Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 9 (3): 488-496, 2009
ISSN 1812-5654

Design and Implementation of a Novel High Performance Content Processor for Storage Disks Using an Exact String Matching Architecture

A. Peiravi and M.J. Rahimzadeh
Department of Electrical Engineering, School of Engineering,
Ferdowsi University of Mashhad, Mashhad, Iran

Abstract: In this study, a high performance content processor for storage disks is proposed which could be easily installed in any host as an interface between the hard disk and the system bus. Moreover, a novel and powerful exact string matching architecture is presented using which we can search for several thousand strings at high rates. The average database size and associated software support systems are growing at rates that are greater than the increase in general purpose processor performance. Also, at the physical level there is a remarkable growth in disk storage performance. However, with the ever-increasing amounts of information available, the ability to accurately and quickly search and retrieve desired information has become a critical issue. We have implemented the proposed architecture on a Xilinx XC4VFX100 Field Programmable Gate Array (FPGA) and shown that the system can search for over sixteen-thousand 32 byte strings with a speed near the maximum stated in ATA-7 standard. Present design implementation has a much better performance as measured in Throughput/(LogicCells/Char) when compared with the best existing designs.

Key words: Custom computing, string matching, storage devices, programmable hardware, bloomier filter

INTRODUCTION

There is an ever increasing growing trend in the size of data bases and the associated systems. This is due to the increased public use of data storage especially with the rapid developments in the use of internet. On the other hand, there has been a great deal of progress in storage systems such as high capacity disk storage devices. Thus, there is a need for a more reliable search technique as a basis of many modern data processing applications. Another important application is data security which is becoming more and more important as internet-based transactions are on the rise. The threats posed by potential attackers who seek to discover important features of the data being transmitted should be considered in the design and implementation of data security applications. This also reinforces the need for fast and reliable search algorithms which do not slow down the data transfer process.

The software based techniques are not fast enough to meet these requirements. Thus, there is a serious need for the design and implementation of high performance hardware based content processors. A good place for such hardware devices is the data storage devices of the host processors where a large amount of data is constantly being transferred at a high speed. This could include the hard disk of a personal computer or the server of an electronic mail host.

In this study, the design and implementation of a high performance content processor for storage disks using a novel exact string matching architecture is presented. This content processor could act as an interface between the main CPU and the disk as a hard disk accelerator and present the capabilities of a content processor to search and find specific data in the text being transferred without using up the main processor's resources. It includes a powerful search core using a novel architecture which is scalable and can search in a large number of text strings at a high speed. The proposed system is implemented using an FPGA.

STRING MATCHING ALGORITHMS

There are many string matching algorithms already available. The problem which is addressed in string matching problems is defined as follows. Given a text, T, where T is an array T[1..n] of characters and a pattern P, where P is an array P[1..m] of characters, where we usually have $m<n$ and typically $m<<n$, we wish to know whether P appears as a substring of T and if so, where and may be,

Corresponding Author: Ali Peiravi, Department of Electrical Engineering, Faculty of Engineering,
Ferdowsi University of Mashhad, Mashhad 9177948974, Iran
Tel: 098-0511-881-5100/653 Fax: 8763302

how often. The applications of these algorithms are in text editors, web searching and genetics. The common algorithms for string matching are the naive algorithm or brute force method with a time complexity of O(n-m+1)×m) and the pre-computation method where we compute some extra information about P and/or T before starting the search to speed up the search for P in T as a result of which this can be faster than the naive algorithm. The history of this dates back to Karp and Rabin (1987), who presented what later became known as the Karp-Rabin algorithm where a hash function h is used for strings of length m. Other algorithms were presented later. For example, in the Boyer-Moore Algorithm both character-jump heuristic and looking glass heuristic are applied. Here the worst case run-time is $O(n\times m+|\sum|)$ and the runtime can be improved to O(m+n) by also using good-suffix heuristic. There are also many other algorithms such as the Morris-Pratt, Knuth-Morris-Pratt, Galil-Seiferas, Boyler-Moore, Turbo-Boyler-Moore, Tunnel-Boyler-Moore, Zhu-Takaoka, Berry-Ravidran, Smith, Raita, Horsepool, etc. to name a few. The details of these algorithms are presented in many studies which could be referred to by the interested reader.

IMPLEMENTATION APPROACHES

Data security could be implemented using either a software or a hardware approach. The software approach is limited to single user/client-side applications or very low bandwidth server applications because of the computational intensity of authentication and encryption algorithms. Software implementations only support modest throughput while better performance could be achieved with a reasonable cost with hardware. The next approach would be to accelerate software solutions with custom hardware which is common in x86- and Network Processor Unit (NPU)-based system designs. Several companies such as Cisco, NetScreen and PMC-Sierra produce ASIC security programmable co-processors. In many applications an in-line architecture is more appropriate where in-line processors integrate data security directly into the system data path. This approach is used in communications-centered applications such as VPN gateways and storage security appliances.

Many researchers have worked on FPGA-based string matching for data security due to their reconfigurability. ASIC and FPGA solutions provide much better performance when compared to software solutions. Many ASIC intrusion detection systems have been commercially developed. They usually store their rules in large memory blocks and examine incoming packets in integrated processing engines. ASIC programmable security co-processors are expensive, complicated and do not achieve impressive performance. The memory blocks that store the data security rules are re-loaded whenever an updated rule set is available. The most common technique for pattern matching in ASIC intrusion detection systems is the use of regular expressions. Sailesh *et al*. (2006) used regular expression-based pattern matching since it offers superior expressive power and flexibility. The approach presented by Sailesh Kumar *et al*. (2006) could reduce the number of distinct transitions between states. The architecture which was proposed performed deep packet inspection at multi-gigabit rates using multiple on-chip memories in such a way that each remains uniformly occupied and accessed over a short duration, thus effectively distributing the load and enabling high throughput providing a cost-effective packet content scanning at OC-192 rates with memory requirements consistent with current ASIC technology. The problem with their approach is that it is not trivial to update the rule set and the system should be able to support a variation of rules, with sometimes complex syntax and special features. However, FPGAs are better since they are reconfigurable and exploit parallelism. An FPGA-based system can be entirely changed by just keeping the interface constant. Pryor *et al*. (1993) were the first to propose a flexible, reprogrammable hardware solution to the acceleration of text-based keyword search problems. Their solution employed an attached processor called Splash 2, which exploits the speed and reconfigurability of field programmable gate array technology. Their algorithm, implemented on Splash-2 platform succeeded to perform a dictionary search that consisted of English alphabet characters without case sensitivity. They achieved a great performance and performed a low overhead and-reduction of the match indicators using hashing. The Splash 2 system was designed and built and it was comprised of an interface board to a Sun Sparc-2 host and up to 16 Splash boards, each of which contained 16 Xilinx 4010 FPGAs interconnected in a linear array and also through a 16-way full crossbar switch. Each Xilinx chip was coupled with a 4 Mbit static RAM through a dedicated interface. The text searching program implemented on a one-board Splash 2 system was capable of processing text at an estimated rate of 50 million characters per second.

Since 1993, many others have worked on implementing FPGA-based string match systems. Sidhu and Parasanna (2001) presented a method for finding matches to a given regular expression in given text using FPGAs. To match a regular expression of length n, a serial machine requires $O(n^2)$ memory and takes 0(1) time

per text character. Their approach required $O(n^2)$ space and processed a text character in 0(1) time (one clock cycle). The improvement that they had obtained was due to the Nondeterministic Finite Automaton (NFA) used to perform the matching. They implemented their algorithm for conventional FPGAs and the Self-configurable Gate Array (SRGA) and mapped the NFA logic onto the Virtex XCV100 FPGA and the SRGA.

Several techniques have been proposed for string matching especially in the context of network intrusion detection in recent years. 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. Hence, scalability with string set size is the main concern with purely FPGA-based approaches.

From the scalability viewpoint, memory-based techniques are attractive since memory chips are cheap. Unfortunately, memory access speed becomes a bottleneck in using memory-based techniques. An optimized hardware-based Aho-Corasick algorithm was proposed by Tuck *et al*. (2004) that drastically reduces the amount of memory required and improves its performance on hardware implementations. 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 would suffer from power consumption issues.

A Bloom filter based algorithm proposed by Dharmapurikar *et al*. (2004) is made 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. Dharmapurikar *et al*. (2004) argued that since the strings of interest are rarely found in the packets, the quick check in 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 (2005) modifies the classic Aho-Corasick algorithm to allow it to consider multiple characters. This modification allows parallelizing the Aho-Corasick algorithm and using multiple instances of it to achieve a 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 perform the necessary memory accesses which would slow down the string matching process. Tan and Sherwood (2006) proposed a technique for reducing the out-degree of string-matching state machines. The technique presented by Tan and Sherwood (2006) is allowed state machines to be represented using significantly less state memory that would be required in a naive implementation. Through the use of bit-splitting, a single state machine is split into multiple machines that handle some fraction of the input bits. The earlier works did not provide many of the requirements for a realistic implementation. The basic architectural design proposed by Tan and Sherwood (2006) was adapted to an FPGA implementation by Jung *et al*. (2006), who showed that by considering FPGA implementation details such as restriction of a block RAM size, pin count and routing delay, this architecture does not use on-chip memory efficiently and wastes a high percentage of on-chip block RAMs. Sourdis *et al*. (2005) and Papadopoulos and Pnevmatikatos (2005) tried to combine hashing and use of memory in order to present a cost-effective exact string matching solution. However, they had to use additional indirection memories to reduce the sparseness of the memory and compact storing of the search strings because of their hashing schemes. Moreover, the implementation of their hashing modules uses FPGA lookup tables and the place & route of such a design takes a couple of hours.

BLOOMIER FILTER

The key to the proposed string matching architecture in this study is that it is based on the Bloomier filter introduced by Chazelle *et al*. (2004), which is a generalization of Bloom filter. While a Bloom filter is designed to represent a set, a Bloomier filter is designed to represent a function on a set. 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 set, S, the filter always returns f(x) if $x \epsilon S$. 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. We call this data structure the lookup table. Also, the k hash values of an element x, $\{h_1,..., h_k\}$ where $0 \le h_i \le m$, are collectively referred to as its neighborhood, denoted by N(x).

The lookup table is constructed such that for every x, a location $\tau(x)$ among N(x) can be found in such a way that there is a one-to-one mapping between all x and $\tau(x)$. Because $\tau(x)$ is unique for each x, collision-free lookups are guaranteed. Later on in the study we describe how to tune the design parameters such that the probability of finding such a mapping is arbitrarily close to 1.

The idea is to construct the lookup table so that a lookup for x returns $\tau(x)$. Then we can store f(x) in another data structure called the result table, at address $\tau(x)$. Thereby, we can guarantee deterministic, collision-free lookups of arbitrary functions. Since the Result Table will be addressed by $\tau(x)$, it must have as many locations as the lookup table.

During lookup table setup, once we find $\tau(x)$ for a certain x, we store V(x) in the location $\tau(x)$ which is computed as follows:

$$V(x) = \left(\bigoplus_{\substack{i=1 \\ i \neq h_\tau(x)}}^{k} D[h_i(x)]\right) \oplus h_\tau(x) \quad (1)$$

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

$$h_\tau(x) = \bigoplus_{i=1}^{k} D[h_i(x)] \quad (2)$$

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

DESCRIPTION OF THE PROPOSED ARCHITECTURE

Here, we describe the proposed string matching architecture, which is based upon the Bloomier filter. First we consider the issue of tuning the filter design parameters. Next we describe how our architecture addresses false positives. Then we investigate the scalability of our architecture as number of strings increases. At the end of this section, we present an overview of our architecture.

TUNING THE DESIGN PARAMETERS

Chazelle *et al.* (2004) showed that for a Bloomier filter with k hash functions, n elements and a lookup table size $m \geq kn$, the probability of setup failure p(fail) is upper bounded as follows:

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

First, we investigate how P(fail) varies with the lookup table size m and the number of hash functions k. In Fig. 1 we show a graph of P(fail) versus the ratio m/n for n = 4K elements. There is a separate curve for each value of k. We note that the failure probability decreases marginally with increasing m/n, but decreases significantly with increasing k. However, a high value of k comes at the expense of large storage to maintain $m \geq kn$. Hence, we should choose a suitable value of k to balance system cost against probability of setup failure. Next, a graph of P(fail) versus n while fixing k = 4 and m/n = 4 is shown in Fig. 2. We note that P(fail) decreases dramatically as n increases. For values of n typical in

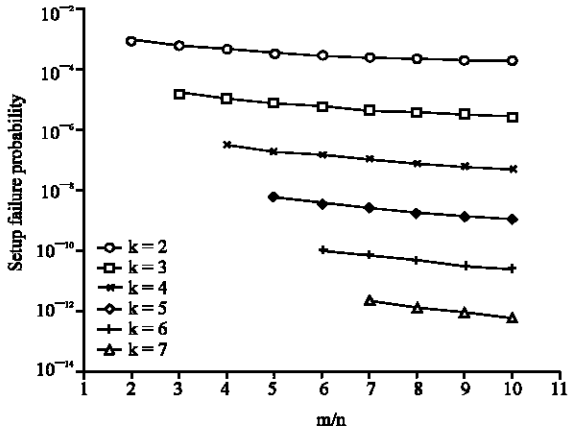

Fig. 1: Setup failure probability versus m/n (n = 4K)

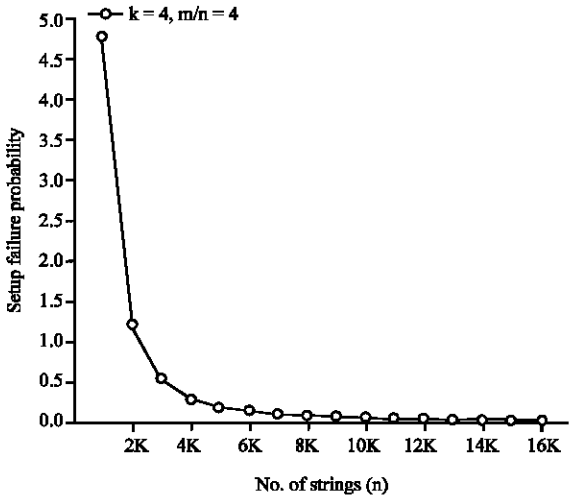

Fig. 2: Setup failure probability versus n

applications like intrusion detection (a few thousand), P(fail) is about 1 in 3 millions or smaller (i.e., 1 setup out of 3 millions setups may fail to converge).

For present architecture, we choose k = 4 and m/n = 4 because this provides a very small failure probability, yet yields a total storage requirement of a few bits per character for lookup table.

Eliminating false positives: In 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 $h_\tau(x)$. Hence, because $\tau(x)$ is not null as expected, this will return an incorrect f(x) which lies at location $\tau(t)$ in the result table. Chazelle *et al.* (2004) address such false positives by using concatenation of a checksum to $h_\tau(x)$ instead of $h_\tau(x)$ in Eq. 1 during setup. After looking up element x, checksum is computed and compared with the checksum obtained from the concatenation yielded by Eq. 2 during lookup. The wider this checksum is, the smaller the Probability of False Positives (PFP) would be. Thus, Chazelle *et al.* (2004) effectively trade off storage space to reduce PFP.

Note that a non-zero PFP is unacceptable for applications like intrusion detection, no matter how small it may be, 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 which has been proposed by Hasan *et al.* (2006) is used to eliminate false positives in our string matching architecture. The basic idea is to store all original strings in a table and compare them with the search strings. In 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 (i.e., $m \geq kn$), 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 *et al.* (2006), we can dissociate the size of the lookup and result tables. During setup, instead of encoding $h_\tau(x)$, 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 (Eq. 1) is modified as follows:

$$V(x) = \left(\bigoplus_{\substack{i=1 \\ i \neq h_\tau(x)}}^{k} D[h_i(x)]\right) \oplus p(x) \quad (4)$$

On the other hand, during lookup, p(x) is extracted from the lookup table using Eq. 2 and x is read out of location p(x) in the result table. Then we compare the search string against the value of x. If the two match, then there is a correct result. Otherwise, it is a false positive.

The lookup table which stores pointers with maximum value n, will require $\log_2(n)$ bits ($q = \log_2(n)$) while in the naive approach which encodes $h_\tau(x)$ into the lookup table, it will require only $\log_2(k)$ bits per element ($q = \log_2(k)$). This approach reduces the overall required storage compared to the naive one despite this increase in the Lookup Table size along with a reduction in the size of the result table.

Scalability: Now it will be shown how our architecture scales with increasing number of strings. The total storage space required by underlying data structure of the proposed architecture consists 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). In Table 1 this storage requirement is shown for string set sizes in the range of 4K to 32K for several values of k and L.

While the storage space required by the result table (nL) scales linearly with increasing number of strings, the storage space required by the lookup table ($kn \log_2(n)$) scales faster than linear. However, as the lookup table takes more part in the overall required storage of our architecture (L is much larger than $k \log_2(n)$ in our target applications), the overall storage requirement of the

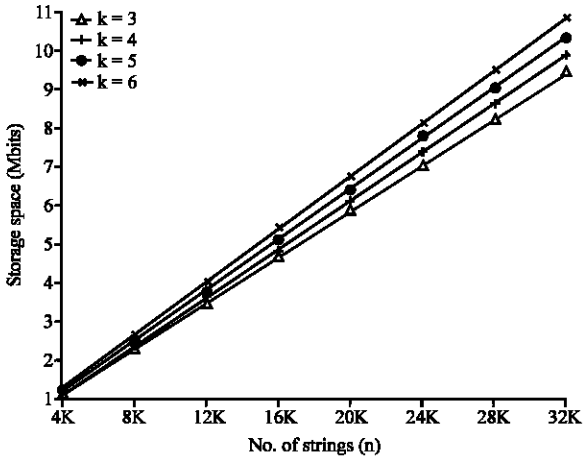

Fig. 3: Storage requirements vs. number of strings for L = 32 bytes

Table 1: Storage requirements of our proposed architecture

	Storage requirement (Mbits)							
	L = 32				k = 4			
No. of strings	k = 3	k = 4	k = 5	k = 6	L = 16	L = 32	L = 48	L = 64
4K	1.14	1.19	1.23	1.28	0.69	1.19	1.69	2.19
8K	2.30	2.41	2.51	2.61	1.41	2.41	3.41	4.41
12K	3.48	3.64	3.80	3.96	2.14	3.64	5.14	6.64
16K	4.66	4.87	5.09	5.31	2.87	4.87	6.87	8.87
20K	5.84	6.12	6.40	6.68	3.62	6.12	8.62	11.12
24K	7.03	7.37	7.71	8.05	4.37	7.37	10.37	13.37
28K	8.21	8.62	9.02	9.43	5.12	8.62	12.12	15.62
32K	9.41	9.87	10.34	10.81	5.87	9.87	13.87	17.87

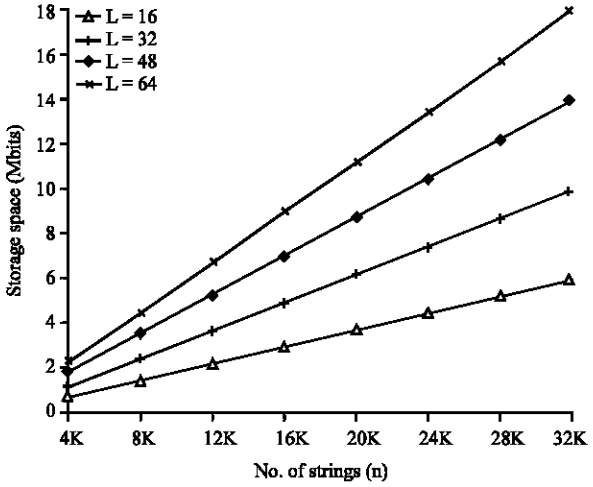

Fig. 4 Storage requirements vs. number of strings for k = 4

proposed architecture scales approximately linearly with increasing number of strings. In Fig. 3 we chose string length equal to 32 bytes and graphed a separate diagram for values of k ranging from 3 to 6 while in Fig. 4, we chose number of hash functions equal to 4 and graphed a separate diagram for values of L ranging from 16 to 64 bytes.

Architecture overview: The basic architecture proposed is shown in Fig. 5. The Bloomier filter engine analyzes a window of L bytes from data stream that arrives at the rate of one byte per clock cycle.

As shown in Fig. 5, the architecture data path involves three main operations consisting of hash computations and two memory accesses. Appropriate pipelining is utilized in the data path to achieve a higher operating frequency. In this way, the system can verify the matching of the L bytes string in a single clock cycle.

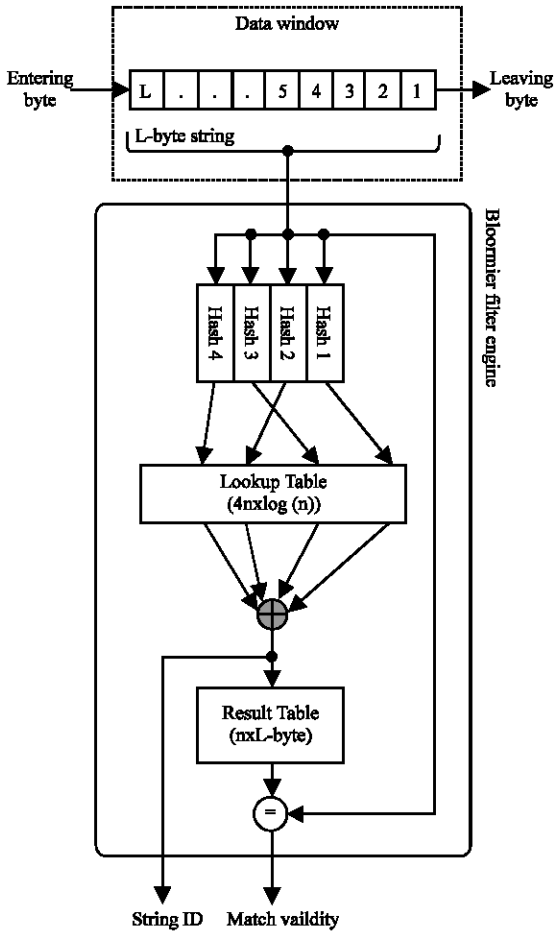

Fig. 5: Basic proposed architecture implementation details and considerations

IMPLEMENTATION DETAILS AND CONSIDERATIONS

It is possible to employ some optimization techniques to reduce the amount of required resources and improve the design efficiency. We can group the strings 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 computing. Also, to reduce the size of the result table, we can group strings such that strings of the same group have unique prefixes or have unique suffixes in order to distinguish them using hashing. Generating groups of strings that have unique prefixes or suffixes increases the chance to fit more strings in memory. The effectiveness of these techniques, however, depends on string types and varies from case to case. In order to analyze the worst case performance of the proposed scheme, we do not include such techniques in our design and consider the basic architecture.

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

Hash functions: The proposed scheme uses hash functions which are from a class of universal hash functions described by Ramakrishna *et al.* (1994). Hardware implementation of these functions is relatively inexpensive.

In our string matching architecture k hash functions are needed. These hash functions are calculated as described below.

For any string consisting of b bits, represented as, $<x_1, x_2, x_3,...x_b>$, i^{th} hash function over this string is calculated as:

$$h_i(x) = d_{i1}.x_1 \oplus d_{i2}.x_2 \oplus d_{i3}.x_3 \oplus ...d_{ib}.x_b \quad (5)$$

where d_{ij} is a predetermined random number in the range [1 . . . m].

It can be observed that the hash functions are calculated cumulatively. Hence, the results calculated over the first j bits can be used for calculating the hash function over the first j+1 bits. To support strings with different lengths, this property of the hash functions results in regular and less resource consuming hash functions.

Realizing memories: Each hash function corresponds to a random memory access in the lookup table. Thus, for 4 hash functions, the lookup table memory should be able to support 4 random accesses every clock cycle in our proposed architecture.

Memories with such density and lookup capacity required by our proposed architecture could be realized using embedded RAMs in modern FPGA devices. For instance, Xilinx FPGAs have a memory core called Block RAM which is physically a dual-port memory. Hence, using this memory, two random memory locations can be read in a single clock cycle. However, due to the block RAM's fast access performance, it is possible to create quad-port memories by time-division multiplexing the signals in and out of memory at the cost of some additional logic and access latency for each port. This involves a trade-off of data access time (halved) versus functionality (doubled). To improve the design performance, we utilized multiple clock domains to overcome memories latency.

The size of lookup table is kn $\log_2$ (n) bits (since k = 4 in our architecture). Hence, by using the FPGA's embedded memories in quad-port configuration, the total memory required by the lookup table remains 4n $\log_2$ (n).

The result table is realized by making use of FPGA's on-chip Block RAMs too. The total memory required by this table is nL and on-chip Block RAMs are used to implement this memory in a straightforward manner.

EVALUATION METRICS

The proposed string matching scheme is 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). In order to compare our proposed scheme with previously reported research, we use a metric which takes into account both performance and area cost simultaneously: Throughput/(LogicCells/Char).

The proposed architecture was implemented on Xilinx Virtex4 FPGA family and a Virtex4 FX 100 using Xilinx ISE 8.1i. was used which supports up to 16K single size signatures of 32 bytes. The architecture has 4 hash functions and a quad-port memory with the required lookup table size is used to perform 4 concurrent memory accesses. The lookup table (4×16k×14 bits) and result table (16k×256 bits) were both implemented using the FPGA's on-chip embedded SRAM blocks.

All hardware elements of our scheme are implemented on the FPGA, while software functions such as the setup algorithms are executed on the host processor. A clock speed of 150 MHZ was achieved on the FPGA. Therefore by processing one byte per clock cycle, present design can achieve 1.2 Gbps of throughput.

PRESENTING RESULTS AND COMPARISONS

Table 2 shows the results of our scheme against latest reported research on string matching designs. Note that methods like the ones proposed by Dharmapurikar *et al.* (2004) and Attig *et al.* (2004) need off-chip memories to verify their approximate matching

Table 2: Results of our scheme and other FPGA-based string matching approaches

Approach	References	Input bits/ cycle	Device	Through put (Gbps)	Chars	Memory (kbit)	Logic cells	Logic cells /char	Throughput/ (logic cells/ char
Present Approach		8	XC4VFX100	1.2	512K	4,992	27,925	0.04	22.6
USC Bitsplit[6]	Jung *et al.* (2006)	8	XC4VFX100	1.6	16,715	6,000	4,513	0.27	6
Bloom filters[13]	Attig *et al.* (2004)	8	XCV2000E	0.6	420K	629	36,720	0.09	7
Phmem[7]	Sourdis *et al.* (2005)	8	XC2V1000	2.1	20,911	288	6,832	0.32	6.5
HashMem+ Reuse[8]	Papadopoulos and Pnevmatikatos (2005)	8	XC2VP7	2.7	18,636	558	2,759	0.15	18
RDL+ROM[15]	Cho *et al.* (2004)	8	XC3S1000	1.9	20,800	162	~8,000	~0.38	5
DCAM[16]	Sourdis, and Pnevmatikatos (2004)	8	XC2V3000	2.6	18,036	0	17,538	0.97	2.7
Tree-based[17]	Baker and Prasanna (2006)	8	XC2VP100	1.9	19,584	0	6,340	0.32	5.9

results which are not shown in this table. More importantly, we should note that most of the approaches listed in this table use Snort rule set in their implementations. In these schemes, by preprocessing the signature strings and taking advantage of the optimization techniques 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 our reported results. Therefore changing the string set does not affect the performance and efficiency of our work while it may affect the performance and efficiency of previously reported methods.

However, the comparison shown in Table 2 indicates that our design has a better performance and higher efficiency seen by the Throughput/(LogicCells/Char) metric value compared to the best reported schemes. Since our design mainly makes use of embedded on-chip memory blocks and the logic resources required for implementing our scheme are independent of the number of strings, our design has a better metric value (more than 25% better) compared to the best result of previously reported schemes.

CONCLUSIONS

In this study, a storage-efficient architecture for the String Matching problem was presented. This scheme is based on a recently proposed hashing scheme called 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 FPGA.

Present scheme can be used as a basic building block to architect solutions for network intrusion detection as well as for generic content searches. As network speed increases, our scheme will be invaluable for maintaining the required performance and efficiency.

REFERENCES

Attig, M., S. Dharmapurikar and J. Lockwood, 2004. Implementation results of bloom filters for string matching. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, IEEE Computer Society, PP: 322-323.

Baker, Z.K. and V.K. Prasanna, 2006. Automatic synthesis of efficient intrusion detection systems on FPGAs. IEEE Trans. Dependable Secure Comput., 3: 289-300.

Chazelle, B., J. Kilian, R. Rubinfeld and A. Tal, 2004. The bloomier filter: An efficient data structure for static support lookup tables. Proceeding of the 15th Ann. ACM-SIAM Symposium on Discrete Algorithms, 2004, pp: 1-10.

Cho, Y.H. and W.H. Mangione-Smith, 2004. Programmable hardware for deep packet filtering on a large signature set. 1st Watson Conference on Interaction between Architecture, Circuits and Compilers, 2004. http://cares.icsl.ucla.edu/cares/content/papers/wassa04.pdf.

Dharmapurikar, S., P. Krishnamurthy, T. Sproull and J.W. Lockwood, 2004. Deep packet inspection using parallel bloom filters. IEEE Microbiol., 24: 52-61.

Dharmapurikar, S. and J. Lockwood, 2005. Fast and scalable pattern matching for content filtering. Proceeding of the Symposium on Architecture for Networking and Communications Systems, (ANCS'2005), ACM. New York, pp: 183-192.

Hasan, J., S. Cadambi, V. Jakkula and S. Chakradhar, 2006. Chisel: A storage-efficient, collision-free hash-based network processing architecture. ACM. Sigarch Comp. Architecture News, 34: 203-215.

Jung, H., Z.K. Baker and V.K. Prasanna, 2006. Performance of FPGA implementation of bit-split architecture for intrusion detection systems. Proceeding of the 20th IEEE Parallel and Distributed Processing Symposium, 2006, IPDPS (RAW), pp: 177-177.

Karp, R.M. and M.O. Rabin, 1987. Efficient randomized pattern-matching algorithms. IBM. J. Res. Dev., 31: 249-260.

Papadopoulos, G. and D. Pnevmatikatos, 2005. Hashing + memory = low cost, exact pattern matching. Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL), August 2005, pp: 39-44.

Pryor, D.V., M.R. Thistle, N. Shirazi, 1993. Text searching on splash. Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, April 1993, California, pp: 172-177.

Ramakrishna, M., E. Fu and E. Bahcekapili, 1994. A performance study of hashing functions for hardware applications. Proceedings of the 6th International Conference on Computing and Information, ICCI'94, May 26-28, Peterborough, Ont., pp: 1621-1636.

Sailesh, K., D. Sarang, Y. Fang, C. Patrick and J. Turner, 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer, SIGCOMM'06, September 11-15, Pisa, Italy, pp: 339-350.

Sidhu, R. and V.K. Prasanna, 2001. Fast regular expression matching using FPGAs. Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2001, Rohnert Park, CA, pp: 227-238.

Sourdis, I. and D. Pnevmatikatos, 2004. Predecoded CAMs for efficient and high-speed nids pattern matching. Proceeding 12th Ann. IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM'04), IEEE CS Press, pp: 258-267.

Sourdis, I., D. Pnevmatikatos, S. Wong and S. Vassiliadis, 2005. A reconfigurable perfect-hashing scheme for packet Inspection. Proceeding of the 15th International Conference on Field Programmable Logic and Applications (FPL), 2005.

Tan, L. and T. Sherwood, 2006. Architectures for bit-split string scanning in intrusion detection. IEEE Microbiol., 26: 110-117.

Tuck, N., T. Sherwood, B. Calder and G. Varghese, 2004. deterministic memory-efficient string matching algorithms for intrusion detection, In IEEE Infocom. http://www.cs.usd.edu/users/calder/papers/INFOCOM-04-IDS.pdf.