost reduction the more widely researched. The reason is that augmenting throughput elevates the hardware cost. Therefore, there will be fewer resources available to detect more strings. That is why the most used practice has been to obtain a low cost architecture and replicate it in order to obtain more throughput.

The throughput of a data flow processing hardware architecture is defined as the amount of bits processed per time unit. Basically, there are three strategies to increase the throughput. The first strategy is focused on the frequency, while the other two concern the amount of data processed. Achieving a high operational frequency depends on several factors such as: the complexity of the placed logic, the critical path signals, and the underlying device technology. Well know techniques as pipelining can be applied in order to reduce timing at the cost of introducing latency in the architecture.

The second strategy is works well when the scanned data flow is divided in streams of data chunks or packets. The data packets are distributed into several identical string matching units working in parallel. The overall throughput is the sum of the throughput provided by each unit. This is commonly called aggregated throughput. Due to the packetized nature of some data flows, this form of processing is not very efficient. This is because it is possible

☆ This document is a collaborative effort.

* Corresponding author.
E-mail addresses: jbande@cenatav.co.cu (J.M. Bande), jpalancar@cenatav.co.cu (J. Hernández Palancar), rcumplido@inaoep.mx (R. Cumplido).

0141-9331/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2013.08.001

to have some units with no data to process at all, while there are still data waiting for being processed in other units. This strategy is easy to implement by just replicating as many processing units as possible.

The third strategy consists on augmenting the amount of data symbols processed from the data flow at each clock period. Architectures with this feature are commonly called multi-character architecture. This form is more efficient as it does not underutilize hardware resources and it can be used on either, packetized or continuous data streams. On the other hand, when several characters are processed at each clock cycle, the signatures may appear unaligned regarding the input. The simple approach to face this misalignment problem is a replication of the matching logic linear to the amount of character processed per clock cycle.

The last two naïve strategies to increase the throughput requires resources to invest in hardware replication. In this work we propose a not naïve multi-character approach which reduces considerably the hardware replication. Our architecture solves the misalignment problem in real time. The hardware consumption presented is lower than that obtained in naïve strategies for multi-character architectures. The implementation for four characters per cycle utilizes 63% of a FPGA Virtex5 fx100t device. For a signature set counting more than 84,000 characters the architecture achieves a throughput of 6.4 Gbps. Our strategy is to invest resources in a cost-effective alignment pre-detection and correction phase. In doing so, resources sharing are possible and the replication of the hardware is reduced.

In this work we take leverage of the fact that in a set of strings, each string may have a substring which is unique regarding the rest of the members in the set. The length of the unique substring may be equal or lower than its substring; in case of being equal, the unique substring is the string itself. For the set of signatures included in the most popular Network Intrusion Detection System, Snort, the length of unique substrings tend to be shorter than the length of the container signatures.

The proposed architecture consists of two processing steps. In the first step, the signature unique substrings are matched obtaining the alignment of the candidate signature. A candidate signature indicates a signature likely to exists in the data flow. In the second step the inputs of signature matchers are aligned in correspondence with the information provided by the first step. Since the signature matchers inputs will be aligned with the candidate signature in real time, no hardware replication will be needed in the second step.

We present a method and corresponding hardware architecture for string matching. A previous work that addresses a similar problem was published in [3]. This described an architecture that detects and corrects the signature alignment in the data flow through a unique substring predetection. This work presents a completely new architecture aimed at obtaining a significant reduction in area requirements when compared to the previous reported work. The main contributions of this work are twofold:

- Reduction of area resources of around 40% when compared against the naïve approach for multi-character architecture.
- A new partitioning criterion named security threshold which allows resources sharing and the elimination of ambiguities in matching process due to the misalignment problem.

The rest of the paper is as follows. In Section 2 we expose the works related with special emphasis in multi-character architectures in both, naïve and not naïve approaches. In Section 3, the central ideas and our partitioning scheme are explained in detail. In Section 4, the architecture is presented as well as its functioning. The results of the implementation and tests are discussed in Section 5. Conclusions and future work will be exposed in Section 6.

2. Related work

In general, the most part of hardware-based solutions for string matching fall into the following categories. Brute Force comparators (BF) [4,5,7], Automata based architectures with the Aho-Corasick automaton (AC) as the most implemented [2,3,8,11–13,20]. Content Addressable Memory based (CAM) architectures [6], and Hash based architectures [10]. Naturally, there are combinations of them [22,24,17–19].

Naïve multi-character architectures has been proposed in [4–9]. Sourdis et al. [4] proposed pipelines of discrete characters comparators in order to match an entire string. Their four-character input version was capable of achieving more than 10 Gbps. In a shift-and-compare pipeline, each character line feeds a shift register, by selecting the proper offsets, the character lines are "ANDed" to obtain a final match for a corresponding string. Sourdis et al. [5] proposed a shift-and-compare architecture using SLR16 shift register, while Baker and Prassana in [7] proposed partitioning scheme that allows resource sharing. Sung et al. [6] proposed an algorithm for a CAM memory-based architecture that processes four bytes per cycle, achieving more than 10 Gbps. Clarck and Schiemmel [8] perform a deep analysis of the hardware cost when extending the architecture for processing more bytes per clock cycle. They proposed a NFA logic-based architecture, where they can reach the impressive throughput of 99 Gbps. However, due to the high hardware cost introduced they just match up to 250 characters. Hardware implementation of the shift-or algorithm is proposed in [9]. Processing four character per cycle they achieve 16 Gbps for a 1500 character set.

In [10] the double port feature of modern embedded memory blocks are employed to process 2 bytes per cycle. Their architecture uses hashing in a first step to identify a possible match. Then, in a second step, the string is fetched from memory and is compared one-to-one with the characters in the data flow. Yang et al. [11] present a multi-character logic-based regular expression matching architecture. They propose an algorithm for extending regular expression to multi-character. Additionally, they propose the implementation of character classes operators using embedded RAM. Similarly, in [12] static strings which are part of regular expressions are detected by a classical AC using an off-chip memory, then logic-based NFAs match the regular expression metacharacters. A memory-based multi-character regular expression matching architecture is proposed by Brodie et al. in [13]. They define the concept of Equivalent Class Index, ECI. An ECI represents multiples sequences of characters sharing the same states in a multi-character-extended NFA. Their architecture first encode the data flow into ECI characters, then it uses an state machine to implement the NFA. In addition, they apply several optimizations and codifications for reducing the amount of memory required per NFA state. Yiang et al. [2] Proposed a pipelined and multi-character AC automaton where failure transitions are not needed since each level of the AC trie has its own hardware. A similar approach is proposed in [14], where the firsts states of the AC automaton are implemented using a pipeline of binary search trees.

A not naïve multi-character memory-based architecture is proposed by Cho et al. in [15]. They use the string prefix for pre-detection and alignment correction. Since several strings may have the same prefix, the string set must be partitioned so that each string in a subset possesses a unique prefix. The same strategy is used by in Chang et al. [16] with the difference that instead of memories, and brute force matching, they use logic-based NFA matchers. Serrano et al. [3] proposed the use of unique substrings instead of unique prefixes, since unique substrings tend to be shorter and better suited as partitioning criterion.

The average case throughput is improved in [17,18]. In [17] the authors present an Aho-Corassick architecture where they uses hashing techniques for the state selection. In [18] the most visited states are cached in memory avoiding on average traditional AC next state selection. On this way, these architectures can accept more one character per cycle on average. In [19] AC memory is shared among several string matching modules in a time multiplexed access scheme.

Recent works are more focused in hardware cost reduction than in throughput increment. In the automata based solutions the state cost reduction is an important issue. In [20] AC states with similar transitions are merged. The authors propose a mechanism to efficiently rectify the functional errors caused by the states merging. In doing so a reduction of 24% in the cost is achieved. A perfect hashing technique for indexing the AC states is proposed in [21]. Meiners et al. [22] introduced TCAMs to store RE transitions, the method is patented in [23]. They leverage the ternary nature and the first-matching semantic of the TCAMs to codification the transitions. In doing so, a TCAM entry can be shared by multiples states. Peng improves the state codification in [24] achieving a higher reduction in the number of TCAMs entries. On the same basis, Yun [25], avoid to store AC failure transitions. Guinde and Ziavras [26] proposed a compression method for the string set. In doing so, the memory required for store the strings is significantly reduced. A newly approach is proposed in [27], where the support vector machine theory [28] is introduced for network intrusion detection systems.

3. Partitioning scheme

Our main target is reducing the hardware cost in a multi-character architecture by avoiding the naïve approach efficiently. If we could know the alignment of the signature in real time, the resources used in the replication of the hardware could be released to be used in a less expensive pre-detection phase. Keeping this in mind both, a cost-effective alignment detection phase and a cost-effective alignment correction phase are needed.

The proposed method in this work consists in two partitioning levels. The first one was proposed in [3] and is further explained in Section 3.1. The second one, proposed in this work, works over the sets created by the first partitioning level. This is explained in Section 3.2. The architecture presented is brand new, and it takes advantage of both levels of partitioning.

3.1. Partitioning based on unique substrings

It is straightforward to demonstrate, that a random string set has the following properties.

- Given a set of strings P, a subset $U \subseteq P, |U| \geqslant 1$, can be found such that each string in U will contain at least one substring not repeated in any other string in U.
- If $R = P - U$, then the former property is also valid over R.

By using these properties as partitioning criteria, a partition of P will contain at least one U subset. Let us denote as u-sets all the U subsets of the partition. Each string in a u-set will contain at least one substring which is not repeated in any other signature of the set. Let us denote this unique substrings as u-substring. The u-substring becomes into an exclusive identifier of its corresponding signature. In other words, the detection of such u-substring is a necessary condition for the existence of the signature in the data flow. Therefore, detecting a u-substring imply that the corresponding signature, and no other, is a possible match.

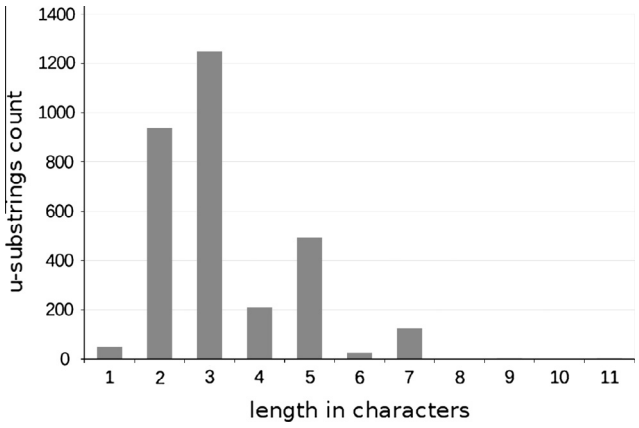


Fig. 1. Histogram of u-substrings in u-substrings length.

The Fig. 1 shows the u-substring distribution over the u-substring length, of the Snort signatures set. Note that most of u-substrings are less than seven character in length, while 68.7% have four characters or less. The shorter the u-substring, the less expensive is the detection. Taking advantage of this fact, matching u-substrings leads to a cost-effective alignment pre-detection phase.

Matching u-substrings is more efficient than matching unique prefixes [15,16]. The reason is that unique prefixes tend to be longer than u-substrings. If two strings have unique prefixes, both common and uncommon characters have to be taken into account. For example the strings "*abcdf*" and "*abtdf*" have "*abc*" and "*abt*" as unique prefixes but "*t*" and "*c*" as unique substrings.

A string of length m has $\sum_{i=0}^{m}(m-i)$ u-substrings and just m prefixes. Since the number of u-substrings contained in a string is greater than the number of prefixes, the probability of find a unique substring is also greater. As a consequence, u-sets will be greater than the sets resulting from a partitioning based on unique-prefixes.

The first step in the architecture construction is to partition the set of signatures into several u-sets. The whole Snort rule data base in its February-2012 release, contains 13,767 distinct statics strings. In the default configuration of Snort, just 3981 of them are used. We have selected this subset of signatures for testing our partitioning scheme. We have found that some signatures are equivalent since its hexadecimal representation is the same. Therefore, we have converted all the signatures to hexadecimal codification and we have eliminated duplicated signatures, resulting in a set of 3928 signatures.

Since we employ 6-input LUTs as comparators, the cost per signature of our architecture is lower than the naïve approach only for those signatures greater than seven characters. In correspondence, we first extract all the signatures of seven characters or less from the original set resulting in a subset of 819 short signatures. This represents the 20.8% of the overall set. The short signatures are matched in a naïve approach.

The longer signatures, counting 3109 are partitioned according the unique substrings criteria. In Table 1 the resulting partitions are represented. Note that most of the signatures are grouped in

Table 1
Number of strings on each u-set.

U-partition	Strings
1	2937
2	138
3	16
4	4
Total	3095

the first u-set and the size of the following u-sets decrease exponentially.

In Table 1 just those u-sets counting with more than one string are represented. Note that the total number of signatures differs from the initial set in 14 signatures. Actually, there are 14 single-string u-sets. In Section 4 it will be demonstrated how resources can be shared on each u-set. These single-signature u-sets represents no resources savings in our architecture. Therefore, for simplicity they are implemented as simple multi-character logic-based NFAs [8].

3.2. Partitioning based on security threshold

In order to have an errors-free alignment correction phase, it is mandatory to have just one u-substring match at the time. Meaning that, just one signature of the u-set can be candidate to be matched against the data flow. This is not a problem when just one character is accepted per clock cycle. But in multi-character processing, there are some conditions where more than one u-substrings is matched. In order to eliminate such conditions we propose a second partitioning level. The additional partitioning eliminates ambiguities in the matching process.

The Fig. 2a shows two consecutive signatures, "*abcdt*" and "*abedt*", transiting by a four character pipeline. These signatures have u-substring "c" and "e" respectively. The Fig. 2a also shows the lines for detecting u-substrings at any misalignment. The shadow square represents a matched u-substring.

There are two possible scenarios: (a) Two subsequent strings are candidate, i.e. are members of the same u-set, Fig. 2a. (b) One or both signatures are impostors; meaning that they contain some u-substrings but they are not signatures. See Fig. 2b. In the first case just one u-substring will match at the time. In the second case there is a double matching. See in Fig. 2b that the u-substrings "e" and "c" match in the same clock cycle, but just the first string is a signature. In this case it is impossible to predict which of both matching results is a candidate signature if any. The situation would be the same if the impostor string were the last one.

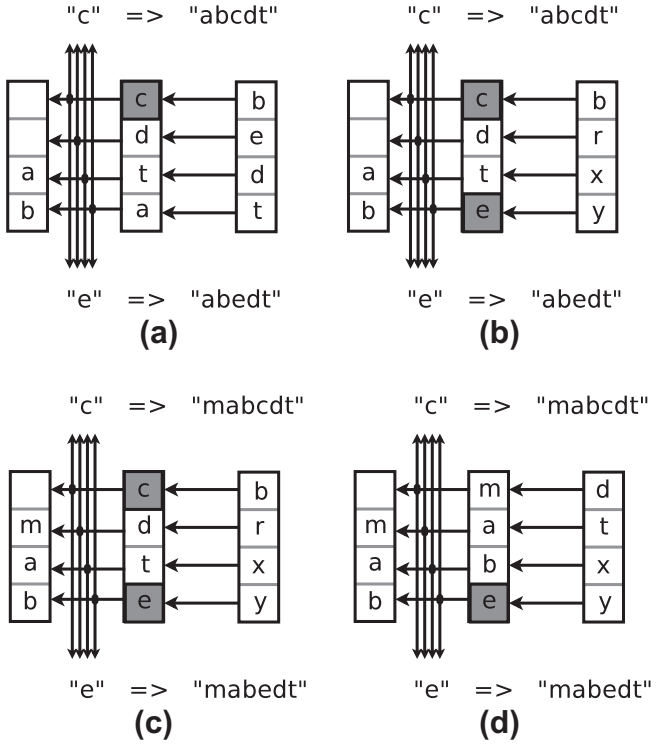


Fig. 2. (a,d) One unique substring detected, one candidate signature. (b) Two unique substrings detected, both signatures are candidate. (c) Two unique substrings detected, the left-most signature is candidate.

Now let us consider another couple of signatures as example. For simplicity, let us use the same signatures but with the character "*m*" added at the beginning. Additionally, in case of double matching let us assume the first string as the candidate and the last string as the impostor. For the signatures in Fig. 2c, double matching is only possible when both are impostors, or when the first one is the candidate and the second one is impostor. A double matching never happens when the first one is impostor and the second one is candidate, Fig. 2d. In other words for this couple of signatures a double matching indicates that just the first one in the sequence is the candidate one.

Let us denote as q the amount of characters read per clock cycle. We define the function $liu()$ for computing the distance in characters between two u-substrings u_1,u_2, of two different consecutive signatures p_1 and p_2. For instance, with $u_1 = c$ and $u_2 = e$, liu("*mabcdt*","*mabedt*") = 5. For any pair of signatures the function $liu()$ can be calculated in two ways, $liu(p_1,p_2)$ and $liu(p_2,p_1)$, and the results are not necessarily equal. For that reason we define the security threshold as, $save_{th}(p_1,p_2) = min(liu(p_1,p_2), liu(p_2,p_1))$. Our second partitioning level is based on the following observation.

- If $save_{th}(p_1,p_2) > q$ and u_1 and u_2 match at the same moment. Just p_1 is the candidate string.

Let us define a partition S over an u-set, such that for each pair of strings x, $y \in s$, $s \in S$, $save_{th}(x,y) > q$. We name the S partition, security threshold based partition.

The next step in the construction of our architecture is the partitioning of the u-sets by applying the security threshold criteria. We have implemented an algorithm that computes the $save_{th}$ of any couple of strings in a set. Once the algorithm is executed, each u-set is divided into two sets, t-set and h-set. The h-set accumulate signatures with large prefixes, taking as prefix the characters from the beginning of the string to the beginning of the u-substring. The t-set accumulate signatures with large suffixes, taking as suffix the characters from the beginning of the u-substring to the end of the string. The algorithm also returns the signatures not compliant with the security threshold criteria, usually a little portion of the overall signatures. In the same way we proceeded with the single-string u-sets, this outlier signatures will be detected using a naïve multi-character architecture.

In Table 2 the resulting partitions from applying the security threshold criteria over u-partitions are presented. The subindex indicates the u-sets they come from, and the letter indicates whether they are h-set or t-set. The number of signatures not satisfying the security threshold criteria sum 52, being the 1.7%.

The security threshold partitioning has several advantages. Since it warranties that in a (h) t-set just one signature will be candidate for a matching each time; an important portion of the resources dedicated for the alignment correction can be temporary shared. This is the main source of efficiency of our architecture.

Table 2
Partition based on security threshold.

Partition	Signatures	Chars.
h_1	1289	45841
t_1	1604	36301
h_2	35	689
t_2	96	1038
h_3	3	42
t_3	12	123
t_4	4	39
Total	3043	84073

Moreover, no extra resources are needed to eliminate ambiguities in matching process. Because of the whole string set is divided into several sets, long paths signals and signal fan-out are reduced, thus improving the operating frequency. The architecture that shares resources on each partition is further explained in the next section.

4. Architecture

Each (h) t-set resulting from the partitioning algorithm is implemented as an independent hardware module called, matching engine. Fig. 3 illustrates the matching engine functional blocks. In the first block, u-substrings are matched in a naïve fashion. This is a shift-and-compare like architecture [7]. It follows, the alignment codification block. This encodes from one-hot to binary the result from the u-substring matching block, returning the alignment of the candidate signature to the next block. The third block is the alignment correction module. Taking benefit our partitioning properties, the resources invested in this block are shared by the rest of the architecture. The next block is the signature matching module. Here the signatures are matched by pipelined discrete comparator modules. Each comparator can match up to six characters lines. In this block no hardware replication is needed because the signatures are matched as if it were a single-character architecture. Finally, a pipelined encoder converts the signature matching results into unique identifiers.

4.1. Alignment detection phase

Fig. 5 shows an example of our alignment detection phase. At the input, each character is decoded into a 256 bit vector having one bit signal per character. There are four binary to one-hot decoders, one for each character input.

The character lines feed a pipeline of registers. This pipeline is shared by all the matching engines. The u-substrings are matched by "ANDed" the character lines in the proper offsets. This technique is known as shift-and-compare. Different to traditional shift-and-compare approaches where the strings are matched as soon as possible, we define a common stage in the pipeline from which all the character lines for matching the u-substrings are connected. The dashed line in the Fig. 5 represents the common stage. This is calculated for a matching engine by adjusting all its signatures as depicted Fig. 4. See in Fig. 4 that the signatures are aligned according to the u-substring (cells in gray color) locations.

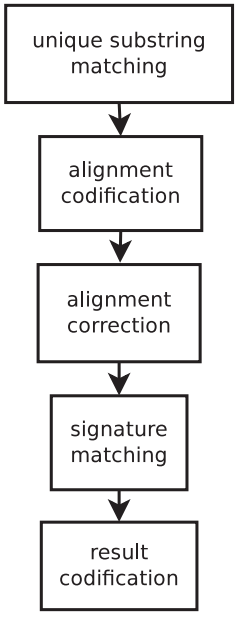


Fig. 3. Functional blocks.

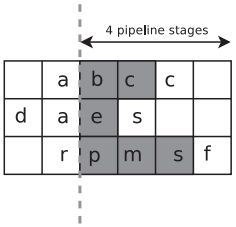


Fig. 4. Character matrix.

The Fig. 4 shows a matrix of characters where each row contains a signature. The signatures has been properly aligned such that the first characters of all u-substrings fit in the same column. The arrow marks the number of pipeline stages before the common stage where the matching logic is placed. In doing so, just one match result is obtained at once.

Each u-substring matcher detects four version of the same u-string; one for each possible alignment. Therefore, four one-bit outputs tells the next block which is then alignment of the candidate signature. Seeing that all the string matchers works in parallel, just the alignment of the signature is needed. Because of that, the outputs of the all u-substring matchers representing the same alignment are "ORed" together. In order to obtain low timing this OR functions are implemented as pipelined trees of LUTs. At the end, the result is encoded by a high priority one-hot to binary encoder. This provides to the alignment correction phase with the alignment of some candidate signature.

Although in the first part of the architecture we use a naïve approach to detect the u-substrings, their short length lowers the hardware cost. Furthermore, with the overhead introduced by the alignment codification logic, the cost is still lower than a full-string-matching shift-and-compare approach.

4.2. Signature matching phase

In the signature matching phase, the input character lines are selected according to the detected alignment. Then the signature matchers perform the matching operation over the corresponding character lines. Finally, the match result is encoded into a binary identifier which represents the detected signature.

The Fig. 6 shows the character matrix module. This performs the alignment correction. Each matrix element corresponds to a character, and is implemented as a single-LUT multiplexor. Character matrix component follows the signature set representation as proposed in Fig. 4. In figure Fig. 6 note that multiplexors in dashed line correspond to u-substring characters. A multiplexers column corresponds to a character position in the signatures. Each four-to-one multiplexor selects one of the four character signals from the pipeline, according to the current alignment.

In large string sets it is very common signatures sharing characters in the same locations. For instance, in Fig. 4, in the second column from left to right, the "a" character is repeated. In consequence, we can share the character multiplexors among signature matchers. Note that in Fig. 5 just one multiplexor for the "a" character is needed. This resource sharing is possible because, our partitioning eliminates ambiguities in the matching process. Meaning that we will always have just one valid, and likely, match option. Hence, all the circuitry can be temporary configured to respond accordingly.

The signatures matchers are 6-input LUTs oriented. Each LUT acts as a discrete comparator capable of matching up to six characters. Pipeline stages are introduced as needed for long signatures. For example a 12 characters signature would need two pipeline

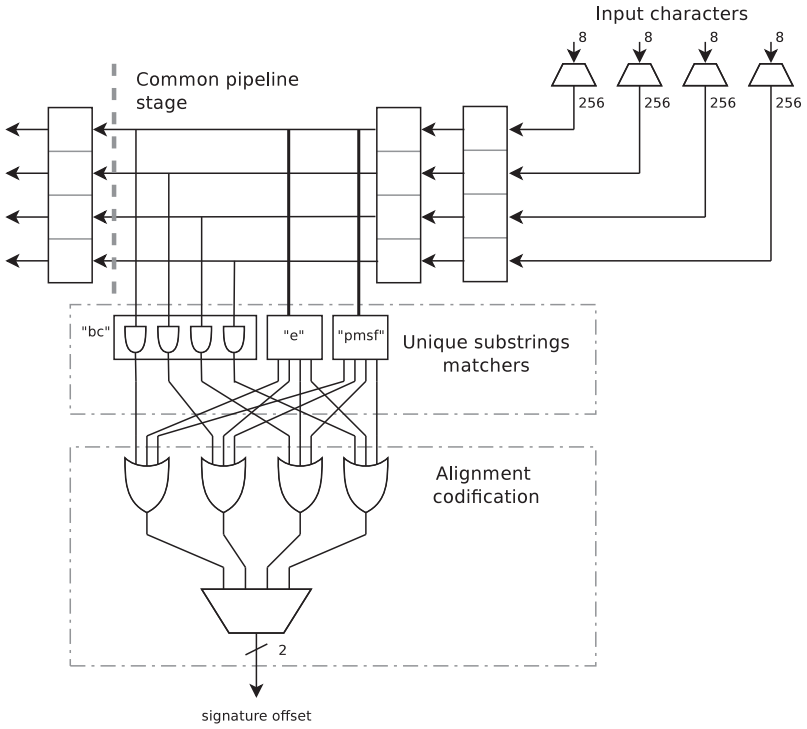


Fig. 5. Alignment detection phase.

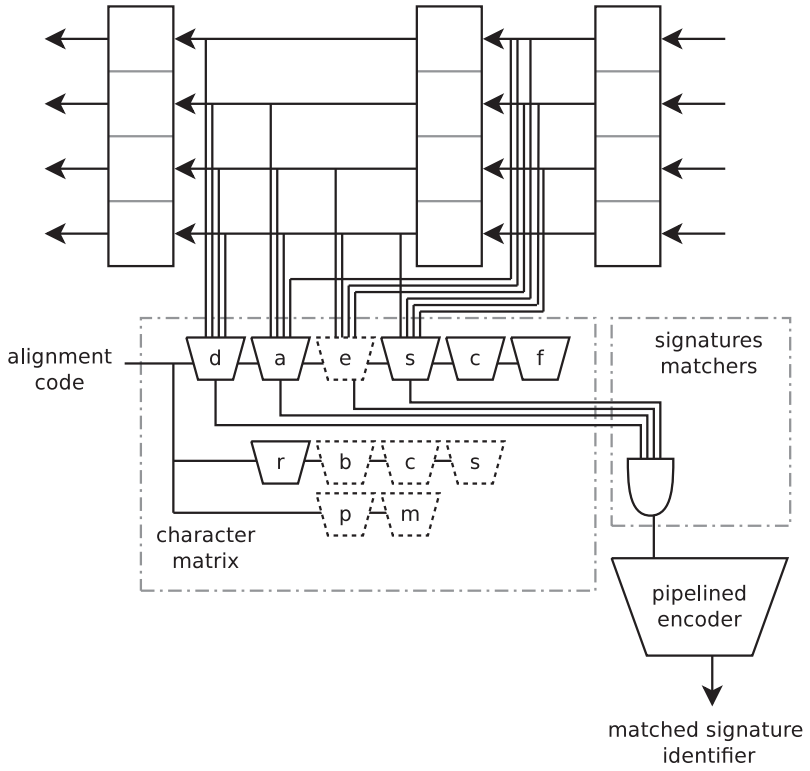


Fig. 6. Signatures matching phase.

states, two 6-input LUTs and one 2-input LUT. All signatures are matched parallelly, taking few clock cycles to get the result depending on the signature length.

The final component is a pipelined one-hot to binary encoder. This is a commonly used technique to encode long one-hot vectors with reduced timing. With 6-input LUTs, a few pipeline stages are needed to obtaining the identifier of a matched signature.

In the proposed architecture the input pipeline and the input characters decoders are shared by all the matching engines. Additionally, on each matching engine the character matrix module is

Table 3
Implementation results.

Strings	Chars.	LUTs	FFs	LUT/char	FF/char	Naive est. LUT/char	Device usage (%)	Thput. (Gbps)
1446	21,591	21,572	18,151	0.84	0.90	1.12	33	8.3
3095	84,403	38,598	40,329	0.45	0.47	1.03	63	6.4

shared among the signature matchers. In the next section, the implementation results of the proposed architecture are discussed.

5. Implementation, results, and comparisons

We have developed a python program to autogenerate all the architecture VHDL source code from the Snort rules set. The architecture was synthesized and implemented using Xilinx ISE 13.1. After successfully simulation it was configured into a Virtex-5 FX100T counting with 64,000 LEs. In Table 3, the results of two implementations are presented. The first implementation can recognize 1445 signatures from the Snort signatures set. For the second, the number of signatures was incremented to 3095. The first implementation counting with 21,591 characters occupy 33% of the device. Meanwhile, the second with almost four times more characters than the first, barely duplicates the mentioned device occupation percentage. This is product of the resources sharing provided by our architecture.

The implementation counting with 84,403 characters occupies 63% of the device logic. Since the registers utilization is greater than the LUTs utilization, this number, represents the hardware use in terms of registers.

It is notable the per-character cost of the architecture. The column labelled as, naïve est. shows the estimated cost of the architecture in a naïve approach. On the other hand, the columns LUTs/char and FF/char contains the cost of our architecture as a whole. For the first implementation the per-character cost is very close to the naïve approach. While for the second, the cost of 0.47 LUTs/char represents a reduction of more than 50% regarding to the naïve approach. This means, that our architecture is more efficient in the signature set increment.

When the second implementation is synthesized for a Virtex5 LX330 as target device, the hardware utilization is just 19% of the device. So, it could be up to five times replicated, achieving an aggregated throughput of 34 Gbps. Otherwise, different sets of similar size can be placed together with an estimated character capacity over 400 K characters.

Also note that both implementations may fit in this device. Working in parallel, they could match a string set of 105,994 characters. Different configurations of string sets can be tested, looking for a best throughput and set size ratio. For example two instances of the second implementation working in parallel could obtain an aggregated throughput of 16.6 Gbps.

In Table 4 comparisons with other approaches are offered. When compared with other architectures having the same input width. Our approach presents the best per character cost.

Another relevant aspect is that our approach presents a well balance between character capacity and throughput. Consider that our architecture doubles the throughput of the approach with highest character capacity [14]. Meanwhile, it has 27 times the character capacity of the architecture with higher throughput [9]. As already mentioned, in devices with higher resources available, the architecture could be replicated in many ways depending on the primary interest, high character capacity or elevated throughput.

For real testing we have used a Virtex5 FX100T placed on the development kit HTG-V5-PCIE-DDR3 [29]. A PCIExpress end-point was also implemented and properly connected with the architecture. The working flow starts in an input buffer which feeds the architecture with the packets to be analyzed. During the processing, an output buffer saves the identifiers of signatures founded in the packet. The result is then transferred to the PC. The transference between hardware accelerator and the PC is held through Direct Memory Access (DMA) technique. Randomly generated packets containing some signatures were passed to the architecture, all the signatures were successfully detected. Running at 125 MHz the throughput of the architecture was of is 4 Gbps.

6. Conclusions and future work

The main objective of this work is to lower the resources requirements when more than one character are processed per clock cycle. The motivation is to obtain high throughput and high character capacity. To meet this objective, we proposed a not naïve architecture, meaning that we avoid as much as possible the hardware replication as a choice for elevating the throughput. We chose instead, a cost-effective string alignment detection phase and an efficient signature matching phase working together. To attain this objective we propose a two partitioning levels that combined allow resource sharing. In doing so a reduction of more than 50% in the hardware cost regarding the naive approach is achieved. The proposed partitioning criteria were, the unique substrings and the security threshold. They allow to group a large amount of signa-

Table 4
Comparisons with other works.

Approach	Device	Input width	Chars.	LE (char.)	Bit (char.)	Thput. (Gbps)
Our approach	virtex5FX100T	32	84403	0.47	0	6.4
			21592	0.84	0	8.3
Baker-Prasanna [7]	virtex2P100	32	19584	0.65	0	7.3
Cho et al. [15]	Spartan3-2000	32	19021	1.4	0	8
Hwang et al. [9]	StratixERS140	32	3028	1.5	0	11.6
Serrano and Palancar [3]	virtex5FX100T	32	20235	1.65	0	3.26
Chang et al. [16]	virtex5LX85T	32	16028	1.89	0	7.27
Yang et al. [11]	virtex4LX40	32	15000	2.2	0	7.46
Sourdis et al. [5]	virtex2-6000	32	17592	3.56	0	9.7
Sourdis et al. [4]	virtex2-6000	32	2457	19.4	0	8
Tseng et al. [18]	virtex2P	–	25642	n/a	n/a	10.5
Kennedy et al. [19]	Stratix	16	109467	0.63	61	7.4
Yang et al. [14]	virtex5LX330	16	>700 K	n/a	n/a	3.2
Lin and Chang [20]	n/a	8	36359	n/a	4	4
Guinde and Ziavras [26]	virtex2P70	8	105763	0.052	17.7	2.4

tures in a few partitions where the resources are shared, at the same time, several characters are accepted as input.

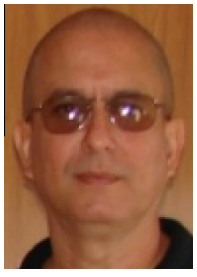
It was also demonstrated that our architecture is more efficient on large signatures set, presenting a reduction of the hardware cost per character, while the set size increase. Due to the reduced hardware cost several instances of the architecture can be placed together to match up to more than 400 K characters for larger devices, while attaining an elevated throughput. It is also possible to increase the aggregated throughput by replicating several times the same architecture.

As future work we study the addition of memory based string matchers to work in parallel with the logic-based ones. On this way to exhaustively use this two kind of resources, i.e. Logic Cells and Memory Blocks, in order to achieve higher character capacity.

References

[1] SONET:www.sonet.com.
[2] W. Jiang, Y.-H. Yang, V. Prasanna, Scalable multi-pipeline architecture for high performance multi-pattern string matching, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–12.
[3] J.M.B. Serrano, J.H. Palancar, String alignment pre-detection using unique subsequences for FPGA-based network intrusion detection, Computer Communications 35 (6) (2012) 720–728.
[4] I. Sourdis, D. Pnevmatikatos, Fast- large-scale string match for a 10 Gbps FPGA-based network intrusion detection system, in: P.Y.K. Cheung, G. Constantinides (Eds.), Field Programmable Logic and Application, Lecture Notes in Computer Science, vol. 2778, Springer, Berlin/Heidelberg, 2003, pp. 880–889.
[5] I. Sourdis, D. Pnevmatikatos, Pre-decoded cams for efficient and high-speed NIDS pattern matching, in: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 258–267.
[6] J. Sung, S. Kang, Y. Lee, T. Kwon, B. Kim, A multi-gigabit rate deep packet inspection algorithm using TCAM, in: Global Telecommunications Conference, GLOBECOM'05, vol. 1, IEEE, 2005, p. 5.
[7] Z.K. Baker, V.K. Prasanna, Automatic synthesis of efficient intrusion detection systems on FPGAs, IEEE Transactions on Dependable ane Secure Computing 3 (4) (2006) 289–300.
[8] C.R. Clark, D.E. Schimmel, Scalable pattern matching for high speed networks, in: FCCM 2004: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, Los Alamitos, CA, USA, 2004, pp. 249–257.
[9] W. jyi Hwang, C.-M. Ou, Y.-N. Shih, C.-T.D. Lo, High throughput and low area cost FPGA-based signature match circuit for network intrusion detection, Journal of the Chinese Institute of Engineers 32 (3) (2009) 397–405.
[10] I. Sourdis, D. Pnevmatikatos, S. Wong, S. Vassiliadis, A reconfigurable perfect-hashing scheme for packet inspection, in: Proceedings of 15th Int. Conf. on Field Programmable Logic and Applications, 2005, pp. 644–647.
[11] Y.-H.E. Yang, W. Jiang, V.K. Prasanna, Compact architecture for high-throughput regular expression matching on FPGA, in: ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM, New York, NY, USA, 2008, pp. 30–39.
[12] Hiroki Nakahara, Tsutomu Sasao, Munehiro Matsuura, A regular expression matching circuit: decomposed non-deterministic realization with prefix sharing and multi-character transition, Microprocessors and Microsystems 36 (8) (2012) 644–664.
[13] B.C. Brodie, D.E. Taylor, R.K. Cytron, A scalable architecture for high-throughput regular-expression pattern matching, SIGARCH Computer Architecture News 34 (2) (2006) 191–202.
[14] Y.-H.E. Yang, H. Le, V.K. Prasanna, High performance dictionary-based string matching for deep packet inspection, in: Proceedings of the 29th Conference on Information Communications, INFOCOM'10, IEEE, 2010, pp. 86–90.
[15] Y.H. Cho, W.H. Mangione-Smith, Deep network packet filter design for reconfigurable devices, ACM Transactions on Embedded Computing Systems 7 (2) (2008) 1–26.
[16] Y.-K. Chang, C.-R. Chang, C.-C. Su, The cost effective pre-processing based NFA pattern matching architecture for NIDS, in: Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications, AINA '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 385–391.
[17] K.-K. Tseng, Y.-C. Lai, Y.-D. Lin, T.-H. Lee, A fast scalable automaton-matching accelerator for embedded content processors, ACM Transactions on Embedded Computing Systems 8 (3) (2009) 19:1–19:30.
[18] K.-K. Tseng, Y.-L. Lai, C.-C. Chen, C.-Y. Hsu, A fuzzy-updated cache of automaton-matching for embedded network processor, Journal of Circuits, Systems, and Computers (JCSC) 20 (3) (2010) 401–415.
[19] A. Kennedy, X. Wang, Z. Liu, B. Liu, Ultra-high throughput string matching for deep packet inspection, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2010, pp. 399–404.
[20] C.-H. Lin, S.-C. Chang, Efficient pattern matching algorithm for memory architecture, IEEE Transactions on Very Large Scale Integration Systems 19 (1) (2011) 33–41.
[21] Y. Xu, L. Ma, Z. Liu, H.J. Chao, A multi-dimensional progressive perfect hashing for high-speed string matching, in: Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, ANCS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 167–177.
[22] C.R. Meiners, J. Patel, E. Norige, E. Torng, A.X. Liu, Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems, in: Proceedings of the 19th USENIX Conference on Security, USENIX Association, Berkeley, CA, USA, 2011, p. 8.
[23] X.A. Liu, C.R. Meiners, E. Torng, Regular Expression Matching Using TCAMs for Network Intrusion Detection, US Patent 20,120,072,380, March 2012.
[24] K. Peng, Q. Dong, M. Chen, Tcam-based DFA deflation: a novel approach to fast and scalable regular expression matching, in: Proceedings of the Nineteenth International Workshop on Quality of Service, IWQoS '11, vol. 13, IEEE Press, San Jose, California, Piscataway, NJ, USA, 2011, pp. 13:1–13:3.
[25] S. Yun, An efficient TCAM-based implementation of multipattern matching using covered state encoding, IEEE Transactions on Computers 61 (2) (2012) 213–221.
[26] N.B. Guinde, S.G. Ziavras, Efficient hardware support for pattern matching in network intrusion detection, Computers & Security 29 (7) (2010) 756–769.
[27] Y.-H.C.T.-W.K.R.-J.C.J.-L.L.C.D.P. Shi-Jinn Horng, Ming-Yang Su, A novel intrusion detection system based on hierarchical clustering and support vector machines, Expert Systems with Applications 38 (2011) 306–313.
[28] V. Vapnik, The Nature of Statistical Learning Theory, Springer Verlag, New York, NY, 1995.
[29] www.hitechglobal.com, 2012.

José M. Bande received the B.Eng from Telecommunications and Electronics Engineering at the Jose Antonio Hechevarria University, (CUJAE), in 2009. He is currently a PhD student in the Advanced Technologies Applications Centre (CENATAV). His research interests include High Performance Computing and Reconfigurable Computing.



José Hernández Palancar works in the Advanced Technologies Application Center (CENATAV) from 2003, place where he is a Senior Researcher and Deputy Director for Applied Research, before he was the Head of Data Mining Department. In CENATAV his research interests focus on Parallel Processing applied to Data Mining and Pattern Recognition algorithms and Biometric.

René Cumplido received the B.Eng. from the Instituto Tecnologico de Queretaro, Mexico, in 1995. He received the M.Sc. degree from CINVESTAV Guadalajara, Mexico, in 1997 and the Ph.D. degree from Loughborough Univerity, UK in 2001. Since 2002 he is a professor at the Computer Science Department at INAOE in Puebla, Mexico. His research interests include the use of FPGA technologies, custom architectures and reconfigurable computing applications. He is co-founder and Chair of the ReConFig international conference and founder editor-in-chief of the International Journal of Reconfigurable Computing. He also serves as associate editor of several international journals.