Bloom Filter Accelerator for String Matching

M. Nourani and P. Katta
Center for Integrated Circuits & Systems
The University of Texas at Dallas, Richardson, Texas 75080
{nourani,ppk031000}@utdallas.edu

***Abstract*— In this paper we present a hardware architecture for string matching. Our solution based on using a Bloom filter based pre-processor and a parallelized hashing engine is capable of handling wire line speeds with zero false-positive probability. String matching modules are extensively used in the network security domain especially in network intrusion detection systems where they are required to operate at wire line speeds. Our analysis shows that our system is capable of matching** 16000 **strings and achieves throughput in excess of** $100Gbps$ **(i.e. capable of handling** 10 $OC-192$ **links comfortably).**

***Index Terms*— Bloom filter accelerator, Computer network security, String matching, Pattern matching, Security monitoring**

I. Introduction

Network Intrusion Detection Systems (IDS) identify intrusive or malicious behavior by monitoring network activity. By performing deep packet inspection on packet payloads in addition to packet headers, IDS systems are able to limit spread of worms by scanning for malicious payloads. Popular network IDS systems like Snort [1] identify suspicious patterns that may indicate an attempt to attack, break into or compromise a system. The most common approaches used in IDS are statistical anomaly detection and string-match based detection. To be effective, these systems need to scan every byte of the packet payload and take appropriate action based on string-match detection. Both of these systems use string matching in one way or another. It has been shown [2] that string matching is the most expensive part of execution in Snort consuming 31% of its processing time and thus the most critical component to improve snort's (and equivalently any IDS system's) performance. With the average network speeds doubling every year and the processor speeds doubling only every 18 months [3], it is becoming increasingly difficult for software based IDS systems to keep up with the increasing network speeds, necessitating new special purpose hardware solutions for string matching.

We describe a new architecture for a string matching engine suitable for use in high performance IDS applications. Our approach attempts to utilize a Bloom filter based accelerator to improve on the processing speeds of current string matching engines. A Bloom filter based accelerator engine is used to reduce the number of strings that need to be inspected by the core string-match unit boosting the throughput of the overall system. We expect that the technique will be suitable for use in high speed networks where string matching needs to be performed at multi gigabit speeds in order to prevent becoming a bottleneck. Additionally, most network processors like Intel IXP1200 [4] and IBM PowerNP [5] now contain a dedicated hashing unit. Using a string matching engine based on hashing allows for a tighter integration with network processors and reuse of dedicated hash generation circuitry.

A. String Matching Problem Definition

Given a set of n strings $S = \{s_1, s_2, s_3, \ldots, s_n\}$ and a stream of packets P, we would like to find out all occurrences of S in P. The data P is streaming, which means we can look at the data only once. Practically, n is a large number (e.g. thousands). Also, as the location of the strings S in the packet stream P is unknown, a string matching system should be able to detect the strings at any location in the data stream.

B. Related Work

String matching problems have been extensively studied. New algorithms like the Aho-Corasick-Boyer-Moore (ACBM) [6] and setwise Boyer-Moore-Horspool [2] algorithm have a better average case performance than classical algorithms like Knuth-Morris-Pratt (KMP) [7], Boyer-Moore [8], Aho-Corasick [9] and Commentz-Walter [10] but the performance is still not good enough for multi-gigabit network speeds at reasonable cost.

Several hardware architectures have also been suggested for string matching. Techniques proposed in [11], [12], [13] use DFA or NFA in FPGA for detecting strings. FPGA allows for addition of strings through reconfiguration. These techniques can perform string matching at very high speeds but require reconfiguration which is expensive for addition and deletion of strings. Also these architectures cannot be migrated to the ASIC platform.

The simplest approach to string matching involves using a ternary content addressable memory (TCAM) [14] to store all the strings. Searches can be made in a single clock cycle and as all the strings involved are distinct, such a TCAM-based approach would require minimal decoding and would be blazingly fast. However, the cost of such a system makes this infeasible for sizes larger than a few strings.

Another interesting approach for string-matching is based on Bloom filters [15] . Bloom filters initially conceived by Burton H Bloom in 1970 are space efficient probabilistic data structures that support membership queries on a large set of elements. While false positives (i.e. identifying harmless strings as suspicious) are possible, false negatives (i.e. overlooking a pre-defined suspicious string) are not. Elements can be added to a set easily but removing elements is somewhat tricky (though this can still be handled using counting Bloom filters [18]). Bloom filters have received a significant amount of interest in literature and several variants have been suggested. Spectral Bloom filters [16] are used to estimate frequencies, Attenuated Bloom filters [17] are used to store neighbor information. Bloomier filters [19] are extended Bloom filters which allow execution of any arbitrary function on the set unlike Bloom filters which limit to membership queries. When applied to string matching, a Bloom filter based approach [20]

1-4244-1251-X/07/$25.00 2007 IEEE.

uses a Bloom filter with a very low false positive rate to filter out most strings.

C. Main Contribution

When a packet is sent over the Internet, it typically traverses a number of different networks before reaching its destination. There is a high probability of packet fragmentation in transit. Also when a number of packets are sent, these packets can arrive out-of-order at the destination. Before checking the data for malicious strings, the packet stream has to be reassembled and separated based on flows. Architectures for packet stream reassembly [21] [22] and for flow based stream separation already exist. These systems are capable of presenting a reassembled stream of data for further processing. We consider that such a data stream is available to our system and deal with the problem of efficiently finding malicious strings in these streams.

Our approach uses a combination of Bloom filters and parallel hashing. The data stream is initially passed through a Bloom filter which acts as an accelerator. These selected strings are then fed to a string dispatcher which dispatches the strings to the PH (parallel hashing) engine [23]. The PH engine performs a hash comparison and in case of a hash hit compares the input string with the actual string to eliminate any false-positives.

The PH engine uses distributed comparison and lookup logic. By distributing comparison and lookup logic, the PH engine processes significantly higher throughput than other hashing based architectures while at the same time handling significantly biased packet payloads without a significant drop in performance. We show that even in the worst case our system is no worse than existing hash based systems while achieving an average throughput significantly higher than such systems.

The rest of the paper is organized as follows. Section II gives an overview of the system architecture. Section III provides an overview of the Bloom filter accelerator. Section IV provides an analysis of the throughput of the system and Section V reports the experimental results.

II. Parallel Hashing Engine

Using Bloom filters for pre-processing is generic enough to allow incorporation into any traditional string matching engines. To show this, we integrate the accelerator with our Parallel Hashing engine (PH). In this section we provide a brief overview of the PH engine. For more details please refer to [23].

The PH engine is a string-match engine based on parallel hashing that processes multiple bytes of data per clock cycle. In order to guarantee zero false-positivity the hash comparison stage is followed by a string comparison phase. The engine is capable of tolerating hash collisions and can be configured in different ways depending on the application. Also as malicious data can start at any offset in the data stream, the data window (i.e the data that is currently being checked by the string match engine) the window can only be extended by one byte every step as shown in Figure 1.

The PH engine uses n_h bit wide hash generators to identify if any of the strings in its string table are present in the input

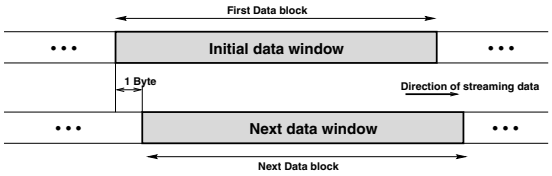

Figure 1. Data windows for PH engine

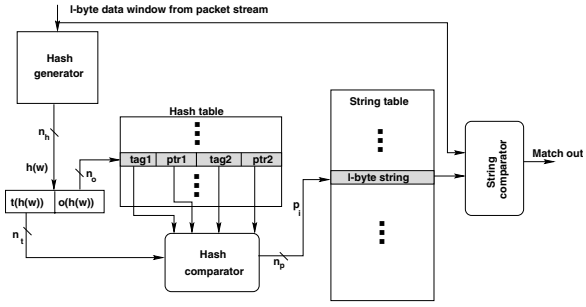

Figure 2. Data flow through the engine

data stream. Each of the strings in the string table are of a fixed l byte long. We hash the l byte strings to a n_h bit value.

In the simplest case as the hash function is n_h bits wide, the size of the required hash table would be $(2^{n_h} \times n_p)$ bits as n_h bits are used to index into the hash table and each location contains a n_p bits pointer that points to the corresponding string in the string table.

Instead we divide the hash table into a tag field n_t bits wide and an offset field of n_o bits, we use the n_o bits of the offset to index into the hash table and store two sets of tag, pointer in each location (i.e. we have two hash entries per location). As the tag and offset fields are derived from the hash field, $n_h = n_t + n_o$ and the size of the hash table is now reduced to $(2^{n_o} \times 2(n_p + n_t))$ bits. This causes 2^{n_t} strings to hash to the same location in the hash table. Reducing n_t reduces the number of strings that may be hashed to the same location and hence reduces the probability of a hash collision. (i.e increase the loadability of strings into the system) at the cost of more memory consumption.

When loading a string s_i into the system, we first compute the hash for the string $h(s_i)$ of size n_h bits. This is then split into the tag $t(h)$ of size n_t and the offset $o(h)$ of size n_o bits, the string s_i can be loaded into the engine only if there is at least one free entry in the hash table at location $o(h)$ and no entry at this location has the same tag value. If loadable, the string is loaded into the string table at location p_i and the entry $(t(h), p_i)$ is loaded into the hash table at location $o(h)$.

In the querying phase, a hash $h(w)$ is computed on the current l byte window of data, the computed hash is split into the tag $t(w)$ and the offset $o(w)$. The two tags at $o(w)$ are extracted and compared to $t(w)$. If a match is found, the data in the window has to be compared with the actual string extracted from the string table using p_i to rule out the possibility of a false positive. This ensures that the system never reports any false-positives and forms the basis for our zero false-positive claim. The data flow through the system is shown in Figure 2.

Instead of having one hash table as in the basic string

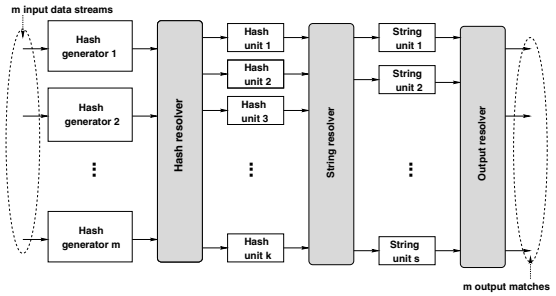

Figure 3. Parallel hashing string match engine

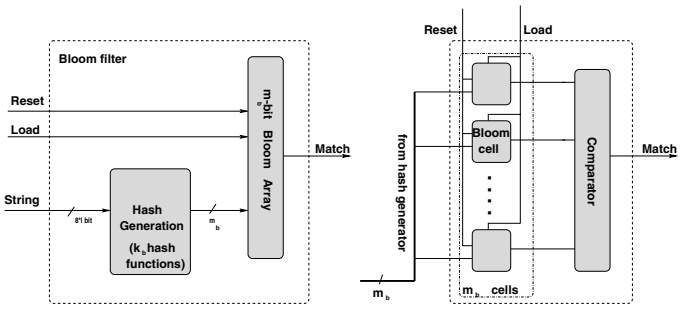

Figure 4. (a) The Bloom filter (b) Memory and comparison Logic

match engine, we split the hash table into k equal sized blocks (called the hash blocks). Each of the k hash units shown in Figure 3 contains one hash block along with a dedicated comparator. The same idea is extended to string comparison. The string table is broken into s equal sized blocks, each string unit contains one string block with a dedicated comparator. Multiple hashes can be compared in a single cycle if the hashes index into hash blocks in different hash units. However, more than one hash generators may index into the same hash block, in order to resolve this, we have to arbitrate access to the hash units. When more than one hash generators contend for the same hash unit, only one of the contending generators is allowed to access the hash unit, the remaining generators contend for the unit in the next cycle. The same idea is extended to the string units. Access to the string units from the hash units is arbitrated by the string resolver.

In essence, we now have the $(m,\ k,\ s)$ architecture of Figure 3. m hash generators to generate hashes, k hash units to compare the hashes and s string units to perform the string comparison. As matches are signaled only after the data in the window is compared against the actual string, there is no possibility of false-positives forming the basis for our zero false-positive claim. If matches need to be generated on a per-channel basis in order to identify the stream carrying the malicious data, an additional output resolver is added to demultiplex the output from the string units.

Systems utilizing hashing have to deal with hash collisions. In an IDS system, the set of pattern strings is dynamically changing and hence it is impossible to construct a perfect hash function. We deal with hash collisions in the PH engine using an overflow TCAM. When a string cannot be loaded into the engine because of a hash collision, it is loaded into the overflow TCAM. During the querying phase the overflow TCAM is searched in parallel to the engine.

III. Bloom filter Accelerator

A. Overview of the Bloom Filter

The Bloom filter uses n_b independent hash functions to generate n_b non-distinct hash values in the range *1 to* m_b for every l byte string. This m_b bit vector can be considered as a fingerprint of the hashed string. The initial programming phase involves generating the m_b bit fingerprint for each string in the set S and then using these fingerprints to set bits in the m_b bit Bloom array. Previously set bits are never reset by subsequent fingerprints and hence the Bloom array can be considered as an logical OR of all the fingerprints of strings in set S. In the comparison phase, the m_b bit fingerprint is generated for every $l-byte$ string in the data stream and compared against the value stored in the Bloom array. If at least one of the bits set in the fingerprint are not set in the Bloom array, then the string can be safely ignored. If however, the fingerprint matches, then further comparisons need to be made to ensure that the string is not a false positive. The characteristics of Bloom filter ensure that the false positive rate can be decreased if either we decrease the number of elements in the set n_b, increase the length of the bit vectorm_b or increase the number of hash functions k_b.

B. Accelerator Design

The architecture of our Bloom filter is shown in Figure 4. The Bloom filter consists of a hash generation block that uses n_b independent hash functions to generate an m_b bit fingerprint which is compared in the Bloom array to generate a single match bit. The architecture of the Bloom array is shown in Figure 4. The Bloom array consists of m_b independent Bloom cells and a single comparator.

The Bloom cell shown in Figure 5 consists of a D flip flop and a few additional gates. The load line (active high) is used to program the Bloom filter and hence program each of the Bloom cells. During the load process (i.e when load is at logic 1), the D flop gets set if hash is asserted. Once set, the feedback ensures that the D flop cannot be reset through either the hash or the load lines. In order to reset the entire Bloom filter, a separate reset connected to the clear input of the D flop is used. Loading to Bloom cells refers to the process of storing information about a set of strings in the Bloom filter so that they can be compared against strings in the incoming data stream.

In the comparison phase (i.e load is logic 0), the propagate line is asserted if either the cell is not referenced (i.e. hash is logic 0) or if the cell was referenced and a match was found (i.e hash is logic 1 and D-flop has been set). Using this modified signal simplifies the comparator reducing it to a set of AND gates instead of the more expensive XOR gates.

The truth table for the Bloom cell is shown in Table I. The value of the propagate line is ignored when load is asserted as loading and comparison are not performed at the same time. When in the comparison phase (i.e load is logic 0) the values in the propagate column are the values fed to the comparator. Unless a mismatch occurs in this Bloom cell (i.e this bit is set in the fingerprint of the the string currently being compared

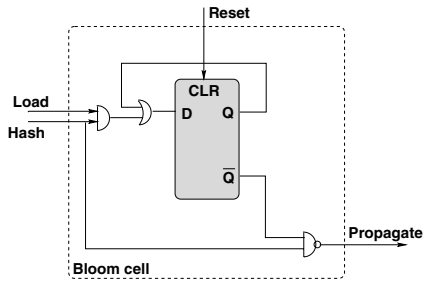

Figure 5. Architecture of the Bloom cell

but this bit was never set in the loading phase) propagation must be allowed. This scenario is represented in the second row of the truth table. The values in the first and fifth rows refer to cases in which the Bloom cell is not queried (i.e this bit is not set in the fingerprint of the string currently being compared) while the values in the seventh row refer to the case where this Bloom cell matched (i.e this bit is set in the fingerprint of the string currently being compared and also during the loading phase).

For all but the second row in the truth table, this particular Bloom cell cannot predict a mis-match and hence allows values from other cells to propagate through the comparator to the output, hence the name.

C. Advantages

Applications of Bloom filters in string-matching has traditionally focused on achieving lowest possible false positive rates. Architectures like those proposed in [20] use a large number of hash functions (e.g. 35 hash functions used in [20]) combined with significant amount of memory (e.g. $\frac{m_b}{n_b} = 50$ in [20]) to achieve low false positive rates. Our approach is based on using a much small number of hash functions (one or two depending on the filter selected. see Table II) and memory ($\frac{m_b}{n_b} = 1$ *or* 2 depending on the filter selected) to achieve a significant reduction in the amount of work done by the PH engine.

The performance of Intrusion detection systems (generally those that use hash based string matching engines) typically depends on the mix of malicious data in the data stream. Such systems are usually designed with a fail-close mode whereby the intrusion detection process is stopped if the system is excessively overwhelmed and data is allowed to flow into the system. The Bloom filter based accelerator when installed to process data flowing into such systems would help the system (providing a boost) when needed. The Bloom filter based accelerator can be bypassed under normal circumstances to minimize the latency through the IDS system.

D. Integration

The flow of data through the system is shown in Figure 6. The data stream is fed through to the Bloom filter and also to the dispatcher. This allows the Bloom filter to be bypassed if required when the data rate is low. When passing through the Bloom filter, a miss from the Bloom filter simply forwards the data window. On the other hand, if the Bloom filter indicates a possible match, the same l byte string from the data window

TABLE I
BLOOM CELL TRUTH TABLE

Q	Load	Hash	Q+	Propagate	Meaning
0	0	0	0	1	Propagation is allowed
0	0	1	0	0	Propagation NOT allowed
0	1	0	0	1	Ignored (loading phase)
0	1	1	1	1	Ignored (loading phase)
1	0	0	1	1	Propagation is allowed
1	0	1	1	1	Propagation is allowed
1	1	0	1	1	Ignored (loading phase)
1	1	1	1	1	Ignored (loading phase)

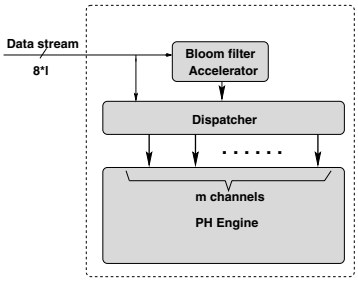

Figure 6. (a)Architecture of the complete system

is now fed to the PH engine through the dispatcher which performs a more thorough comparison. The packet dispatcher is a simple load balancer that is used to dispatch data to the different channels of the string match engine.

Normally, data flows through the packet dispatcher and through the PH engine. However, if the dispatcher observes that the PH engine is getting overloaded, the data stream is routed through the Bloom filter which reduces the rate of data flowing into the PH engine, as some of this data is processed completely by the Bloom filter.

Due to the asymmetric nature of processing in the PH engine (i.e some data windows are eliminated in the hash comparison phase, while others need to be forwarded to the third,string comparison phase), the PH engine may sometimes get overwhelmed and start becoming a bottleneck. Our architecture uses the Bloom filter as an aid in reducing the load on the PH engine instead of using it as an extensive filter with very low false positive. Hence, we get away with higher false positive rates. Also, additionally our strategy is completely hardware based and guarantees zero false positive string matching primarily due to the string comparison phase in the PH engine. As we store the strings in a separate table, we can store additional information about the strings (for example what actions needs to be taken) which allows our system to comprehensively deal with different possible actions.

E. Handling Multiple Sized Strings

Our discussion up to this point has dealt with processing strings of a fixed size using the Bloom filters. However in order to handle real life traffic, our system can be extended to deal with strings of multiple sizes. Let the lengths of the strings vary from l_{min} to l_{max}.

Strings whose length lie in the range $l+1$ to $2l$ can be written as two strings of length l. These two strings can now

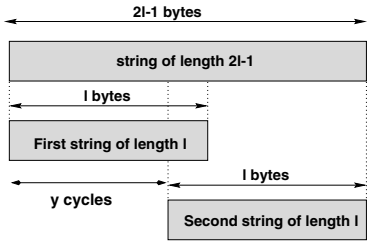

Figure 7. Handling multiple size strings using a single PH engine

be loaded into the Bloom filter based accelerator and the PH engine configured for processing strings of size l. Generalizing this further, strings whose length varies between $(x-1)l+1$ and $(x)l$ can be written as x strings of length l. A string of length $l+y,\ y<l$ can be detected by consolidating two matches generated y cycles apart. This strategy requires a stateful external processor to consolidate matches and compromises on capacity of the PH engine to offer more flexibility. Figure 7 shows an example where a string of size $2l-1$ is broken into two strings of size l which are $l-1$ cycles apart.

IV. Throughput Analysis

In order to estimate the affect of adding the Bloom filter based accelerator to the system, we compute the number of bytes passing (i.e throughput) through the system. The following notations are used:

- m **:** Number of channels of data (i.e. hash generators) in PH engine
- k **:** Number of hash comparison units in PH engine
- s **:** Number of string comparison units in PH engine
- p_h **:** Probability of a hash comparison without a collision
- q_h **:** Probability of a hash comparison colliding
- p_s **:** Probability of a string comparison without a collision
- q_s **:** Probability of a string comparison colliding
- f_b **:** The false positive rate of the Bloom filter
- m_b **:** The number of bits in the Bloom array
- n_b **:** The total number of strings loaded into the system
- f **:** The clock frequency (in Hz) at which the PH engine operates
- R^{PH-BF} **:** The number of bytes that can be processed by the system per second by the system (i.e. Throughput).

In order to save space, we provide the rationale and list the throughput of the PH engine R^{PH}. A detailed approximation is provided in [23].

The delay d experienced in advancing the window consists of the delay in generating the hash d_g, the delay in comparing the hash d_h and the delay in performing the string comparison d_s. Ideally, all three of these should be equal to one. Collisions in the hash units and in the string units cause d_h and d_s to be greater than one.

$$d = d_g + d_h + d_s \tag{1}$$

The Throughput of the PH engine depends on the characteristics of the data stream. If the data stream contains a high mix of suspicious strings, more references are made into the string comparison stage causing more processing delay and hence reducing the overall throughput. The best and the worst case throughput of the PH engine are given by:

$$R^{PH}_{best} = \frac{f \cdot m}{1+\overline{n}_h} Bytes/second \tag{2}$$

$$R^{PH}_{worst} = \frac{f \cdot m}{1+\overline{n}_h+\frac{\overline{n}_s}{l}} Bytes/second \tag{3}$$

When a Bloom filter is added into the system, the Bloom filter discards some of the data in the system forwarding a selected data stream to the PH engine for further processing.

Bloom filters have been extensively studied and it is well known [24] that the false positive rate of a Bloom filter fp is dependent on the number of bits in the Bloom array m_b, the number of independent hash functions in the hash generator k_b and the number of strings inserted into the Bloom filter n_b. The false positive rate of the system is the ratio of number of unsuspicious strings that were forwarded to the PH engine to the total number of strings processed by the Bloom filter. It is an indicator of the amount of un-suspicious strings sent to the PH engine. The false positive rate of the Bloom filter [24] is:

$$f_b = (1-e^{\frac{-n_b k_b}{m_b}})^{k_b} \tag{4}$$

Let the total amount of data that can be processed by our system be denoted by R^{PH-BF} The amount of data being processed by the PH engine R^{PH} is given by:

$$R^{PH} = R^{PH-BF} \times f_b \tag{5}$$

If the effects of the dispatcher are neglected and we assume that the Bloom filter can always keep up with the requirements of the PH engine, then the throughput of the system can be computed by re-arranging and substituting the true value of R^{PH} from Equations 2 and 3. The resulting throughput of the system R^{PH-BF} is now given by:

$$R^{PH-BF}_{best} = \frac{R^{PH}_{best}}{f_b} = \frac{\frac{f \cdot m}{1+\overline{n}_h}}{f_b} Bytes/second \tag{6}$$

$$R^{PH-BF}_{worst} = \frac{R^{PH}_{worst}}{f_b} = \frac{\frac{f \cdot m}{1+\overline{n}_h+\frac{\overline{n}_s}{l}}}{f_b} Bytes/second \tag{7}$$

V. Experimental Results

In order to report the results in this section, we synthesized the design using Synopsys design compiler [26] using library files from Artisan targeting the $180nmTSMC$ fabrication process. we extract the clock frequency from Synopsys design compiler [26] and use it along with the simulation data to calculate the projected throughput of the system. We believe this to be an accurate estimate of the performance of the engine when integrated into an network processor or other such processors. Although the engine can operate at speeds up to $350MHz$, we report results at $250MHz$ due to the limitations of commercially available SRAM IP cores for the $180nmTSMC$ process. Assuming SRAM cores that operate at $350MHz$ were available, the throughput as shown in Figure 8 would increase proportionally.

Traditionally very long Bloom filters (i.e. with $\frac{m_b}{n_b}$ around 50) have been used in string matching applications [20].

TABLE II

CHARACTERISTICS OF THE BLOOM FILTERS

Bloom filter	$\frac{m_b}{n_b}$	k_b
Filter #1	2	2
Filter #2	3	1
Filter #3	3	2

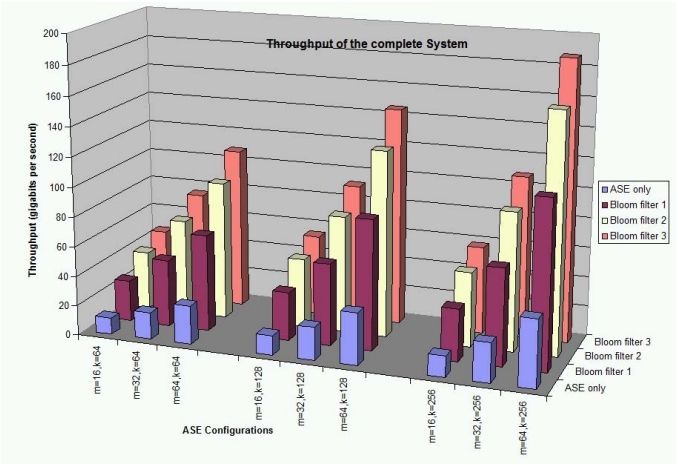

Figure 8. Number of bits processed through different configurations checking for 16000 strings operating at a clock frequency of 250MHz

These approaches required very large Bloom filters in order to guarantee low false positive rates. On the other hand we use Bloom filters solely to accelerate the string matching process and hence can get away with a much higher false positive rate. It can be observed from [24] that allowing for a slightly higher false positive rate significantly reduces the length of the Bloom filter Accelerator. We experimented with three of the smallest possible Bloom filters in order to minimize the resource consumption. Assuming larger Bloom filters were available, then higher throughput (as a result of lower false positive rates in Bloom filter accelerator) can be achieved.

Figure 8 gives the projected throughput for various configurations of the system. The configurations of the three Bloom filters listed in Figure 8 are given in Table II.

These results match reasonably well with the analytical results calculated from Equations 11 and 12 for the Bloom filter configuration of Filter #1 is shown in Figure 9. When calculating R_{best}^{PH-BF} and R_{worst}^{PH-BF} in Section 3, we consider data streams containing strings none of which have been loaded into the engine and data streams that contain only strings loaded into the system. The difference between the experimental values and the calculated values can be attributed to the randomness of the experimental data. It can be observed that the experimental R closely matches the average throughput, i.e $\frac{R_{best}^{PH-BF}+R_{worst}^{PH-BF}}{2}$.

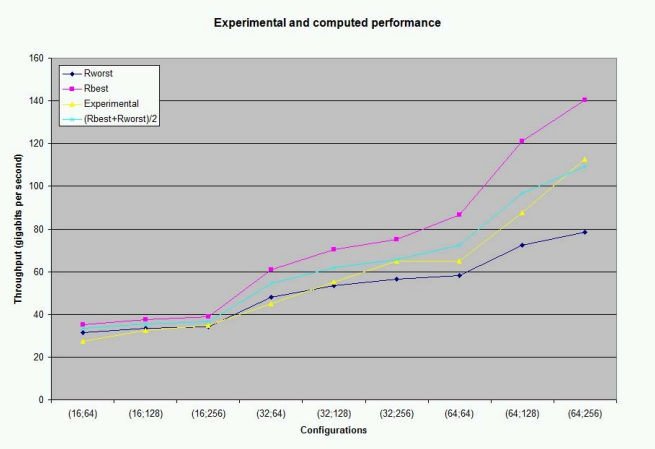

Figure 9. Comparison of expected and computed speeds of the system containing Bloom filter with $\frac{m_b}{n_b} = 2$ and $k_b = 2$

REFERENCES

[1] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," *Proceedings of the 13th Systems Administration Conference*, 1999.

[2] M. Fisk and G. Varghese, "Fast Content-Based Packet Handling for Intrusion Detection," *Technical report UCSD CS2001-0670*, 2001.

[3] Internet Speed Mark in Guiness World Records Book, *www.eurekalert.org*,2006.

[4] Intel Corporation, "IXP2400 Handbook," *http://www.cs.ucr.edu/ bhuyan/ cs203A/IXP2400.pdf*, Oct. 2006.

[5] R. Haas, C. Jeffries, L. Kencl, A. Kind, B. Metzler, R. Pletka, M. Waldvogel, L. Freléchoux and P. Droz, "Creating Advanced Functions on Network Processors: Experience and Perspectives," *IEEE Network*, 17(4), pp 46-54, 2003 2001.

[6] S. Staniford C. J. Coit and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *DARPA Information Survivability Conference and Exposition*, 2001.

[7] D. Knuth, J. H. Morris and V. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing, 6(2):323350*, 1977.

[8] J. Moore and R. Boyer, " A fast String Searching Algorithm," *Communications of the ACM*, 1977.

[9] M. Corasick, A. Aho, " Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, 1975.

[10] B. Commentz-Walter, "A String Matching Algorithm Fast on the Average," *Proc. ICALP'79, LNCS v. 6, pages 118-132*, 1979.

[11] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGA," *FCCM*, 2001.

[12] Z. Barker and V. Prasanna, "Time and Area Efficient Pattern Matching on FPGA," *FPGA*, 2004.

[13] D. Caraver, R. Franklin and B. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *FCCM*, 2002.

[14] R. Katz, F. Yu and T. Laskhman, "Gigabit Rate Packet Pattern Matching with TCAM," *International Conference on Network Protocols (ICNP)*, 2004.

[15] B. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *ACM, 13(7):422-426*, 1970.

[16] S. Cohen and Y. Matias, "Spectral Bloom filters," *Proc. SIGMOD*, 2003.

[17] S. Rhea, and J. Kubiatowicz, "Probabilistic location and routing," *Proc. INFOCOM*, 2002.

[18] Fan, L., Cao, P., Almeida J., Broder A., "A summary cache: a scalable wide-area web cache sharing protocol," *Proc. IEEE / ACM transaction on Networking*, 8 281-293, 2000.

[19] C. Bernard, J. Kilian, R. Rubinfield and A. Tal, "The Bloomer Filter: An Efficient Data Structure for Static Support Lookup Tables," *Proc. Fifteenth ACM-SIAM symposium on Discrete Algorithms*, pp30-39, 2004.

[20] T. Sproull, J. Lockwood, S. Dharmapurikar and P. Krishnamurthy, "Deep Packet Inspection using Parallel Bloom Filters," *Symposium on High Performance Interconnects (HotI)*, 2003.

[21] M. Necker, D. Contis and D. Schimmel, "TCP-Stream Reassembly and State Tracking in Hardware," *Proc. 10th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, 2002.

[22] L. Andreas, L. Lucas and S. Stefan, "An analysis of FPGA-based UDP/IP stack parallelism for embedded Ethernet connectivity," *Proc. NORCHIP Conference*, , pp94-97, Nov. 2005

[23] P. katta, M. Nourani and R. Panigrahy, "String Matching using Parallel Hashing," *Proc. Parallel & Distributed computing* , 2006.

[24] L. Fan, P. Cao, J. Almeida, and A. Broder, "Bloom Filters - the math," *http://www.cs.wisc.edu/ cao/papers/summary-cache/node8.html*, 2003.

[25] Altera Corporation, "Stratix-II Device Handbook," Dec. 2005.

[26] Synopsys Inc., "User Manuals for SYNOPSYS Toolset Version 2005.06," 2005.