A Novel Scalable and Storage-Efficient Architecture for High Speed Exact String Matching

Ali Peiravi and Mohammad Javad Rahimzadeh

String matching is a fundamental element of an important category of modern packet processing applications which involve scanning the content flowing through a network for thousands of strings at the line rate. To keep pace with high network speeds, specialized hardware-based solutions are needed which should be efficient enough to maintain scalability in terms of speed and the number of strings. In this paper, a novel architecture based upon a recently proposed data structure called the Bloomier filter is proposed which can successfully support scalability. The Bloomier filter is a compact data structure for encoding arbitrary functions, and it supports approximate evaluation queries. By eliminating the Bloomier filter's false positives in a space efficient way, a simple yet powerful exact string matching architecture is proposed that can handle several thousand strings at high rates and is amenable to on-chip realization. The proposed scheme is implemented in reconfigurable hardware and we compare it with existing solutions. The results show that the proposed approach achieves better performance compared to other existing architectures measured in terms of throughput per logic cells per character as a metric.

Keywords: String matching, content scanning, Bloomier filter, Pearson's hash.

Manuscript received June 20, 2008; revised July 12, 2009; accepted July 28, 2009.

Ali Peiravi (email: ali_peiravi@yahoo.com) and Mohammad Javad Rahimzadeh (phone: +985118815100 ext 653, email: mjrahimzadeh@yahoo.com) are with the Department of Electrical Engineering, Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad, Iran.

doi:10.4218/etrij.09.0108.0353

I. Introduction

Several modern network processing applications need to analyze network traffic and deeply scan packet contents. Network security applications, such as intrusion detection systems or antivirus scanners, which have become essential tools for protection of networks, search for thousands of predefined signature strings which belong to known attacks, worms, and viruses in the packet payloads. For instance, a popular network intrusion detection system, Snort, contains 5,076 patterns that average 13 bytes each [1]. The most up-to-date "ClamAV" virus database contains about 90,000 patterns, most of which are over 100 bytes long [2].

At the heart of most of such applications, there exists a string matching engine capable of checking every byte of every packet to analyze whether the packet contains the target strings. As network speeds increase, matching against thousands of string characters at line rate becomes a computationally intensive task which critically affects the overall system performance. In the case of applications like intrusion detection systems, a string matching engine must provide worst-case performance to stand up to performance-based attacks. Therefore, there is a great need for specialized and efficient high performance hardware-based solutions which are scalable enough to handle a large number of strings with reasonable resource requirements and can operate at high speeds.

Up to now, several interesting hardware-based string matching techniques for networking applications have been developed. Most of these techniques have been developed for network intrusion detection and use the reconfigurable logic resources and embedded memories available on a field programmable gate array (FPGA) to build a high-speed string matching engine. However, many of them suffer from lack of

 © 2009 ETRI

scalability in terms of either speed or the number of strings.

In this paper, a new architecture is proposed which is scalable in terms of network speed and the number of strings. The principle contribution of this work is to provide a high performance space-efficient architecture which is an exact string matching solution based upon a recent data structure called the Bloomier filter, which was introduced by Chazelle and others [3]. The Bloomier filter is a generalization of the Bloom filter [4] and it can encode arbitrary functions and support approximate evaluation queries. Like Bloom filters, Bloomier filters achieve a small space overhead by accepting a small probability of false positives; therefore, all potential matches must be verified by exact matching. However, Bloomier filters are more powerful than Bloom filters because they can indicate which strings are candidate matches. Because only one suspect string should be checked instead of all strings, finding an exact match using a Bloomier filter is much faster than using a Bloom filter. By eliminating the Bloomier filter's false positives in a space efficient way, a simple yet powerful architecture which can support exact matching of several thousand strings at more than 2 Gbps processing a single input character per cycle with the help of several on-chip memory blocks can be constructed. This architecture is amenable to on-chip realization and can fit entirely on a single FPGA device in an efficient way with no need for off-chip memory access. Staying on-chip for the search process results in a simpler design with no off-chip bandwidth, small pin requirements, and reduced power consumption during operation. Moreover, since the proposed architecture is based upon a simple memory-efficient data structure, adjustment to new string sets is achieved simply by updating the memory locations, which can be done without a temporary loss of service.

The rest of the paper is organized as follows. In the next section, related works in hardware-based string matching are summarized. In section III, Bloomier filter theory is reviewed. In section IV, the proposed architecture is described, and some important details of the hardware implementation are considered in section V. The results obtained and a comparison with other related works are presented in section VI. Finally, we conclude the paper in section VII.

II. Related Works

In recent years, several techniques have been proposed for string matching, especially in the context of network intrusion detection. Most hardware-based techniques use reconfigurable logic/FPGAs. Some of the FPGA-based techniques use on-chip logic resources to compile strings into parallel state-machines or combinational logic. Although these techniques are fast, they are known to use up most of the chip's resources with just a few thousand strings, thereby limiting the number of strings that can be processed. Hence, scalability with string set size is the main concern with purely FPGA-based approaches.

From the scalability viewpoint, memory-based techniques are attractive because memory chips are cheap. However, memory access speed turns out to be a bottleneck when memory-based techniques are used. An optimized hardware-based Aho-Corasick algorithm was proposed by Tuck and others [5]. This algorithm relies on run length encoding and/or bit-mapping adapted from similar techniques used to speed up IP-lookup. Although the algorithm is very fast even in the worst case (8 Gbps scanning rate), it assumes the availability of an excessively large memory bus, such as 128 bytes, to eliminate the memory access bottleneck and suffers from excessive power consumption.

The Bloom-filter-based algorithm proposed by Dharmapurikar and others [6] makes use of a small amount of embedded-memory along with off-chip memory to scan a large number of strings at a high speed. Upon the approximate matching done using on-chip Bloom filters, the presence of the string is verified by using a hash table in the off-chip memory. They argue that since the strings of interest are rarely found in the packets, the quick check in the Bloom filter reduces more expensive memory accesses and greatly improves the overall throughput. However, their work does not offer worst case guaranteed bandwidth.

An algorithm presented by Dharmapurikar and Lockwood [7] modifies the classic Aho-Corasick algorithm to allow it to consider multiple characters. This modification allows parallelization of the Aho-Corasick algorithm and the use of multiple instances of it to achieve the required speedup by advancing the text stream by multiple characters at a time. With the help of Bloom filters in implementing the automation, off-chip memory access can be suppressed to a great extent. However, when a string of interest appears in the text stream, their machine should still perform the necessary memory accesses, and this slows down the string matching process.

Tan and Sherwood [8] proposed a technique to reduce the out-degree of string-matching state machines. This innovative technique allows state machines to be represented using significantly less state memory than would be required in a naive implementation. Through the use of bit-splitting, a single state machine is split into multiple machines, each of which handles some fraction of the input bits. The earlier work did not provide many of the requirements for a realistic implementation. Jung and others [9] adapted the basic architectural design proposed by Tan and Sherwood [8] to an FPGA implementation. They showed that by considering FPGA implementation details, such as restriction of a block RAM size, pin count, and the problem of routing delay, this

architecture does not use on-chip memory efficiently and wastes a high percentage of on-chip block RAM.

The techniques proposed by Sourdis and others [10] and Papadopoulos and Pnevmatikatos [11] seek to combine hashing and use of memory to create a cost-effective exact string matching solution. However, they need to use additional indirection memories to reduce the sparseness of the memory and compact storage of the search strings because of their hashing schemes.

III. Bloomier Filter Theory

The proposed string matching architecture is based upon the Bloomier filter introduced by Chazelle and others [3], which is a generalization of Bloom filter. Specifically, the Bloomier filter computes arbitrary functions defined only in a small subset *S* of size *n* taken from a universal set *U* of size *N*. For an arbitrary function *f* defined over a subset *S*, the filter always returns *f*(*x*) if $x \in S$. Otherwise, the filter will return null with a high probability. Just as a Bloom filter can have false positives, a Bloomier filter can return a non-null value for an element not in the set.

The data structure consists of *m* *q*-bit locations and is addressed by *k* hash functions, where *m*, *q*, and *k* are design parameters. This data structure is called the lookup table. Also, the *k* hash values of an element *x*, $\{h_1, \cdots, h_k\}$, where $0 \le h_i \le m$, are collectively referred to as its neighborhood, denoted by *N*(*x*).

The lookup table is constructed so that for every *x*, a location *τ*(*x*) among *N*(*x*) can be found in such a way that there is a one-to-one mapping between all *x* and *τ*(*x*). Because *τ*(*x*) is unique for each *x*, collision-free lookups are guaranteed. In section IV.1, the way to tune the design parameters such that the probability of finding such a mapping is arbitrarily close to 1 is described.

The aim is to construct the lookup table such that a lookup for *x* returns *τ*(*x*). Then, we can store *f*(*x*) in another data structure called the result table at address *τ*(*x*). Thus, we can guarantee deterministic, collision-free lookups of arbitrary functions. Since the result table is addressed by *τ*(*x*), it must have as many locations as the lookup table.

During the lookup table setup, once we find *τ*(*x*) for a certain *x*, we store *V*(*x*) in the location *τ*(*x*), which is computed as

$$V(x) = \left(\bigoplus_{i=1, i \ne \hbar_\tau(x)}^{k} D[h_i(x)] \right) \oplus \hbar_\tau(x), \qquad (1)$$

where $\oplus$ denotes the XOR operation, *k* is the total number of hash functions, $h_i(x)$ is the *i*-th hash value of *x*, $D[h_i(x)]$ is the data value in the $h_i(x)$ location of the lookup table, and $\hbar_\tau(x)$ identifies which hash function produces *τ*(*x*). Now, during a lookup for *x*, $\hbar_\tau(x)$ can be computed as

$$\hbar_\tau(x) = \bigoplus_{i=1}^{k} D[h_i(x)]. \qquad (2)$$

We can use $\hbar_\tau(x)$ to obtain *τ*(*x*). This can be done by recomputing the $\hbar_\tau(x)$-th hash function or by remembering all hash values and selecting the $\hbar_\tau(x)$-th one. Then, we can read *f*(*x*) from the *τ*(*x*) location of the result table.

IV. Description of the Proposed Architecture

In this section, the proposed string matching architecture is described. First, we consider the issue of tuning the filter design parameters. Next, we describe the way the proposed architecture addresses false positives. Then, we investigate the scalability of the proposed architecture as the number of strings increases. Finally, we present an overview of the proposed architecture.

1. Tuning the Design Parameters

Chazelle and others [3] have shown that for a Bloomier filter with *k* hash functions, *n* elements, and a lookup table size $m \ge kn$, the probability of setup failure *P*(*fail*) is upper-bounded as

$$P(fail) \le \sum_{s=1}^{n} \left(\frac{e^{\frac{k}{2}+1}}{2^{\frac{k}{2}} s} \right)^{s} \left(\frac{sk}{m} \right)^{\frac{sk}{2}}. \qquad (3)$$

First, let us investigate how *P*(*fail*) varies with the lookup table size *m* and the number of hash functions *k*. The graph of *P*(*fail*) versus the ratio *m*/*n* for *n* = 4k elements is shown in Fig. 1. There is a separate curve for each value of *k*. Note that the failure probability decreases marginally with increasing *m*/*n*,

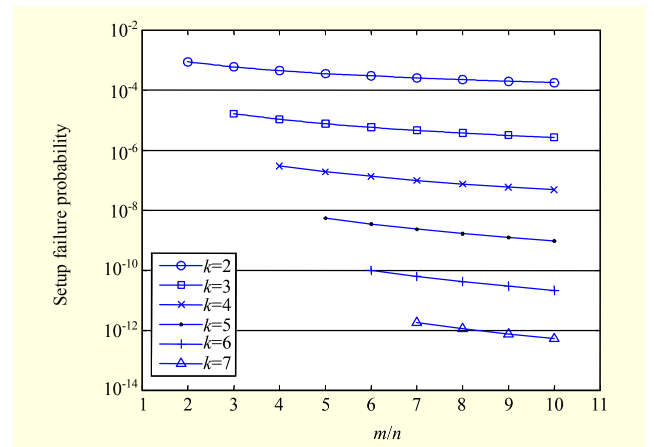

Fig. 1. Setup failure probability versus *m*/*n* (*n* = 4k).

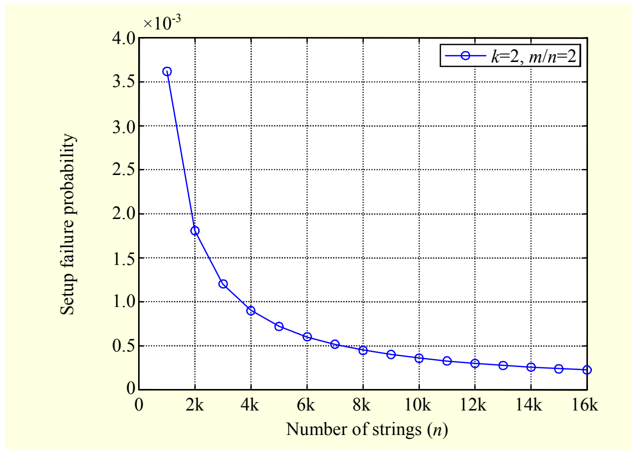

Fig. 2. Setup failure probability versus n.

but it decreases significantly with increasing k. However, a high value of k comes at the expense of large storage to maintain $m \geq kn$. Hence, a suitable value of k should be chosen to balance the system cost against the probability of setup failure. For the proposed architecture, k = 2 and m/n=2 are chosen since this provides a small failure probability yet yields a total storage requirement of a few bits per character for the lookup table. Moreover, from an implementation point of view, two hash functions map well to the dual ports of each embedded ram block of modern FPGA devices. Figure 2 is a graph of $P(fail)$ versus n with k = 2 and m/n = 2. Note that $P(fail)$ decreases dramatically as n increases.

2. Eliminating False Positives

In the Bloomier filter, a false positive can happen when some element x which was not among original elements used in filter setup produces some value for $\hbar_\tau(x)$. Hence, because $\tau(x)$ is not null as expected, this returns an incorrect $f(x)$ which lies at location $\tau(t)$ in the result table. Chazelle and others [3] addressed such false positives by using concatenation of a checksum to $\hbar_\tau(x)$ instead of $\hbar_\tau(x)$ in (1) during setup. After element x is looked up, a checksum is computed and compared with the checksum obtained from the concatenation yielded by (2) during lookup. The wider this checksum is, the smaller the probability of false positives (PFP) would be. Thus, Chazelle and others [3] effectively trade off storage space to reduce PFP.

Note that a non-zero PFP, no matter how small, is unacceptable for applications like intrusion detection because it would lead to false alerts. Therefore, reducing the probability of collisions does not guarantee the worst-case deterministic lookup rate desired by the line rate, and such network intrusion detection systems would be vulnerable to denial of service attacks.

A storage-efficient scheme proposed by Hasan and others [12] is used to eliminate false positives in the string matching architecture. The basic idea is to store all original strings in a table, and compare them with the search strings. In the Bloomier filter architecture, the result table can be used for this purpose. It means $f(x)$ which is extracted from location $\tau(t)$ in the result table is a possible match string. This table has as many locations as the lookup table (that is, $m \geq kn$), but only n locations are really needed to store the n elements. Hence the naive way would need k times more storage than what is needed to actually store all the elements. Instead, as described by Hasan and others [12], we can dissociate the size of the lookup and result tables. During setup, instead of encoding $\hbar_\tau(x)$, for each x, a pointer $p(x)$ which directly points into a result table having n locations is encoded. Thus, the lookup table encoding equation of (1) is modified as

$$V(x) = \Big(\bigoplus_{i=1, i\neq \hbar_\tau(x)}^{k} D[h_i(x)]\Big) \oplus p(x). \qquad (4)$$

On the other hand, during lookup, $p(x)$ is extracted from the lookup table using (2), and x is read out of the search string against the value of x. If they match, there is a correct result; otherwise, it is a false positive.

The lookup table, which stores pointers with maximum value n, requires $\log_2(n)$ bits ($q = \log_2(n)$). The naive approach, which encodes $\hbar_\tau(x)$ into the lookup table, requires only $\log_2(k)$ bits per element ($q = \log_2(k)$). In spite of this increase in the lookup table size along with a reduction in the size of the result table, this approach reduces the overall required storage compared to the naive approach.

3. Scalability

We will now discuss how the proposed architecture scales with an increase in the number of strings. The total storage space required by the underlying data structure of the proposed architecture consisting of two tables (lookup table and result table) is $kn\log_2(n)+nL$, where k is the number of hash functions, n is the number of strings, and L is the maximum string length (assuming all strings have the maximum length). The storage requirements for string set sizes in the range from 4k to 32k for several values of k and L are shown in Table 1.

While the storage space required by the result table (nL) scales linearly with an increase in the number of strings, the storage space required by the lookup table ($kn\log_2(n)$) scales faster than linear. However, the overall storage requirement of the proposed architecture scales approximately linearly with an increase in the number of strings because the result table takes a greater proportion of the overall required storage. Note that L is much larger than $k\log_2(n)$ in our target applications. This can

Table 1. Storage requirements of the proposed architecture.

Number of strings	Storage requirement (Mbits)							
	L=32				k=2			
	k=2	k=3	k=4	k=5	L=16	L=32	L=48	L=64
4k	1.09	1.14	1.19	1.23	0.59	1.09	1.59	2.09
8k	2.20	2.30	2.41	2.51	1.20	2.20	3.20	4.20
12k	3.31	3.48	3.64	3.80	1.81	3.31	4.81	6.31
16k	4.43	4.66	4.87	5.09	2.43	4.43	6.43	8.43
20k	5.55	5.84	6.12	6.40	3.05	5.55	8.05	10.55
24k	6.68	7.03	7.37	7.71	3.68	6.68	9.68	12.68
28k	7.80	8.21	8.62	9.02	4.30	7.80	11.30	14.80
32k	8.93	9.41	9.87	10.34	4.93	8.93	12.93	16.93

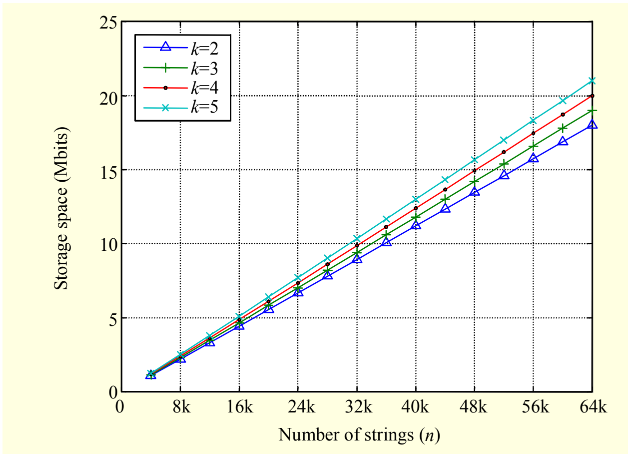

Fig. 3. Storage requirements vs. number of strings for L = 32 bytes.

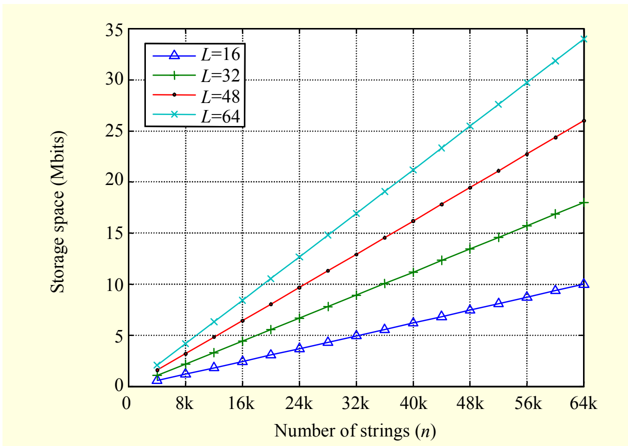

Fig. 4. Storage requirements vs. number of strings for k = 2.

be seen more clearly in Figs. 3 and 4 in which the storage requirements vs. the number of strings are shown. In Fig. 3, a string length equal to 32 bytes is chosen, and a separate graph

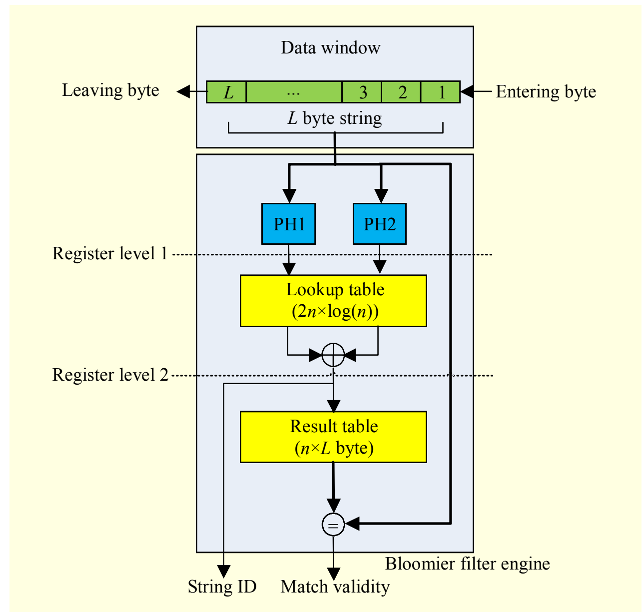

Fig. 5. Basic proposed architecture.

for values of k ranging from 2 to 5 is shown. In Fig. 4, the number of hash functions is chosen to be equal to 2, and a separate graph for values of L ranging from 16 bytes to 64 bytes is shown.

4. Architecture Overview

The basics of the proposed architecture are shown in Fig. 5. The Bloomier filter engine analyzes a window of L bytes from the data stream that arrives at the rate of one byte per clock cycle. As shown in this figure, the architecture data path involves three main operations, namely, hash computation and two memory accesses. First, an L byte data window is passed to two hashing blocks, PH1 and PH2. The hashing method

used in this architecture is Pearson hashing, which is described in the next section. Each PH produces $\log_2(2n)$ hash values used to address the lookup table. Two outputs of the lookup table are XORed to form the pointer showing a candidate match string. This pointer is used to look up the result table, and the resultant output is compared with the input data window to verify match validity.

To achieve a higher operating frequency, appropriate pipelining in the data path is utilized, which includes L stages in hashing blocks followed by two registered memory lookups. In this way, the system can verify the matching of the L byte string in a single clock cycle.

V. Implementation Details and Considerations

It is possible to employ some optimization techniques to reduce the amount of required resources and improve the design efficiency. The strings can be grouped according to their length such that strings of each group have a unique substring. Subsequently, we reduce the length of substrings, keeping only the bit-positions that are necessary to distinguish the strings. Finally, we generate the hash over these reduced substrings. This method requires fewer resources for hash computation. Also, to reduce the size of the result table, strings can be grouped such that strings of the same group have unique prefixes or suffixes in order to distinguish them using hashing. Generating groups of strings that have unique prefixes or suffixes increases the capacity to fit more strings in the memory. The effectiveness of these techniques, however, depends on the type of string and varies from case to case. To analyze the worst case performance of the proposed scheme, such techniques are not included in the presented design, and only the basic architecture is considered.

In the following subsection, the method to support the required hash functions in the hardware and perform the corresponding random lookups in the on-chip memory is described.

1. Hash Functions

The proposed architecture creates two hash functions based on Pearson hash [14]. This algorithm has been described by the following pseudocode to compute the hash of message C using the permutation table T and the auxiliary array h:

```
h[0] := 0
  for i in 1··· n loop
    index := h[i–1] xor C[i]
    h[i] := T[index]
  end loop
return h[n]
```

Permutation table T is a 256-byte lookup table which holds a permutation of the values 0 through 255. It is a CBC-MAC which uses an 8-bit random block cipher implemented via the permutation table. Since an 8-bit block cipher has negligible cryptographic security, the Pearson hash function is not cryptographically strong. However, it has the following advantages:

- Simplicity
- Fast execution with a few logic resources
- No simple class of inputs for which collisions are especially likely
- For a privileged set of inputs, the permutation table can be adjusted so that those inputs yield distinct hash values yielding a perfect hash function.

Given an input consisting of any number of bytes, it produces as output a single byte which is strongly dependent on every byte of the input [13]. Larger results can be made by running it several times with different initial hash values.

Since a unique one-to-one mapping cannot be produced, the Bloomier filter may fail. However, the failure probability is low due to the use of multiple hash functions. New hash functions with different initial values for h have to be used in case this happens. This ensures good hash selection. The entire process is completely automated and can be completed within only a few minutes [12].

Each PH shown in Fig. 5 is composed of two Pearson's hashing elements (PHEs), producing one byte each, that is, two bytes altogether. As previously mentioned, we need two hashes in our architecture, so a total of four PHEs are needed: PHE1 and PHE2 in PH1 and PHE3 and PHE4 in PH2. The PHE structure is shown in Fig. 6. Each PHE requires L 256×8-bit tables (T), L two-input 8-bit XOR gates, and an initial hash value (IHV). In FPGA devices with 4-input lookup tables (LUTs), each LUT can be configured as a 16×1-bit RAM. By implementing these tables with this distributed RAM method, we need L×256 LUTs for each PHE. Thus, four L×256 LUTs are needed for four PHEs. However, if we select the same IHV for PHE1 and PHE3, these elements can share L 256×16-bit tables (8 bits for the T table of PHE1 and the remaining 8 bits for the T table of PHE3) which could be realized with L×256 LUTs. This optimization could similarly be done for PHE2 and PHE4. Hence, instead of four L×256 LUTs, two L×256 LUTs are sufficient. By taking into account LUTs for implementing 8-bit XOR gates, that is, L×4 LUTs for each PHE making a total of 2×L×4 LUTs for all, we need L×1032 LUTs to realize the hashing section of our architecture. This equals L×1032 logic cells on Xilinx FPGAs.

To increase hashing speed, the PHE is pipelined in L stages as shown in Fig. 6. Each stage includes one XOR operation of an input byte with the previous T table output followed by a

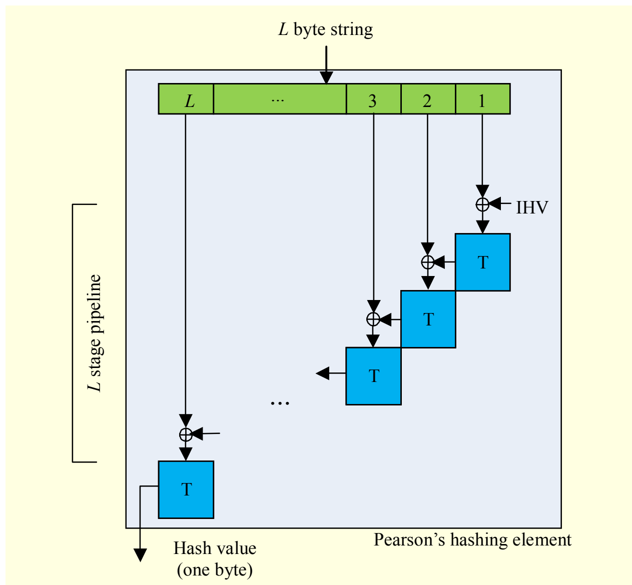

Fig. 6. Pearson's hashing element (PHE).

table lookup.

2. Realizing Memories

Each hash function corresponds to a random memory access in the lookup table. Therefore, in the proposed architecture the lookup table memory should be able to support 2 random accesses every clock cycle for 2 hash functions.

Memories with the density and lookup capacity required by the proposed architecture can be realized using embedded RAMs in modern FPGA devices. For instance, the Xilinx FPGA has a memory core called a block RAM which is physically a dual-port memory. Using this memory, two random memory locations can be read in a single clock cycle. Hence, two hash functions map well to the dual ports of each block RAM. As mentioned in section IV, the size of the lookup table is $kn\log_2(n)$ bits (as $k = 2$ in the proposed architecture). Hence, by using the FPGA's embedded memories in a dual-port configuration, the total memory required by the lookup table remains $2n\log_2(n)$.

The result table is also realized by using the FPGA's on-chip block RAMs. The total memory required by this table is nL, and on-chip block RAMs are used to implement this memory in a straightforward manner.

VI. Evaluation and Comparison with Previous Works

The proposed string matching scheme was evaluated using two main metrics: performance in terms of processing throughput and area cost in terms of required FPGA resources (all post place and route results). To compare the proposed scheme with previously reported research, a metric which takes into account both performance and area cost simultaneously namely *throughput*/(*logic cells*/*char*) was used.

Our implementation targets Xilinx Virtex-4 FPGA devices. The instance of the matching architecture was implemented on a Virtex4 FX 100-11 chip using Xilinx ISE 10.1i. It supports up to 16k single size signatures of 32 bytes. The architecture has 2 hash functions, and in order to perform 2 concurrent memory accesses, a dual-port memory with the required lookup table size is used as described in section V.2. The lookup table (2×16k×14 bits) and the result table (16k×256 bits) were both implemented using the FPGA's on-chip embedded SRAM blocks.

All hardware elements of the proposed scheme were implemented on the FPGA, while software functions such as the setup algorithms were executed on the host processor. A clock speed of 260 MHz was achieved on the FPGA. Therefore, by processing one byte per clock cycle, the proposed design could achieve more than 2 Gbps of throughput.

Table 2 shows the results of the proposed scheme versus the latest reported research on string matching designs. Note that methods like the ones proposed by Dharmapurikar and others [6] and Attig and others [15] need off-chip memories to verify their approximate matching results which are not shown in this table. More importantly, it is worth noting that most of the approaches listed in this table use the Snort rule set in their implementations. In these schemes, by preprocessing the signature strings and taking advantage of the optimization techniques like those mentioned in the beginning of section V, they improve the performance and efficiency of their implementations. In contrast, we did not make any assumptions about the string set; thus, the type of strings has no role in the reported results. Therefore, changing the string set does not affect the performance and efficiency of the proposed work, though it may affect the performance and efficiency of previously reported methods.

Moscola and others [19] proposed an architecture that can handle 3,000 characters at 384 MHz using 3.8k LUTs in the best case. The resource requirements of their design increase linearly as the number of characters increases, and about 31k patterns with an average length of 8 bytes or 250k characters can be accommodated using Xilinx Virtex 4 LX200. Virtex 4FX 100 has twice the logic resources compared with Virtex 4X 100. Hence, the design proposed by Moscola and others [19] requires about four times more logic resources to handle 512k characters compared to our proposed design, and theirs cannot be implemented on a single chip.

The comparison in Table 2 shows that the proposed design achieves better performance and higher efficiency, demonstrated

Table 2. Comparison of the results of the proposed scheme and other FPGA-based schemes.

Description	Input bits/cycle	Device	Throughput (Gbps)	Chars	Memory (kbit)	Logic cells	Throughput / (logic cells/char)
Proposed scheme	8	XC4VFX100	2.1	512k	4,544	35k	30.7
USC Bitsplit [7]	8	XC4VFX100	1.6	16,715	6,000	4,513	6
Bloom filters [15]	8	XCV2000E	0.6	420k	629	36,720	7
PHmem [9]	8	XC2V1000	2.1	20,911	288	6,832	6.5
HashMem +Reuse [10]	8	XC2VP7	2.7	18,636	558	2,759	18
RDL+ROM [16]	8	XC3S1000	1.9	20,800	162	~8,000	5
DCAM [17]	8	XC2V3000	2.6	18,036	0	17,538	2.7
Tree-based [18]	8	XC2VP100	1.9	19,584	0	6,340	5.9
TSM [19]	32	XC4VLX200	12.28	3,000	0	3.8k	9.6

by the throughput/(logic cells/char) metric value compared to the best reported schemes. The proposed design mainly makes use of embedded on-chip memory blocks, and the logic resources required for implementing this scheme are independent of the number of strings. The proposed design results in a much better metric value (more than 70% better) than the best result of previously reported schemes. Moreover, as the number of strings increases in applications such as virus scanning, the efficiency of the proposed architecture become even more obvious.

VII. Conclusions and Future Work

In this paper, a storage-efficient architecture was proposed to address the string matching problem. This scheme is based upon a recently proposed hashing scheme called the Bloomier filter, which is a memory efficient data structure. It leads to a simple yet powerful architecture which can handle several thousands of strings at high rates and is amenable to on-chip realization. Due to its dependence on only on-chip memory, it is more scalable in terms of speed and the size of the string set, when compared to other hardware approaches based on FPGAs.

The proposed scheme can be used as a basic building block in designing solutions for network intrusion detection, virus scanners with large databases, and generic content searches. As the speed of the data stream, which may be network, disk, or file traffic increases, this scheme will be invaluable for maintaining the required performance and efficiency.

The proposed design is based on the original architecture presented by Chazelle and others [3]. It has room for improvement by investigating the possibility of using the simple construction of a Bloomier filter, which Charles and Chellapilla [20] claim is linear in space and requires constant time to evaluate.

References

[1] Snort-The Open Source Network Intrusion Detection System, http://www.snort.org

[2] Open Source Clam Antivirus. http://www.clamav.net

[3] B. Chazelle et al., "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," *Proc. 15th Ann. ACM-SIAM Symp. Discrete Algorithms*, 2004, pp. 30-39.

[4] B. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, 1970, pp. 422-426.

[5] N. Tuck et al., "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," *IEEE INFOCOM*, vol. 4, 2004, pp. 2628-2639.

[6] S. Dharmapurikar et al., "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro*, vol. 24, no. 1, 2004, pp. 52-61.

[7] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Content Filtering," *Proc. Symp. Architecture for Networking and Communications Systems (ANCS)*, 2005, pp. 183-192.

[8] L. Tan and T. Sherwood, "Architectures for Bit-Split String Scanning in Intrusion Detection," *IEEE Micro*, vol. 26, no. 1, 2006, pp. 110-117.

[9] H. Jung, Z.K. Baker, and V.K. Prasanna, "Performance of FPGA Implementation of Bit-Split Architecture for Intrusion Detection Systems," *20th Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2006.

[10] I. Sourdis et al., "A Reconfigurable Perfect-Hashing Scheme for Packet Inspection," *Proc. 15th Int. Conf. Field Programmable Logic and Applications (FPL)*, 2005, pp. 644-647.

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

Programmable Logic and Applications (FPL), 2005, pp. 39-44.

[12] J. Hasan et al., “Chisel: A Storage-Efficient, Collision-Free Hash-Based Network Processing Architecture,” *Proc. 33rd Ann. Int. Symp. Computer Architecture (ISCA)*, IEEE CS Press, 2006, pp. 203-215.

[13] M. Ramakrishna and J. Zobel, “Performance in Practice of String Hashing Functions,” *Proc. 5th Int. Conf. Database Systems for Advanced Applications*, 1997, pp. 215-224.

[14] P.K. Pearson, “Fast Hashing of Variable-Length Text Strings,” *Communications of the ACM*, vol. 33, no. 6, 1990, pp. 677-680.

[15] M. Attig, S. Dharmapurikar, and J. Lockwood, “Implementation Results of Bloom Filters for String Matching,” *IEEE Symp. Field-Programmable Custom Computing Machines*, 2004, pp. 322-323.

[16] Y.H. Cho and W.H. Mangione-Smith, “Programmable Hardware for Deep Packet Filtering on a Large Signature Set,” *First Watson Conf. Interaction between Architecture, Circuits and Compilers*, 2004.

[17] I. Sourdis and D. Pnevmatikatos, “Predecoded CAMs for Efficient and High-Speed NIDS Pattern Matching,” *Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* IEEE CS Press, 2004, pp. 258-267.

[18] Z.K. Baker and V.K. Prasanna, “Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs,” *IEEE Trans. Dependable and Secure Computing*, vol. 3, no. 4, 2006, pp. 289-300.

[19] J. Moscola, J.W. Lockwood, and Y.H. Cho, “Reconfigurable Content-Based Router Using Hardware-Accelerated Language Parser,” *ACM Trans. Design Automation of Electronic Systems*, vol. 13, no. 2, article 28, 2008, pp. 28:1-28:25.

[20] D. Charles and K. Chellapilla, “Bloomier Filters: A Second Look,” *Lecture Notes in Computer Science*, *Proceedings 16th Annual European Symp. Algorithms*, vol. 5193*,* 2008, pp. 259-270.

Ali Peiravi received the BS degree in electrical engineering from the University of Pittsburgh in 1976, the MS degree in electrical engineering from the University of California at Berkeley in 1978, and the PhD degree in electrical engineering from the University of California at Irvine in 1984. He worked as a senior design engineer at Intersil from 1978 to 1980 and at American Microsystems Inc. from 1980 to 1981. He has been a full time faculty member at the Ferdowsi University of Mashhad, Mashhad, Iran, since 1984. His main areas of interest are system reliability, power systems, real time control systems, and design of VLSI. He is currently an associate professor of electrical engineering at the Ferdowsi University of Mashhad.

Mohammad Javad Rahimzadeh received the BS and MS degrees in electrical engineering from the Ferdowsi University of Mashhad, Mashhad, Iran, in 2004 and 2007, respectively. He was a senior member of the FPGA & ASIC Design Lab of Ferdowsi University from 2003 to 2006. He is a co-founder and the chief executive officer of Pouya Fanavar Imen R&D Co. His research interests include analysis, design, and high performance hardware realization of computing algorithms, especially for telecom and network security applications.