Computer Engineering and Intelligent Systems
ISSN 2222-1719 (Paper) ISSN 2222-2863 (Online)
Vol.4, No.2, 2013

Packet Inspection on Programmable Hardware

Benfano Soewito
Information Technology Department, Bakrie University, Jakarta, Indonesia
E-mail: benfano.soewito@bakrie.ac.id

Abstract

In the network security system, one of the issues that are being discussed is to conduct a quick inspection of all incoming and outgoing packet. In this paper, we make a design packet inspection systems using programmable hardware. We propose the packet inspection system using a Field Programmable Gate Array (FPGA). The system proposed consisting of two important parts. The first part is to scanning packet very fast and the second is for verifying the results of scanning the first part. On the first part, the system based on incoming packet contents, the packet can reduce the number of strings to be matched for each packet and, accordingly, feed the packet to a verifier in the second part to conduct accurate string matching. In this paper a novel multi-threading finite state machine is proposed, which improves the clock frequency and allows multiple packets to be examined by a single state machine simultaneously. Design techniques for high-speed interconnect and interface circuits are also presented. The results of our experiment show that the system performance depend on the string matching algorithm, design on FPGA, and the number of string to be matched.

Keywords: Packet inspection, string matching, Field programmable gate array, Traffic classification

1. Introduction

Deep packet inspection (Varghese, 2005) is one of the most promising techniques to provide the lacking security of the Internet. The packet inspection system is built in network instruction detection. The heart of signature-based deep packet inspection is a string matching engine, which identifies suspicious activities by comparing network packet with predefined patterns. These predefined patterns are defined as a set of rules, there are 3305 such rules defined by Snort version 2.4. Each rule consists of two types of strings to be matched: one is header strings with determined position in packet header (e.g., source/destination network address and source/destination port number); another is payload strings with probabilistic position in packet payload (e.g., network worms and computer virus). A suspicious activity is detected when both header strings and at least one of payload strings is matched on the packet.

The simple string matching engine can be the bottleneck of a deep packet inspection system, because of to tens Gigabit per second network traffic and thousands of possible attack rules. Further, the starting position of payload string might be probabilistic; hence it is necessary to scan every byte of a packet. Currently, existing software-based packet inspection can barely keep up with data rate at a few hundred Megabitsper second. Hence different hardware approaches (Baker, 2004; Aldwairi, 2005; Dharmapurikar, 2004; Piyachon, 2006; L. Bu, 2004) have been proposed to this difficult problem. However, they are either lacking performance, scalability to traffic rate and attack rules, or too complicated to design and operate.

To address these concerns, this paper proposes a simple but efficient architecture based on scalable classifiers and novel multi-threading finite state machines (FSMs). Classifier arranges the incoming packet to three categories: malicious, suspected or benign. FSMs are used to further verify whether the suspected packet is malicious one. The key to achieving high performance of this architecture is employing multiple small and fast FSMs. Each of these FSMs searches for a portion of rules on a suspected packet only, and some of the benign packets are offloaded from FSMs.

The soundness of this architecture is based on the following observations from Snort 2.4, which is used in our experiments: 329 unique header rules; 172 rules have header string only; the maximal number of payload strings for particular group header strings is 97; and most packets (85%) are benign packets (Attig, 2005). Our FSMs based on Aho-Corasick is a novel multi-threading FSM, which improves FSM clock frequency and allows multiple packets to be examined by a single FSM simultaneously. String partition and high-throughput interconnect are employed to further improve the system performance. Our experimental results have demonstrated that this proposed architecture achieves superb throughput.

2. Related Works

Packet Inspection and Network Intrusion Detection Systems have been studied in different forms since Denning's classic statistical analysis of host intrusions (Denning, 1987). Yet the capabilities of current packet inspections are lacking the scalability with growing threats (Varghese, 2005) and increasing speed of internet. Different approaches have been proposed to speed up strings matchings, either in software algorithms or using hardware optimized techniques.

The most notable matching algorithms include Boyer-Moore (Boyer, 1977), Aho-Corasick (Aho, 1975), Commentz-Walter (Commentz-Walter, 1979), and Wu-Manber (Wu, 1994). Most of these algorithms are either optimized for average performance, or need a large number of memory access, hence they are suitable only for software implementation. Tuck, et al. (Tuck, 2004) extended worst-case bound of Aho-Corasick using bitmap compression and path compression to reduce the amount of memory needed. This algorithm is very fast, however it requires a excessively large memory bus to eliminate memory accesses bottleneck.

Recently, interesting hardware optimized techniques have been proposed for string matchings. These hardware-based techniques employ commodity technologies such as Bloom Filter (Dharmapurikar, 2004), Network Processors (Benfano, 2009), TCAM (YU, 2004) and FPGAs (Baker, 2004; Song, 2005; Dharmapurikar, 2004; Aldwairi, 2005). Bloom Filter is a powerful technique to quickly isolate the potential malicious packets. However, a large fast on-chip memory is required to implement these multiple Bloom Filters to reduce its well-known high false positive rate. Meanwhile, Network Processor (NPs), programming multiprocessors optimized for packet processing, have been evaluated for multiple strings matching (Benfano, 2008). These NPs-based approaches use hardware hashing engine provided by most NPs. However, its performance is not scalable due to its general purpose for simple packet processing and

relative small on-chip memories. TCAM is very fast and particularly suitable for wild card patterns, however, it suffers from excessive power consumption and high cost.

FPGAs are also used for fast string matching due to its fine-grain parallelism, hardware performance, and easy of implementation. The majority of prior work in this area focuses on efficient ways to map a given rule set down to a specialized circuit. Although very fast, these purely FPGA based approaches are known to exhaust most of the chip resources with just a few thousand patterns. In our work, patterns are partitioned to reduce the complexity of each FSMs with the guidance of the classifier, and high-throughput interconnect are designed to further improve the system performance. The hardware description language and software design kit has made FPGA even attractable; however, this automatic design flow doesn't relieve the designer of the need to deal with parallelism. In our work, a novel multi-threading FSM is proposed, which improves FSM clock frequency and allows multiple packets to be examined by a single FSM at the same time.

3. Methodology

Figure 1 shows our proposed packet inspection system. A packet enters the system, first processed by the dispatcher, then by a classifier, and further processed by a verifier if necessary. Based on the decision, the packet end-up with either discarder or forwarder. The key components are classifier and verifier. Classifier classifies the incoming packet to three categories: malicious, suspected or benign. The verifier further determines whether the suspected packet is a malicious one. If the packet turns out to be benign, this is a false positive. Therefore, the right partition will have to balance the false positive rate and offloading from verifiers.

3. 1. Packet Classifier

The classifier that we consider have 6 dimensions: source and destination addresses that requires exact or prefix matches, source and destination ports that requires exact or range matches, a protocol number that requires an exact match, and contents that requires an approximate match. Comparing with previous work, our classifier checks both header strings and payload strings. Therefore, our classifier must be scalable, high performance, low false-positive rate and easy of updating.

The actual classification is straightforward once attack patterns are partitioned based on off-line profiling. Our detail partitioning procedure is as following. First, we divide the rules set into two groups: rules with header only, and rules with header and contents. For rules with header only, we can further divide it into several groups with each group sharing one same property: such as the same source ports. Choosing which field to classify and how many groups to partition are determined by the number of rules in that group, and the ratio of network traffic belonging to that group. This value will be collected through real network traffic profiling. Re-ordering can also be employed to reduce the processing complexity.

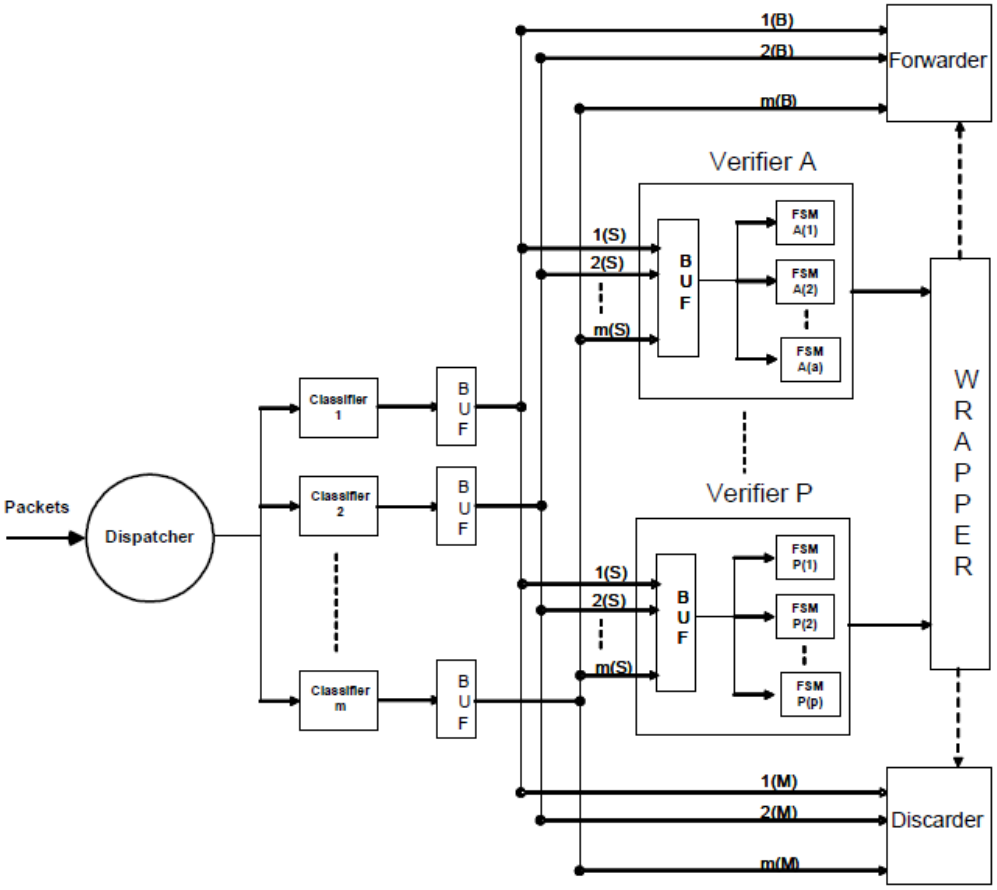

Figure 1. Proposed Packet Inspection System.

For rules with both header and contents, the above similar approach can be taken. First, a set of rules is formed into one group; then this group of strings is further classified into subgroups according to their length; finally a multiprocessor is used to classify the packets. Using a multicore to quickly isolate the suspected packet has been proposed in (Benfano, 2008), here we utilize it for approximate string matching in classifiers.

3.2. Multiprocessor

We utilized a Network Processors as a multiprocessor in our experiment. Network Processors (NPs), a specialized multiprocessor, can provide flexibility and high performance for string matching. Network Processors is a programmable hardware infrastructure which is optimized for packet processing, has become a promising build block for high performance and programmable routers. However, the potential of NPs for IDSs is not fully explored due to the lack of suitable string matching algorithms optimized for NPs but also the difficulty of programming NPs. Most of the available string matching algorithms target to general purpose computer architectures. They either cannot be efficiently executed by NPs which have severe limits on fast memory accesses or cannot fully take advantage of the NPs high-level parallelism. On the other hand, NPs high-level parallelism and heterogeneous architecture are difficult to program for high performance. Today, NP programmers use hand-tuned and manually resource mapping approach, which is

not efficient for early design decision making. Therefore an effective design methodology is required to explore string matching algorithm optimized for NPs (Benfano, 2008). We used an automatic simulation environment, which is able to quickly profile a promising string matching algorithm, application map and performance estimation. Our simulation environment consists of uniprocessor profiling and mapping technique on to multiprocessor.

3.3. String Matching and Mapping Technique to Multiprocessor

The heart of our classifier is a string matching algorithm, which is a very computational intensive task. In this experiment, we use Aho-Corasick algorithm (Aho, 1975). The Aho-Corasick (AC) algorithm is able to match multiple strings simultaneously by constructing a state machine. Starting from an empty root state, each string to be matched is represented by a series of states in the machine, along with pointers to the next appropriate state. This pointer is added from each node to the longest prefix of that node which also leads to a valid node in the machine. The major drawback of AC algorithm is possible of 256 fan-outs, which results in low memory efficiency.

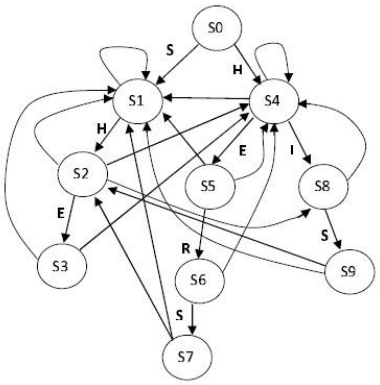

Figure 2. Example of AC State Machine

Figure 2 shows an example of AC state machine used to match “SHE”, “HERS” and “HIS”. Starting at state *s*0, state machine is traversed to state *s*1 or *s*4 depending on the input character is *s* or *h*. When an end state is reached, a string has been said to be matched. In the example in Figure 2, if state *s*7 is reached, string “HERS” has been matched. Each state in the machine has pointers to other states in the machine. If an input character is the next character in a string that is currently being matched, the algorithm moves to the next state in that string, otherwise, the algorithm follows a failure pointer to the first state of another string that begins with that character, or to the initial state of the machine if no other strings begin with that character. An example of this can be seen in Figure 2. If the current state of the machine is *s*5, the last input characters would have been “HE”. If the next input character were to be “R”, then the next state would be *s*6. If the next input character were not “R”, but instead, “S”, then the next state would follow a failure pointer to state *s*1, which is the starting point for the string “SHE”.

We start to analyse AC string matching algorithms using the uniprocessor profiling environment. We characterize them by processing cost, storage requirement and their scalability. Based on these results, we are able to understand workload behaviours of algorithms and identify their bottleneck. To show the

potential parallelism of matching operation, we represent the algorithm using an Annotated Directed Acyclic Graph (ADAG). These ADAGs show processing requirements in terms of instructions, memory accesses and communication cost. Also, the interdependency shown in ADAGs demonstrates the nature of the matching algorithm. Then we use random mapping technique to place theses ADAGs on to multiprocessor.

4. High-throughput Verifier

This section presents techniques to improve the throughput of FSM-based verifier modules. A novel multi-threading FSM structure is described in this section. In conventional FSM-based string matching operations, packets to be examined are fed to the FSM one by one in a serial manner. This is analogous with different tasks being processed by a single-threading microprocessor in a sequential order. Similar to multi-threading techniques improving microprocessor performance, the throughput of FSM-based string matching operations can be improved if multiple packets can be examined by the same FSM in parallel. In this paper, the FSM that can process multiple packets at the same time is referred to as a multi-threading FSM.

A conventional FSM can be represented by a generic sequential circuit model as shown in Figure 3 (a). The combinational circuit generates the FSM next state; while state registers store current FSM states. Operations of the FSM during string matching are depicted in Figure 3 (b). In the diagram, we use $P[i]$ to represent the ith byte of the packet to be examined by the FSM. $S[i]$ denotes the state that FSM reaches after reading the ith byte of the packet. Clearly, the FSM reads a byte at a clock cycle and processes a new packet only after the current packet is completely examined. Assume the FSM clock frequency is f and the packet contains N bytes of data. The total time for the FSM to process a packet is N/f.

In general, the complexity of the combinational circuit is proportional to the number of string patterns encoded in the FSM. Hence, if many string patterns are contained in an FSM, the propagation delay of the combinational circuit is increased and the maximum clock frequency of the FSM is reduced. A straight forward approach to address this problem is to apply re-timing techniques: dividing the logic propagation paths into sub-paths and inserting pipeline registers in between. Although re-timing techniques have been widely applied in designing high-speed data path circuits, the use of such techniques in FSM design is rarely reported. This is mainly because an FSM needs both current input and current state to generate its next state. While adding pipeline stages in the combinational circuit increases the clock frequency, it postpones the generation of FSM current state. As a result, the overall FSM performance is not improved. This is further illustrated in the following example.

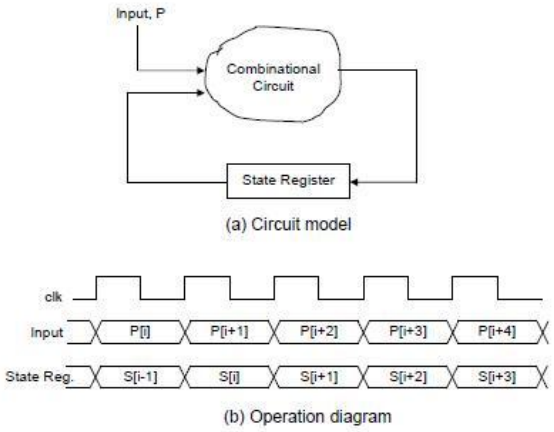

Figure 3. Finite State Machine.

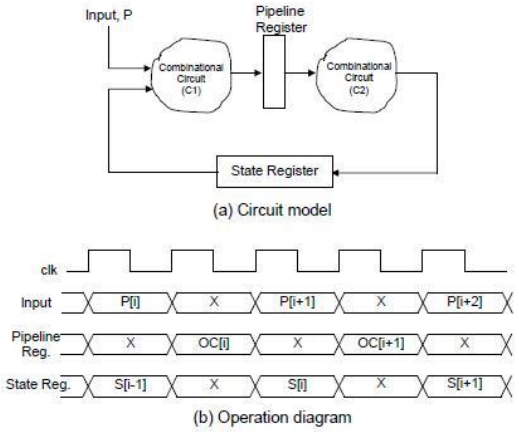

Figure 4. Pipelined FSM

Assume that the combinational circuit of the FSM shown in Figure 3 (a) is partitioned into two parts and pipeline registers are inserted between the partitioned circuits. The resultant pipelined FSM and its operation are sketched in Figure 4. Note that C_1 and C_2 are used to label the two partitioned circuits in the figure. At the ith clock cycle, $P[i]$, the ith byte of packet P, is fed to the FSM and the state register stores state $S[i\text{-}1]$, which corresponds to FSM input $P[i\text{-}1]$. The C_1 output, which is resulted from inputs $P[i\text{-}1]$ and $S[i\text{-}1]$, is latched into the pipeline register at the end of the ith clock cycle, and is further processed by C_2 during the $(i+1)$th clock cycle. The FSM state corresponding to input $P[i]$ is latched to the state register at the end of the $(i+1)$th clock cycle. Therefore, the FSM has to wait till the $(i+2)$th clock cycle to take the next input $P[i+1]$. Apparently, the pipelined design doubles the FSM clock frequency but takes an input every two clock cycles. Hence, the time to process a packet is still N/f, which is the same as that of the conventional FSM..

During half of the operation cycles, the combinational circuits and registers in the above design do not produce or store valid data. For example, the data stored in the state register during the $(i+1)$th clock cycle are useless for the operation of the FSM. For a pipelined FSM, we refer to a clock cycle as a utilized cycle for a register or a combinational circuit if the register stores a valid state or the combinational circuit generates valid outputs. In addition, we call the clock cycles that the hardware does not store or generate valid data as its idle cycles. In Figure 4 (b), we use "X" to label the idle cycles.

An interesting property of pipelined FSMs is that data stored or produced during idle cycles don't affect the correctness of the FSM operation during its utilized cycles. This creates the possibility to let the idle FSM hardware process other packets without affecting the FSM operation designated for the original packet. Thus, multiple packets can be processed by a single FSM and a multi-threading operation is achieved. In network processing domain, there is virtually no dependency between packets, therefore, all packets can be processed in parallel. Figure 5 shows a two-threading FSM and its operations. During the odd clock cycles, data from Packet P_1 are fed to the FSM. In an even clock cycle, the FSM takes input from Packet P_2. As a snapshot of its operation, we assume at the ith clock cycle, where i is an odd number, data $P_1[i]$ (the ith byte of P_1) is fed to the FSM. Also, the FSM state register stores state $S_1[i\text{-}1]$, which corresponds to FSM input

$P_1[i\text{-}1]$. Thus, during the ith clock cycle, combinational circuit C_1 computes partial results, which will be used by C_2 to generate FSM state $S_1[i]$ in the $(i+1)$th clock cycle. Parallel with the computation for Packet P_1, circuit C_2 computes the FSM state $S_2[i\text{-}1]$ for Packet P_2 during the ith clock cycle. Consequently, at the $(i+1)$th clock cycle the state register stores state $S_2[i\text{-}1]$ and circuit C_1 processes input $P_2[i]$ from Packet P_2. Note that for an FSM with M pipeline stages, M packets can be processed simultaneously. Hence, we refer to it as a M-threading FSM.

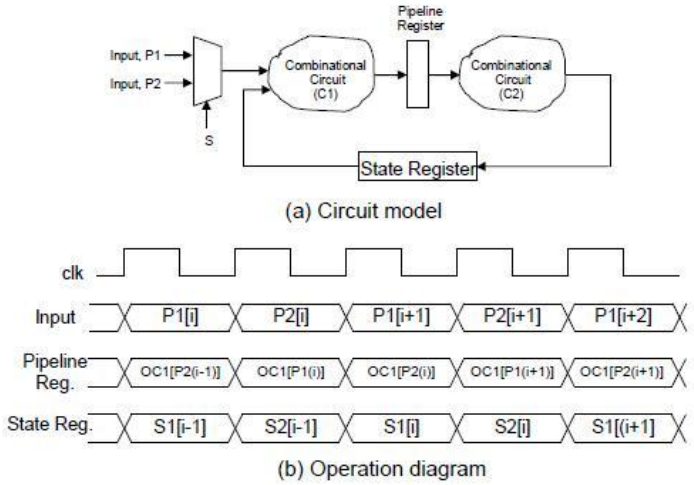

Figure 5. Multi-treading FSM.

If we ignore the performance penalty caused by the FSM input multiplexer and pipeline stage registers, the clock frequency of a M-threading FSM can be M times faster than that of a conventional FSM. Although it still needs the same amount of time N/f to process a single packet, the M-threading FSM can process M packets at the same time and, hence, increases the throughput by M times. To maximize the system throughput, M is preferred to be as large as possible. On FPGA implementations, the value of M can be maximized by adding a pipeline stage after each look up table (LUT). Modern FPGAs have abundant D flip flops (DFFs) to support this deep pipeline scheme. This observation is supported by our experimental results to be presented in Section 5. When the value of M is large, the M-*to*-1 multiplexer in front of the FSM will become the bottleneck that limits the system performance.

By using the multi-threading FSM design technique, the signal propagation delay caused by logic components can be minimized to the delay of a single LUT. As a result, the signal delay between two adjacent pipeline stages is dominated by interconnect delay. To reduce FSM interconnect delay, long wires and global FPGA routing resources should be avoided in FSM implementations. This implies that both the area occupied by the FSM and the fan-out numbers of FSM nets should be kept small. Both of the above conditions can be satisfied if the number of states (number of string patterns) encoded in the FSM is small. To achieve this goal, after partitioning the entire Snort patterns into different classes, we further divide the patterns of a class into smaller subsets. Small FSMs are designed for the partitioned subsets to minimize FSM interconnect delay.

5. Experimental Results

Experiments have been conducted to study the effectiveness of the proposed techniques. The AC string matching algorithm can achieve the maximum throughput using multiprocessor 8x4 or 32 cores as shown in Table 1. From the table it can be seen that the experimental results using 32 cores and 64 cores does not vary much or it can be said there is no difference. This shows that the increasing number of core is not linear with increasing the system performance.

The FPGA hardware used in our study is Xilinx Virtex 4 FX100 device (Xilinx, 2010). String matching rules from the Snort (Snort, 2009) are used to specify the functionality of the FPGA FSMs. In the experiments, we first convert the Snort rules into state transition tables and, consequently, generate Verilog codes that describe FSM behaviors. The Verilog codes are given as input to an FPGA design automation tool to perform logic synthesis, and circuit placement and routing (P&R). The circuit performance and resource utilization are obtained from post-P&R reports and static timing analysis. To implement the proposed multi-threading FSMs, gate-level net lists of the synthesized FSMs are fed to an in-house re-timing program to add pipeline registers. The modified net lists are given as the inputs of the FGPA P&R tool to implement the FSMs on the target FPGA platforms. In the experiments, we also vary the number of string matching rules to be encoded into the FSMs to study how it affects the FSM performance. For the convenience of discussion, the number of string matching rules encoded in an FSM is also referred to as the size of the FSM. For example, if 200 string matching rules are implemented by an FSM, we call the FSM has a size of 200 in the following discussion.

Table 1. Throughput depend on topology of multiprocessor

Depth (*d*)	Width (*w*)	Throughput (*Gbps*)
4	8	7.78
8	2	8.44
8	4	10.57
8	8	10.55

Figure 6 shows the maximum clock frequency versus the thread numbers of multi-threading FSMs. Data collected from FSMs with sizes of 20, 50, 100, and 200 are displayed in the figure. The data points, whose horizontal-axis coordinates (FSM thread numbers) are 1, correspond to conventional FSM implementations. Clearly, the multi-threading FSM design technique significantly increases the FSM clock frequency. Since an FSM takes a byte of data at a clock cycle, its throughput will be eight times of its clock frequency. From the experimental results, we find that 50 is the optimal FSM size for the target hardware platform. With the multi-threading technique, the maximum clock frequency of the FSM with the size of 50 is above 500MHz as shown in the figure. Thus, its maximum throughput is above 4 Gbits/s.

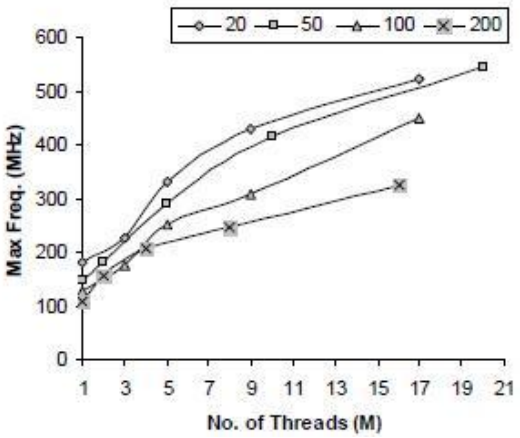

Figure 6. FSM Clock Frequency.

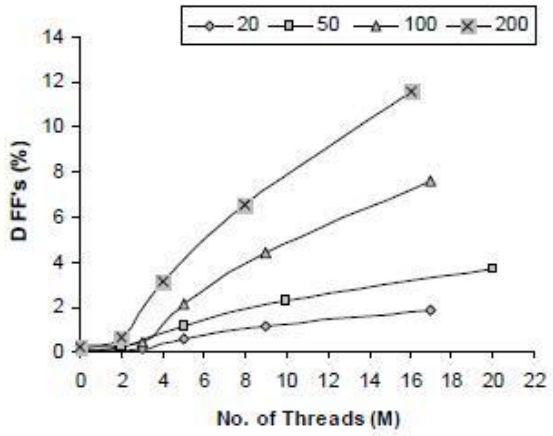

Figure 8. Utilization in Multi-threading FSMs.

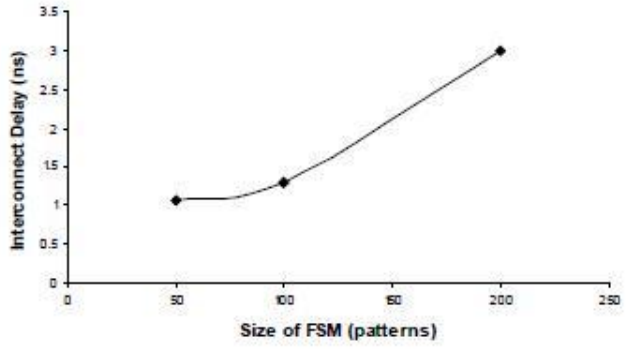

Figure 7. Interconnect Delay with Different FSM Sizes.

Note that for a *M*-threading FSM its realized clock frequency is less than what is predicted (*M* times of the conventional FSM clock frequency) in Section 4.1. This is mainly because signal paths of FPGA FSMs are routed using fixed-length FPGA interconnect resources and their delays are not always proportionally scaling down as the delay of logic components when more pipeline stages are added. For a given FSM design, the maximum number of threads that can be implemented is determined by the logic depth (the level of logic gates) of the FSM combinational circuit. That's why the maximum thread numbers for the reported FSMs are slightly different. In the proposed system, the entire Snort is first partitioned into classes and string matching rules in each class are further divided into subsets with sizes of 50. Hence, FSMs with sizes larger than 200 are rarely used in the proposed system and their performance is not include in the above figure.

Figure 6 also indicates that the maximum clock frequency that can be achieved by multi-threading FSMs degrade with the sizes of FSMs. To illustrate the cause, interconnect delays on the critical paths of different-sized FSMs are plotted in Figure 7. The results are consistent with our previous analysis that large FSMs normally have large interconnect delays and, consequently, lower clock frequencies, even with the use of multi-threading design techniques.

The hardware overhead caused by the multi-threading FSM design approach is also studied in our experiments. Figure 8 shows that DFFs (in terms of the percentage of the total DFFs on the Virtex 4 FX100 device) used in the design increase proportionally with the number of threads in the FSMs. Note that the number of LUTs used in the design will not be affected by the multi-threading technique and, thus, is not reported in the figure. Because of the DFF-rich architectures of FPGAs, the increase demand for DFFs in the multi-threading FSMs does not pose significant problems in system implementations. Even if the maximum thread number is used in the FSM design, the required DFF resource is not dramatically higher than that of LUTs (both are in terms of percentages of the total available resources on the FPGA platform). Thus, the use of multi-threading techniques will not significantly affect the number of FSMs that can be implemented on the FPGA platform.

6. Conclusion

In this work, a high-throughput packet inspection system is presented. It includes packet classifiers and string matching verifiers. The use of packet classifiers can significantly reduce verifier workload and hence improve the system throughput. Various techniques, including multi-threading FSM design and a novel high-speed FSM interface circuit, are developed to improve the performance of the verifier circuits. It is demonstrated that maximum FSM clock frequency can be increased several times by the multi-threading FSM design.

Experimental results show that the implemented verifier throughput is above 4~Gbits/s. Thanks to the pipelined interconnect and out-of-order execution techniques, the throughput of the verifier will not be significantly affected by the numbers of FSMs (or the number of string matching rules) included in the verifier. For the currently considered string matching rules, a single Virtex 4 FX100 device can contain all the verifiers. With continuously increasing number of Snort rules, a larger or multiple FPGA devices can be used to implement the verifiers in the proposed system. By taking advantage of the abundant multi-GBits/s transceiver modules in modern FPGA devices, communication between FPGA devices should not become the bottleneck of the system. In this study, the string matching rules are partitioned into eight classes. If the classifier can evenly dispatch the incoming packets into the eight verifiers, the system throughput will be around 32 Gbits/s. In addition, if the classifiers can off-load a portion of the benign packets from the verifier, the system performance will be further improved.

References

A. Aho and M. Corasick. (1975). Efficient string matching: An aid to bibliographic search in C*ommunications of the ACM*, 18, 333-343.

B. Commentz-Walter. (1979). A string matching algorithm fast on the average in *Proc. of the 6th International Colloquium on Automata, Languages and Programming*, volume 71.

D. Denning. (1987). An intrusion detection model in *IEEE Transactions on Software Engineering*, 13(2):222-232.

Benfano Soewito and Ning Weng. (2009). Concurrent workload mapping for multicore security systems in *Concurrency and Computation Practice and Experience*. Volume 21, number 10, pages 1281-1306.

Benfano Soewito and Ning Weng. (2008). Methodology for evaluating string matching algorithms on multiprocessor in *the 6th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA.* Pages 20-27.

F. Yu, R. H. Katz, and T. V. Lakshman. (2004). Gigabit rate packet pattern-matching using TCAM in ICNP '04 in *the 12th IEEE International Conference on Network Protocols (ICNP'04)*, pages 174-183, Washington, DC, USA.

G. Varghese. (2005). Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices. *Morgan Kaufmann*, 1st edition.

H. Song and J. W. Lockwood. (2005). Efficient packet classification for network intrusion detection using fpga, in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238{245, New York, NY, USA. ACM Press.

L. Bu and J. A. Chandy. (2004). FPGA based network intrusion detection using content addressable memories, in *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 316-317, Washington, DC, USA. IEEE Computer Society.

M. Aldwairi, T. Conte, and P. Franzon. (2005). Configurable string matching hardware for speeding up intrusion detection, in *SIGARCH Comput. Archit. News*, 33(1):99-107.

N. Tuck, T. Sherwood, B. Calder, and G. Varghese. (2004). Deterministic memory-efficient string matching algorithms for intrusion detection, in *Proc. of the IEEE Infocom Conference*, pages 333-340.

P. Piyachon and Y. Luo. (2006). Efficient memory utilization on network processors for deep packet inspection, in *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 71-80.

S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. (2004). Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52-61, Jan. 2004.

R. S. Boyer and J. S. Moore. (1977). A fast string searching algorithm. *Communication of the ACM*, 20(10):762-772.

S. Wu and Manber. (1994). A fast algorithm for multi-pattern searching. *Technical Report TR94-17*, Department of Computer Science, University of Arizona.

Xilinx, Inc. (2010). *Virtex-IV Pro and Virtex-IV Pro X Platform FPGAs: Complete Data Sheet*, http://www.xilinx.com.