2-24-2008

Implementing High-Speed String Matching Hardware for Network Intrusion Detection Systems

Ajay Mahajan
University of Akron, main campus

Benfano Soewito

Sai K. Parsi

Ning Weng

Haibo Wang

Implementing High-speed String Matching Hardware for Network Intrusion Detection Systems

Atul Mahajan, Benfano Soewito, Sai K. Parsi, Ning Weng and Haibo Wang
Department of Electrical and Computer Engineering
Southern Illinois University
Carbondale, Illinois, USA

Abstract

This paper presents high-throughput techniques for implementing FSM based string matching hardware on FPGAs. By taking advantage of the fact that string matching operations for different packets are independent, a novel multi-threading FSM design is presented, which dramatically increases the FSM frequency and the throughput of string matching operations. In addition, design techniques for high-speed interconnect and interface circuits for the proposed FSM are also presented. Experimental results conducted on FPGA platforms are presented to study the effectiveness of the proposed techniques and explore the trade-offs between system performance, strings partition granularity and hardware resource cost.

Keywords

Intrusion Detection System, Parallel Systems, Network Systems, FPGA-based Design, and Performance Evaluation

1 Introduction

Network Intrusion Detection System (NIDS) [6, 11, 16] is one of the most promising techniques to provide the lacking security of the Internet. The heart of signature-based NIDS is a string matching engine, which identifies suspicious activities by comparing network packets with predefined patterns, such as Snort rules. The design of string matching engine for NIDS application is very challenging due to the following reasons. First, the string matching engine needs to compare Internet traffic, at the speed of tens of Gigabits per second, with thousands of possible attack patterns. Second, the string matching engine needs to be easily updated to include new attack patterns. Currently, existing software-based NIDS can be easily updated but barely keep up with data rate at a few hundred Megabits per second. Meanwhile, various hardware architectures for NIDS applications [3, 1, 5, 7, 10] have been proposed. However, most of them are either lacking performance, scalability to traffic rate and attack rules, or too complicated to design and operate.

In this paper, we present a simple but efficient architecture based on scalable classifiers and novel multi-threading finite state machines (FSMs). Its block diagram is shown in Figure 1. The classifier arranges incoming packets into three categories: malicious, suspected or benign. Only suspected packets are fed to FSMs (verifiers) for further verification. In addition, classifiers confine the patterns that need to be checked for each suspected packet. These two features significantly improve the performance of the proposed architecture. 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 [2].

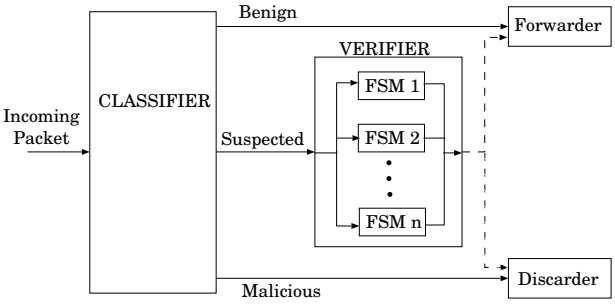


Figure 1. Proposed NIDS architecture.

The design of the scalable classifiers will be discussed in another work. This paper focuses on techniques to implement high-throughput FSMs, which can be used in the proposed architecture as well as other string matching hardware. We present a novel multi-threading FSM design,

which improves FSM clock frequency and allows multiple packets to be examined by a single FSM simultaneously. Techniques to implement high-speed interface circuits for multi-threading FSMs and pipelined interconnects between FSMs are also proposed. Experimental results on implementing the proposed circuit techniques on an FPGA platform are presented to demonstrate the proposed techniques and to study the trade-offs between system performance, strings partition granularity and hardware resource cost.

The rest of the paper is organized as follows. Related work is reviewed in Section 2. Techniques for high-speed verifier design are described in Section 3. Section 4 presents experiment results and Section 5 concludes this paper.

2 Related Work

NIDS have been studied in different forms since Denning's classic statistical analysis of host intrusions [6]. Yet the capabilities of current NIDS are lacking the scalability with growing threats [16] and increasing packet rate. Various hardware optimized techniques have been proposed for string matching. These hardware-based techniques employ commodity technologies such as Bloom Filter [8], Network Processors [9], TCAM [18] and FPGAs [3, 13, 14, 7, 1]. Bloom Filter is a powerful technique to quickly isolate the potential malicious packets. However, large fast on-chip memory is required to implement these multiple Bloom Filters to reduce its well-known high false positive rate. Meanwhile, Network Processors (NPs), programming multiprocessors optimized for packet processing, have been evaluated for multiple string matching. These NPs-based approaches use a 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 memory. TCAM is very fast and particularly suitable for wild card patterns, however, it suffers from excessive power consumption and high cost.

Recently, improved FSM based string matching hardware architectures have been reported in literature. The bit-split architecture presented in [15] focuses on minimizing the size of memory used to store next state pointers. However, due to the concern of the partial match vector size, an extremely large number of very small FSMs have to be used, resulting in complicated interconnect design and degraded hardware efficiency. The FSM presented in [4] improves the throughput by taking multiple bytes at each clock cycle. To combat the exponential increase of state numbers, complicated alphabet encoding and transition table compression have to be performed. Hence, during FSM operations certain pre-computation has to be conducted before next state selection. Since the pre-computation is independent of the FSM current state, the pre-computation and next state selection can be performed at different clock cycles, hence leading to a pipelined implementation. A major! difference between our proposed multi-threading FSM and the design in [4] is that in our design, pipeline registers can be placed at any level and, thus, it supports deep pipeline scheme. In the design of [4], pipeline registers must be before the next state selection circuit, consequently limiting the number of pipeline stages in their design.

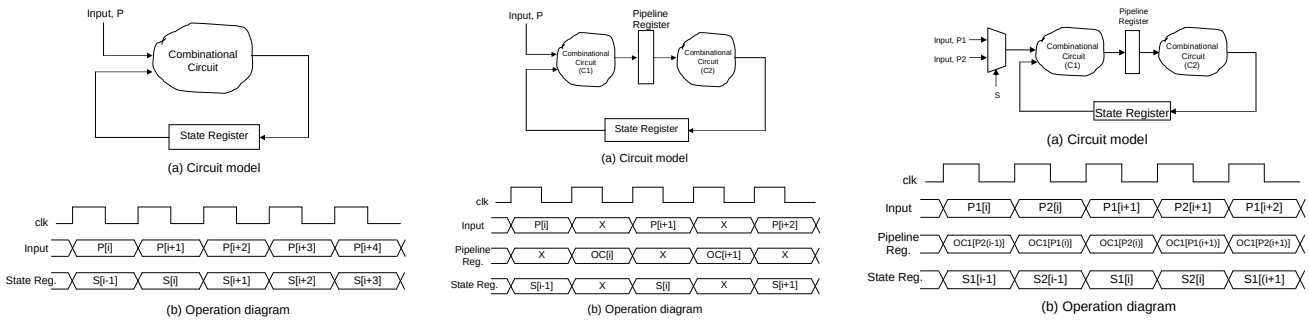
3 Techniques for high-throughput verifier design

3.1 Multi-threading FSM

In conventional FSM-based string matching operations, packets to be examined are fed to the FSM one by one in a serial manner. However, 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*. Techniques to implement multi-threading FSMs are described as follows.

The circuit model of a conventional FSM and its operations are depicted in Figure 2 (I). 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. In general, the maximum clock frequency f_{clk} of the FSM is determined by the longest propagation delay of its combinational circuit. To increase f_{clk}, re-timing techniques can be applied to divide logic propagation paths of the combinational circuit 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 illustrated in the example shown in Figure 2 (II).

Assume that the combinational circuit of the FSM shown in Figure 2 (I) 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 2 (II). 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 $S[i-1]$, which corresponds to FSM input $P[i-1]$. The C_1 output, which is resulted from inputs $P[i-1]$ and $S[i-1]$, is latched into the pipeline register at the end of the ith clock cycle, and is further processed by



(I) Conventional FSM (II) Pipelined FSM (III) Multi-threading FSM

Figure 2. Different FSM implementation and timing diagram.

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 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 is useless for the operation of the FSM (We use "X" to label the useless data in the timing diagram). In network processing domain, there is virtually no dependency between packets, therefore, all packets can be processed in parallel. By taking advantage of this fact, we can feed another packet to the FSM when its combinational circuit c_1 is not used by the first packet. Hence, two packets can be processed by a single FSM. Figure 2 (III) shows the 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 $S_1[i-1]$, which corresponds to FSM input $P_1[i-1]$. 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 $S_2[i-1]$ for Packet P_2 during the ith clock cycle. Consequently, at the $(i+1)$th clock cycle the state register stores state $S_2[i-1]$ 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.

If we ignore the performance penalty caused by the FSM input multiplexer and pipeline stage registers, the throughput of M-threading FSM is M times faster than a conventional FSM (though the latencies of the two are the same). 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 DFFs to support this deep pipeline scheme. This observation is supported by our experimental results to be presented in Section 4.

3.2 High-speed interface circuit design

During the operation of an M-threading FSM, data from M different packets is alternately fed to the FSM. A simple interface circuit for this function is a $M-to-1$ multiplexer. However, if M is large, the $M-to-1$ multiplexer normally has a large delay and potentially become the bottleneck that limits the system performance.

In this paper, we present a fast interface circuit whose schematic is shown in Figure 3. In this design, only Register R_1 communicates with the buffer circuit and FSMs. Thus, it eliminates the use of large multiplexer and de-multiplexer. The operation of the circuit is explained as follows. At the beginning of the operation, the control input S of the $2-to-1$ multiplexer is 0 and, thus, the data from the buffer goes through the multiplexer. During the first clock cycle, Packet P_1 is loaded into R_1 and no data is transferred to the FSMs. During the second clock cycle, the first byte of $P1$ is fed to the FSMs. Meanwhile, Packet P_2 is loaded to R_1 and $P1$ is transferred to Register R_2. Note that a one-bit cyclic right-shifting operation is taking place when moving data from R_1 to R_2. However, no such right-shifting operation is performed when moving packets in the rest of the registers. Following the same manner, packets $P_M, P_{M-1}, \cdots P_1$ are loaded into registers $R_1, R_2, \cdots R_m$ during the first M clock cycles. After that, S switches to 1 and the packets stored in the registers start to circulate along the register chain. During this process, proper bytes from different packets are fed to the FSMs. Again, assume

each packet has N bytes. After $M \cdot N$ clock cycles, S becomes 0 and new packets start to fill the registers. Meanwhile, the packets that have been processed are transferred to an exit queue, which consists of Registers $B_1, B_2, \cdots B_k$. The exit queue introduces the same latency as that caused by the pipeline registers inserted on FSM input and output paths. Hence, the packet is aligned with its matching result generated by the FSMs. According to the matching result, the packet will be routed to the forwarding or discarding hardware.

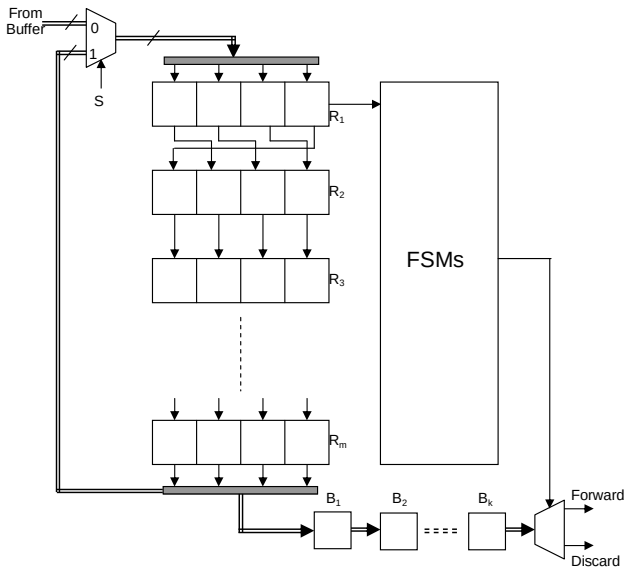


Figure 3. A high-speed interface circuit.

3.3 Minimizing FSM interconnect delay

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.

The use of multiple small FSMs to check if a packet matches a string contained in the class is shown in Figure 4 (a). When a packet is sent from classifiers, it is fed to all the FSMs to check if a match is found. The use of multiple small FSMs, instead of a single large FSM, raises two major design concerns. First, more hardware resources might be needed in the multiple small FSMs (MSF) based approach due to reduced opportunities of hardware sharing. Interestingly, our experimental results show that the hardware implementation cost will not significantly increase in the MSF-based approach. This is partially due to the DFF duplication technique used in modern FPGA synthesis flows. In large FSMs, many nets have large numbers of fan-outs. To minimize the performance degradation caused by the large fan-out nets, FPGA synthesis tools can automatically perform DFF duplication to reduce the fan-out numbers for DFFs in critical paths. Since nets in small FSMs normally have small fan-out numbers, DFF duplication is less frequently used in the MSF-based approach.

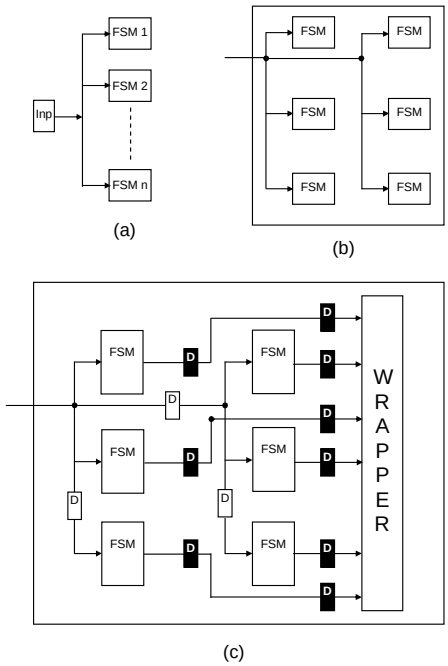


Figure 4. Connecting multiple FSMs.

Another challenge in the MSF-based approach is to minimize the delay of the interconnect that routes the incoming packets to all the FSMs. As shown in Figure 4 (b), the input packet path not only has large fan-out but also travels long distance. To prevent this path from becoming the bottleneck that limits the system performance, two techniques, *pipelined interconnect* and *out-of-order execution*, are used in the design. As shown in Figure 4 (c), pipeline registers are inserted to cut the tree-like input packet routing path into multiple short paths. The throughput of the input interconnect can be improved by increasing the clock frequency. Note that the registers are not evenly added on all of the

FSM input paths. For FSMs that are physically close to the verifier input port, none or few registers are added to their input paths. Meanwhile, for FSMs that are physically far away from the input port, more registers are needed to bring down the input interconnect delay.

Due to the latencies caused by the unbalanced pipelined registers, incoming packets will reach different FSMs at different clock cycles. Thus, the same packet will be examined by different FSMs at different times. We use the term out-of-order execution to refer to this type of operation. The out-of-order execution can be tolerated in string matching applications if the matching results generated by different FSMs at different time can be re-aligned to generate the final matching decision. This can be done by adding registers on FSM output paths. If we use K_{in} and K_{out} to represent the number of registers added to the input and output path of an FSM, the sum of K_{in} and K_{out} should be the same for all the FSMs to achieve proper alignment for the FSM outputs. Note that the registers on the FSM output paths not only align the FSM outputs but also keep the delay of the output path small. Finally, the wrapper circuit will combine all the aligned FSM outputs to generate the matching result.

4 Experimental Results

Experiments have been conducted to study the effectiveness of the proposed techniques. The FPGA hardware used in our study is Xilinx Virtex 4 FX100 device [17]. String matching rules from the Snort [12] 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 netlists of the synthesized FSMs are fed to an in-house re-timing program to add pipeline registers. The modified netlists 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.

Figure 5 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.

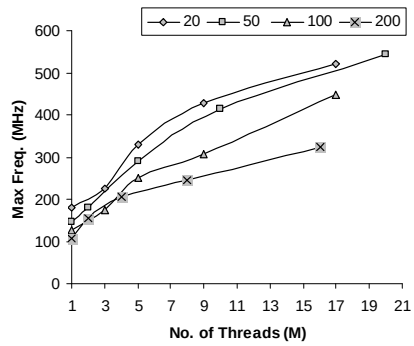


Figure 5. FSM clock frequency versus number of threads.

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 3.1. 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 5 also indicates that the maximum clock frequency that can be achieved by multi-threading FSMs degrades with the sizes of FSMs. To illustrate the cause, interconnect delays on the critical paths of different-sized FSMs are plotted in Figure 6. 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

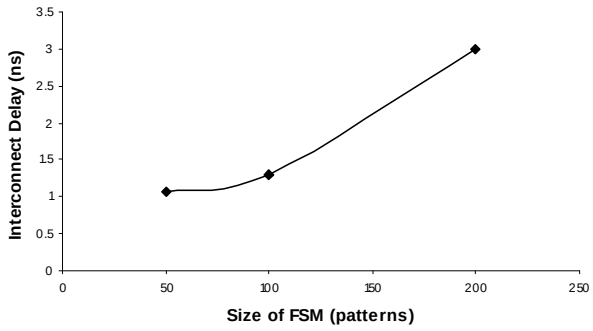


Figure 6. Interconnect delay with different FSM sizes.

FSM design approach is also studied in our experiments. Figure 7 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 increased demand for DFFs in 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.

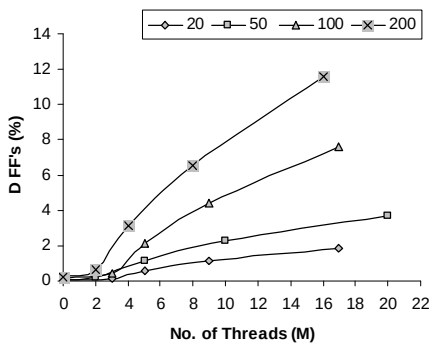


Figure 7. DFF utilization in multi-threading FSMs.

In the proposed system, string matching rules belonging to a class are further divided into subsets and small FSMs are designed for each subset. To study how hardware implementation cost is affected by the MSF-based approach, different FSM sizes are used to design the verifier hardware for a group of 200 string matching rules from the Snort. The design approaches used in the study are: (a) a single FSM with the size of 200, (b) two FSMs of the size 100, and (c) four FSMs of the size 50. The required LUTs and DFFs (in terms of the percentage of the total resources on the Virtex 4 FX100 device) in the three different approaches are compared in Figure 8. The X-axis of the figure indicates the number of FSMs used in the design. Thus, the data points with X-axis coordinates of 1, 2, and 4 correspond to the design approaches a, b and c, respectively. Note that all the FSMs used in this study are multi-threading FSMs with their maximum thread numbers. It shows that the hardware implementation cost will not be significantly increased by using the MSF-based design approach. For this particular case, the amount of LUTs and DFFs are slightly reduced when using four smaller FSMs. This is mainly contributed by the following two factors. First, the FPGA synthesis tool can more efficiently minimize the combinational logic when the circuit size is small. Second, there are less number of nets that have large fan-outs in small circuits than that in large circuits. Thus, DFF duplication techniques are less intensively used in small circuits.

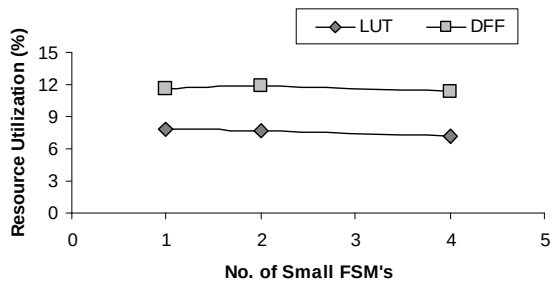


Figure 8. FPGA resource utilization for different FSM design.

The above data also indicates about 12% of DFF and 8% of LUT resources (on the Virtex 4 FX100 device) are needed to implement a verifier that covers 200 string matching rules. There are about 1500 unique strings in the current Snort. (Although a much larger number of strings in the Snort is often referred, that number includes many duplicated rules.) Thus, if we can partition the 1500 unique strings into eight classes and each contains about 200 strings, then all the eight verifiers can be implemented into a single Virtex 4 FX100 device.

Finally, experiments are conducted to show that the delay of the input packet path which connects to multiple FSM inputs will not become the factor limiting the system performance. In the experiments, we assume the input interconnect needs to route the incoming packets from the verifier input port to 10 FSMs. First, the FSMs are synthesized and implemented as modules. At the floor-planning phase, FSM

modules are placed such that some spaces between FSM modules are reserved for routing. After floor planning, an iterative design process is used to manually place interconnect pipeline stages into the reserved routing space. Our experiments indicate that the design goals can be quickly achieved after two or three design iterations. As shown in Figure 9, after four pipeline stages are added to some input path branches, the delay of partitioned interconnect segments can be quickly reduced to less than 2ns, which is small enough to support the FSMs operation at the clock frequency of 500MHz. The numbers of pipelined registers added to the design are also plotted in the figure. It shows that the number of additional DFFs required in the pipelined interconnect design is very small.

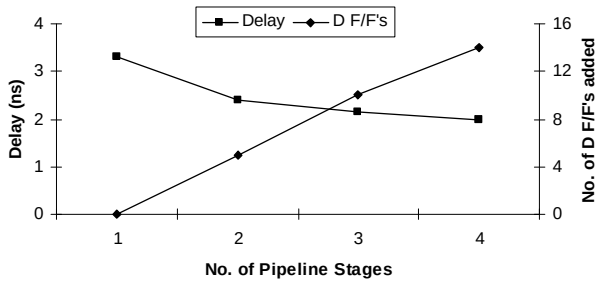


Figure 9. Delay of FSM input path.

5 Concluding Remarks

In this work, techniques, including multi-threading FSM design, high-speed FSM interface circuit, FSM partition and pipelined interconnected are developed to improve the performance of FSM based string matching operation on FPGA platforms. Experimental results demonstrate that the system throughput can be significantly improved by the proposed FSM design. In addition, our experiments show that after applying the multi-threading (deep pipeline) scheme, FPGA interconnect delay starts to dominate the system performance. Partitioning large FSMs into smaller FSM and the use of pipelined interconnects are effective techniques to combat FPGA interconnect delay. Another interesting finding from our experiments is that FSM partitions do not cause significant hardware overhead, which may suggest that smaller FSMs are always preferred on FPGA platforms.

References

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

[2] M. Attig and J. Lockwood. Sift: Snort intrusion filter for tcp. In *Proceedings of 13th Symposium on High Performance Interconnects*, pages 121–127, 2005.

[3] Z. K. Baker and V. K. Prasanna. Time and area efficient pattern matching on fpgas. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 223–232, New York, NY, USA, 2004. ACM Press.

[4] B. Brodie, R. Cytron, and D. Taylor. A scalable architecture for high-throughput regular-expression pattern matching. In *Proc. 33rd International Symposium on Computer Architecture*, 2006.

[5] L. Bu and J. A. Chandy. 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, 2004. IEEE Computer Society.

[6] D. Denning. An intrusion–detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, Feb. 1987.

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

[8] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, Jan. 2004.

[9] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *Trans. on Embedded Computing Sys.*, 3(3):614–633, 2004.

[10] P. Piyachon and Y. Luo. 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, New York, NY, USA, 2006. ACM Press.

[11] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proc. of the 13th Systems Administration Conference*, 1999.

[12] Snort, Inc. *The Open Source Network Instrusion Detection System*, 2004. http://www.snort.org.

[13] H. Song and J. W. Lockwood. 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, 2005. ACM Press.

[14] Y. Sugawara, M. Inaba, , and K. Hiraki. Over 10gbps string matching mechanism for multi-stream packet scanning systems. In *Lecture Notes in Computer Science*, volume 3203, pages 484–493. Springer-Verlag, 2004.

[15] L. Tan and T. Sherwood. Architectures for bit-split string scanning in intrusion detection. *IEEE Micro*, (1):2–9, 2006.

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

[17] Xilinx, Inc. *Virtex-IV Pro and Virtex-IV Pro X Platform FPGAs: Complete Data Sheet*, 2004. http://www.xilinx.com.

[18] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04)*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.