A High-Throughput System Architecture for Deep Packet Filtering in Network Intrusion Prevention*

Dae Y. Kim[1], Sunil Kim[1], Lynn Choi[2], and Hyogon Kim[2]

[1] School of Information and Computer Engineering, Hongik University, 72-1 Sangsu-Dong, Mapo-Gu, Seoul, Korea
{hansol, sikim}@cs.hongik.ac.kr
[2] The Department of Electronics and Computer Engineering, Korea University, Anam-Dong, Sungbuk-Ku, Seoul, Korea
{lchoi, hyogon}@korea.ac.kr

Abstract. Pattern matching is one of critical parts of Network Intrusion Prevention Systems (NIPS). Pattern matching hardware for NIPS should find a matching pattern at wire speed. However, that alone is not good enough. First, pattern matching hardware should be able to generate sufficient pattern match information including the pattern index number and the location of the match found at wire speed. Second, it should support pattern grouping to reduce unnecessary pattern matches. Third, it should show constant worst-case performance even if the number of patterns is increased. Finally it should be able to update patterns in a few minutes or seconds without stopping its operations. We modify Shift-OR hardware accelerator and propose a system architectures to meet the above requirement. Using Xilinx FPGA simulation, we show the new system scaled well to achieve a high speed over 10Gbps and satisfies all of the above requirements.

1 Introduction

The explosive growth of the Internet and the emergence of new applications, such as P2P file sharing, video-on-demand, and e-commerce, dramatically increases network traffic. Network speed and bandwidth is also rapidly increasing to satisfy demand for high-speed Internet access and high bandwidth. These trends have made malicious network attacks, such as Denial of Service (DoS), e-mail virus, and Internet worm, faster and more destructive and damaging. For example, Code Red worm [1] and SQL Slammer worm [2] spread over the world within a few hours and minutes, respectively to cause billions of dollars in damage.

Network Intrusion Prevention Systems (NIPS) have recently emerged as one of the most promising technologies against such network attacks. The NIPS combines both firewall and NIDS [3]. It inspects both packet headers and payloads as a NIDS does and blocks suspicious packets from entering the network as a firewall does. The NIPS lives in-band on the network and processes packets in

* This work was supported by 2005 Korea Sanhank Foundation Research Fund.

W. Grass et al. (Eds.): ARCS 2006, LNCS 3894, pp. 407–421, 2006.

real-time at wire speed. The performance of NIPS is critical because a poorly performing NIPS would be detrimental to the whole network. At the heart of most NIPS is pattern matching to find attack string patterns in the payload. Pattern matching is computationally intensive. The pattern matching routines in Snort [4], a widely used open-source NIDS/NIPS, account for up to 70 % of total execution time and 80 % of instructions executed on real traces [5]. Therefore, a pattern matching method for NIPS should be highly efficient to keep up with ever increasing speed demand.

Pattern matching for NIPS has several domain-specific characteristics [6, 7, 8]. First, the number of patterns is very large and is keep rising. In Snort, a rule describes the pattern of attack signature and an action to take if a packet matches the signature. The number of the patterns is increasing and more than 2100 in the current Snort. Second, the size of patterns varies widely ranging from 1 to 122, although most pattern sizes are below 32. Third, a large number of string patterns are non-case sensitive. More than half of the string patterns used in Snort are non-case sensitive.

A great deal of research has concentrated on developing pattern matching hardware that satisfies all or some of the domain-specific characteristics. However, there are other important features that pattern matching hardware must support to be useful and effective for NIPS.

1. *Pattern match information at wire speed – at least the pattern index number and the location information should be provided:*
   When a pattern match occurs, rule-checking software further examines the packet to check if other rule options are satisfied. Snort uses pattern index information to find the related rule and the location information to decide whether the packet satisfies other content options, such as *depth* and *offset*, which specify how far into a packet should be searched and where to start searching in the packet, respectively. Such information for all matched patterns should be generated at wire speed. Otherwise, pattern matching hardware eventually stalls to process the information.

2. *Pattern grouping support – only patterns related to the rule group a packet belongs to are checked against:*
   In Snort, rules are divided into rule groups by the protocol type and source and destination port numbers specified in the rules. An incoming packet is classified by its protocol type and source and destination ports, and its payload is checked against only those patterns in the corresponding rule group. Without pattern grouping in hardware, there could be many matches against patterns that belong to other unrelated rule groups and this could results in unnecessary software executions.

3. *Worst-case performance:*
   This requirement is also discussed in [6, 9]. The worst-case performance of a NIPS has to match network speed. Otherwise, an attacker can devise a packet with content that results in the worst-case performance of NIPS and

continuously send the packets to render the NIPS unusable, which will eventually block all other legitimate traffic. In addition the worst-case performance should remain constant and predictable even when more patterns are added. If not, it would be hard to predict when the NIPS would fail to meet the speed requirement.

4. *Fast non-interrupting pattern update:*
   This requirement is important to protect networks from a fast spreading Internet worm like SQL Slammer worm. Content-filtering, used in NIPS, should start to filter the new worm within a few minutes after the worm outbreaks in order to successfully quarantine the worm propagation [10]. This implies that pattern matching hardware should be able to update patterns in less than a few minutes or seconds for Internet worm quarantine. Considering the amount of damages caused by latest worm outbreaks, this feature becomes very important. Only the pattern matching architecture in [9] explicitly addressed this issue.

Most pattern matching hardware based on FPGA [11, 12, 13, 14, 15, 16, 17] likely fails to satisfy the worst-case performance requirement. When the number of patterns is increased, the operating frequency of FPGA pattern matching hardware tends to increase due to the increase in the amount of combinational circuits for state transitions. This makes NIPS performance unpredictable and eventually leads to the failure of NIPS performance at some point, not matching network speed. These approaches also most likely fail to meet the fourth requirement, fast non-interrupting pattern update, because they need to resynthesize and reprogram FPGA for new patterns, which usually take a long time for a large number of patterns. Bit-split FSM approach [9] based on Aho-Corasick algorithm [18] uses SRAM for state transition tables. The approach shows excellent performance and hardware area utilization as well as satisfies two requirements, the worst-case performance and fast non-interrupting pattern update.

In this paper, we propose a pattern matching system architecture for the wire-speed pattern match information and pattern grouping requirements. To the authors' best knowledge, this is the first work that successfully addresses these two requirements. Two papers [16, 17] addressed issues related to the pattern match information requirement and proposed two similar architectures that generate signature indexes using pruned priority binary tree and highly pipelined binary-OR tree. However, these architectures cannot handle multiple matches that simultaneously occur and also do not provide any information on the location of a match in a payload. Our study on string patterns used in Snort shows that there are many suffix matches of patterns. The maximum number of patterns in the same suffix match group is 5. This implies that there could be up to 5 multiple matches for a given input character.

In this paper we also introduce some improvements to Shift-OR pattern matching accelerator [8] for fast non-interrupting pattern update. The Shift-OR pattern matching accelerator uses SRAM as Bit-split FSM does and does

not need to change hardware when a pattern is added. This allows the accelerator to have constant worst-case performance. The proposed pattern matching system architecture with the updated Shift-OR pattern matching accelerator successfully satisfies all of the above four requirements.

We evaluate our proposed architecture using Xilinx FPGA tools. We design the proposed pattern matching system and obtain timing and area results through FPGA simulation. This paper begins by briefly describing Shift-OR algorithm [19] in Section 2. Section 3 presents the pattern matching system that consists of the updated Shift-OR pattern matching accelerator, pattern grouping hardware, and pattern match information system. We evaluate the proposed architecture by Xilinx FPGA tools in Section 4 and conclude in Section 5.

2 Shift-OR Pattern Matching Algorithm

In this section, we briefly describe Shift-OR pattern matching algorithm for a single pattern, which is the basis of the pattern matching architecture we present in this paper. The algorithm uses bitwise techniques. It keeps a bit array of size m (pattern length), a state vector R that shows if prefixes of the pattern match at the current place. For example, there are a pattern $P = p_0 \dots p_{m-1}$ and input string $X = \dots x_{i+j} \dots$. After processing x_{i+j}, $R[j] = 0$ if $x_i \dots x_{i+j}$ matches $p_0 \dots p_j$, otherwise $R[j] = 1$. There is another bit array of size m, a character position vector S_c, denoting the position of character c in pattern P. For example, $S_c[i] = 0$ if $p_i = c$, otherwise $S_c[i] = 1$. If we know that the bit value of $R[j]$ after processing x_{i+j}, we can easily compute $R[j+1]$ by knowing whether the next character x_{i+j+1} appear at pattern position p_{j+1}. $R[j+1]$ can be defined as follows:

$$R[j+1] = \begin{cases} 0 \text{ if } R[j] = 0 \text{ and } S_c[j+1] = 0 \text{ where } c = x_{i+j+1} \\ \\ 1 \text{ otherwise.} \end{cases} \quad (1)$$

$$R[0] = S_c[0] \text{ where } c = x_{i+j+1} \quad (2)$$

$R[m-1] = 0$ means the pattern $x_i \dots x_{i+m-1}$ matches $p_0 \dots p_{m-1}$, that is, the matching pattern is found. The computation of new R for the next input character c reduces to Shift and OR operations ($SHIFT(R)$ OR S_c).

This algorithm easily handles any finite class of symbols, complement symbols and even don't care symbols. If position i of a pattern allows a class of symbols $\{x, y, z\}$, then letting $S_x[i] = S_y[i] = S_z[i] = 0$ handles the case. Complement symbols and don't care symbols can be handled in the same way. Therefore, noncase-sensitive matches can be easily processed without any additional overhead. The algorithm can be extended for multiple patterns. It first concatenates all state vector R for each pattern into one large state vector. It also concatenates all character position vector S_cs for each pattern into one large character position vector for a given character c. The only difference from single pattern match is that when the new bit value of the large R corresponding to the first

position of a pattern, i, is computed, the value is only affected by the large $S_c[i]$, not by shifted value from $i-1^{th}$ position. In the remaining of the paper, '*vector*' denotes a concatenated vector.

3 Pattern Matching System

The pattern matching system presented in this paper has multiple pattern matching units (PMU). PMU is the updated Shift-OR pattern matching accelerator that can do fast non-interrupting pattern update and have constant worst-case performance. The pattern matching system provides pattern grouping and can generate pattern matching information at wire speed.

3.1 Pattern Match Unit

The pattern match unit performs Shift-OR pattern match algorithm for multi-patterns. Figure 1 shows the components of the pattern matching unit (PMU). PMU has a multiport SRAM that stores 256 character position vectors, one per an 8-bit character. Multiple characters are read from the payload and used to address the multiport SRAM. The size of the SRAM is 256 x W bits, where W is the width of the SRAM. It is also the size of the character position vector that the PMU uses. The character position vector is a concatenated character position vector for a character for all the patterns assigned to the PMU. These vectors are precomputed from string patterns and loaded into the SRAM. The number of SRAM ports, N, determines the number of input characters processed together. PMU has four registers for bit vectors: pattern boundary vector (B),

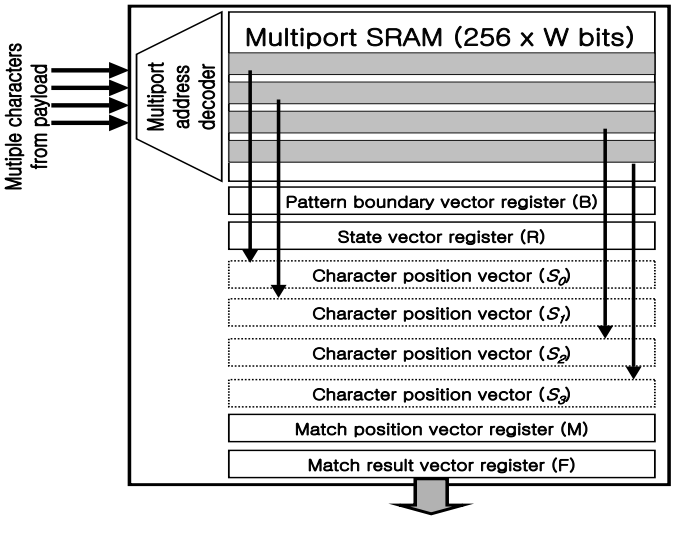

Fig. 1. Pattern Matching Unit

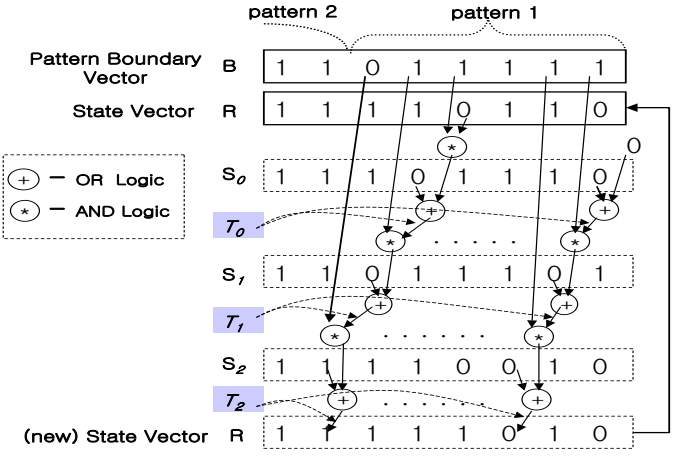

Fig. 2. Shift-OR Computation

state vector (R), match position (M), and match result vector (F). The size of all vector registers is the same, W bit wide.

Once N character position vectors $S_0, \ldots, S_{N-1}$ are read from the multiport SRAM, Shift-OR computation is performed using state vector R generated from the previous cycle and pattern boundary vector B to compute a new state vector R as shown in Figure 2. The pattern boundary vector B denotes boundaries of each pattern by bit value '0' and is used to prevent the Shift-OR computation result from propagating cross pattern boundaries. Initially R is *AND*ed with B, then shifted and *OR*ed with S_0 to generate intermediate state vector T_0. Next, T_0 is *AND*ed with B, then shifted and *OR*ed with S_1 to generate next intermediate state vector T_1. The same computation is performed at each stage until T_{N-1} is generated. The final result T_{N-1} will be stored into R register again for the next cycle computation. The computation is represented in the following equations.

$$T_k(0) = S_k(0) + 0 = S_k(0) \qquad \text{for all } k \tag{3}$$

$$T_0(i) = S_0(i) + (R(i-1) * B(i-1)) \qquad \text{for } i > 0 \tag{4}$$

$$T_k(i) = S_k(i) + (T_{k-1}(i-1) * B(i-1)) \qquad \text{for } k > 0,\ i > 0 \tag{5}$$

As shown in Figure 2, the shift operations are performed by simply connecting the i^{th} position results to one input port of the OR gate of the $i+1^{th}$ position at the next stage. Each stage computation is equivalent to one Shift-OR operation in Shift-OR algorithm. N Shift-OR operations are performed in a single cycle. Combinatorial logic circuit is used for all the computation, and intermediate state vectors, $T_0, \ldots, T_{N-1}$, are generated on the fly and do not need to be stored. The character position vectors, $S_0, \ldots, S_{N-1}$ also do not need to be stored. They are the output of the multiport SRAM and directly used for the computations.

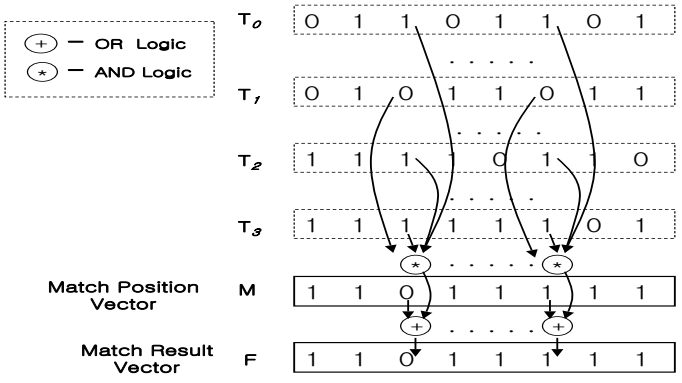

Fig. 3. Match Result Vector Generation

While N input characters are simultaneously processed, several matches can be found. Figure 3 shows the architecture that detects all the matches from all the intermediate state vectors. All the bits at the same position in the intermediate state vectors are *AND*ed and then *OR*ed with the bit at the same position in match position vector M. Match position vector has 0 bit at a pattern's end position. The result is match result vector and stored in F register. If the match result vector has zero bits (*match bits*), it means there are matches. The output of F vector register becomes the output of PMU.

PMU can perform fast non-interrupting pattern update. A pattern can be easily ignored by resetting M vector bit at the pattern's end position to 1. The effect is the same as deleting the pattern. Adding a pattern requires reinitializing pattern boundary vector B and match position vector M registers and reloading the multiport SRAM. Updating and initializing B and M registers can be done by writing the vector values into SRAM, reading the vector data from the SRAM, and finally loading them into the corresponding registers. For this, we can use a separate SRAM or the same multiport SRAM by increasing its depth by 2 for the vectors. The time for reloading the multiport SRAM for new character position vectors takes the same number of cycles as the length of the new pattern. We need to write only the character position vectors for characters in the new pattern. Writing multiport SRAM can also be performed without blocking any read operations. Therefore, pattern updates can be done without stopping the system.

3.2 Pattern Group Unit

In hardware pattern matching where all patterns are searched together, the time taken for the pattern match process itself is not affected by grouping patterns. However, pattern grouping can reduce many unnecessary matches against patterns that belong to other unrelated rule groups. Rule groups are classified by the protocol type and source and destination ports specified in the rules. Patterns in the same rule group form a pattern group. The patterns are assigned to PMUs

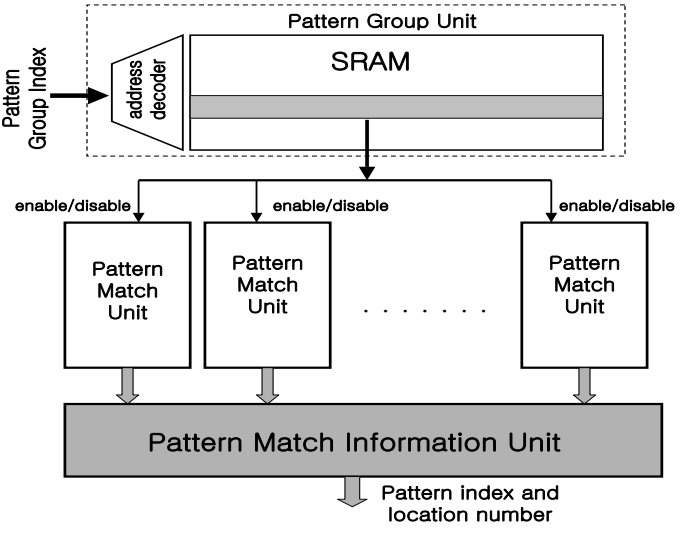

Fig. 4. Pattern Matching System

such that a PMU processes only patterns in the same pattern group. A pattern group index is obtained by inspecting an incoming packet's protocol type and source and destination ports and used to access Pattern Group Unit (PGU) as shown in Figure 4. The PGU enables only those PMUs that handle the corresponding pattern group. A PMU activation vector is read from the SRAM accessed by pattern group index, and each bit of the vector is used to enable or disable a PMU. When a PMU is disabled, the match result vector of the PMU generates all 1's, effectively having no effect on the match result.

3.3 Pattern Match Information Unit

Pattern matching hardware usually raises a signal line when a pattern matches. For Snort, pattern matching hardware may need almost 2000 signal lines and more. The software cannot read all of the signal lines at once. Therefore the signal line index (or *pattern index*) should be provided to rule checking software for further examination. Pattern Match Information Unit (PMIU) reads the match result vectors of all PMUs and generates pattern indexes and the location information of the matching patterns in the payload for all match bits. PMIU speed should match the processing speed of core pattern match hardware. Otherwise, the pattern match hardware would stall at some point to wait for all the match results to be processed.

Example architecture of PMIU is shown in Figure 5. It is a pipelined priority tree with special functionalities. The figure shows how PMIU receives a match result vector from F register and generates a position index of the match bit at the 35^{th} bit position. The position index can be used as a pattern index. First, an Input Vector Encoder (IVE) receives each 4 bits in the match result vector

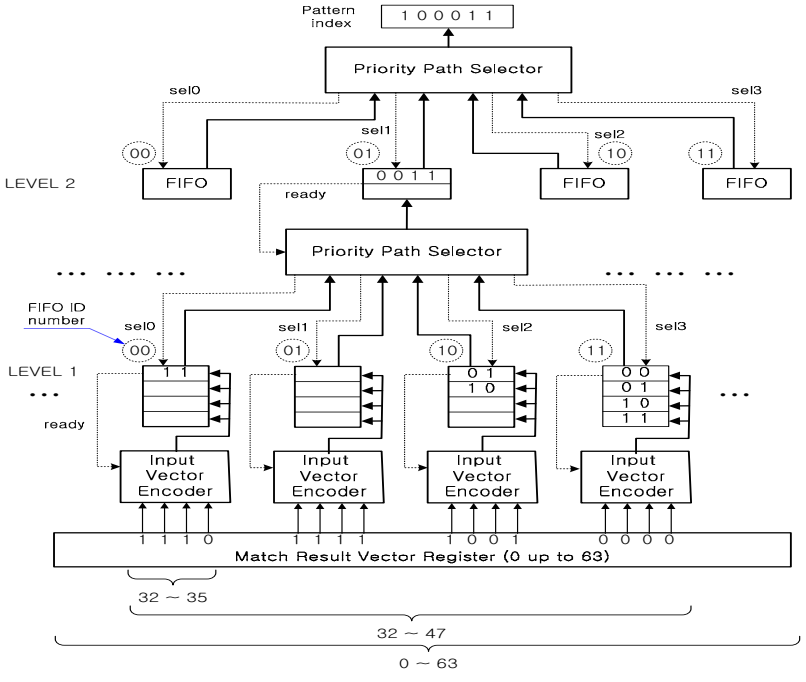

Fig. 5. Example of Pattern Match Information Unit

and encodes the relative position indexes of all match bits in those 4 bits. The encoded values are put into the first-level FIFO. The first-level FIFO with ID '00' has the relative position index of the match bit at the 35^{th} bit position, and the index value is '11'. The next level Priority Path Selector (PPS) selects the first-level FIFO '00' out of the four first-level FIFOs. The four FIFOs have the relative position indexes of match bits in each 4 bits at the bit position 32^{nd} to 35^{th}, 36^{th} to 39^{th}, 40^{th} to 43^{rd} and 44^{th} to 47^{th}, respectively. The PPS uses the relative position index ('11') from the first FIFO and the FIFO ID ('00') to generate position index '0011'. The position index is stored in the second level FIFO '01'. Note that the value '0011' is the relative position index of the match bit at 35^{th} bit position in 16 bits, from 32^{nd} to 47^{th} position. The final PPS selects the position index ('0011') from the second-level FIFOs and uses the FIFO ID number '01' to generate the final position index '100011'. In the next, we describe three main components of the PMIU.

FIFO. The FIFO stores the position index of matching bits. A FIFO entry consists of three control bits, tag (T), dummy (D), empty (E), and index bits (I). In Figure 5, only the index bits are shown for simplicity. The index bits store the relative position index of a match bit. At the first level, the size of the index bits is $\log_2 N$ where N is the number of bits processed by an IVE. As the level goes up, the size of the index bits is increased by $\log_2$ of the number of the lower level FIFOs connected to a PPS.

An empty control bit (E) shows whether the corresponding FIFO entry is empty or not. A tag bit (T) is used to represent the sequence of match results. A tag bit is associated with a match result vector register, and the value is toggled between 0 and 1 every time a new result vector is loaded into the register. All PMUs generate new result vectors at the same time. The tag bit value is moved along with the index bits for the match result vector. We can compute the location information of the pattern by counting the number of the tag bit changes when the final pattern index is generated. Using the tag bit,PPS selects input index bits such that the index bits for earlier match results vector are moved to the upper levels before the index bits for any subsequent match result vectors. The dummy bit tells whether the index bits have a real position index. An IVE fills one FIFO entry with dummy index bits when there is no match bit. This prevents index bits for different match result vectors, but with the same tag value from appearing as inputs of a PPS at the same time. Dummy index bits are eliminated by PPS so that there is one dummy index left at the final stage of PMIU for each match result vector with no matches.

The FIFOs at each level create a pipeline stage. The first-level FIFO connected to IVE is different from FIFOs at the other levels. The size of an IVE input bits determines the depth of the first-level FIFO. All the relative position indexes of match bits are generated and loaded into the FIFO in one cycle. The depth of FIFOs at the other level should be at least two. Two entries are required not to make a bubble in the pipeline stage. With two-entry FIFO, a new relative position index can be generated and stored into the FIFO by the lower-level PPS connected as soon as there is at least one empty entry. This prevents a bubble from being introduced in the pipeline.

Priority Path Selector. A Priority Path Selector (PPS) selects the highest priority index bits from all the connected input FIFOs and generates a new relative position index from the input index bits and the ID number of the FIFO selected. The selected FIFO entry is erased, and the new relative position index is stored in the output FIFO connected to the PPS. The priority selection should consider tag and dummy bit values as well as the priority of input FIFOs. The operation of a PPS is executed in one cycle.

The priority mode of a PPS changes between 1 and 0-mode. In a given priority mode, input FIFO entries with the same tag value as the mode are considered for selection and subsequent operations. The mode changes only when the first entries of all input FIFOs have the same tag value, and then the priority mode is changed to the tag value. This is to compute all the relative position indexes in the same order of the match result vector generation. When a PPS finds at least one non-dummy entry, then it selects one entry among them and erases all dummy entries by sending a select signal to all related input FIFOs. If all entries are dummy entry, only one dummy entry is move to the upper level, and all the other dummy entries are erased as well. By doing this, at most only one dummy entry is left when it reaches the top-level PPS. The dummy entry is for a match result vector with no matches.

Input Vector Encoder. An Input Vector Encoder (IVE) receives a part of a match result vector and generates all the relative position indexes of match bits in the partial match result vector. If there is no match bit, the IVE generates only one dummy index. The IVE stores all the relative position indexes into the first-level FIFO in one cycle and set its mode to the same as the tag value of the match result vector. As far as the mode of an IVE is the same as that of the match result vector, no further operation is performed. When the tag of the match result vector changes, that is, a new match result vector is generated for the new input payload byte and loaded into the vector, the IVE generates the relative position indexes again.

4 Evaluation

We design PMU using the latest Xilinx FPGA Virtex-4 [20]. Virtex-4 has the largest number of embedded RAMs, called block RAMs, which are used to construct multiport SRAM in our design. Unfortunately it has only dual port memory configuration. Therefore, we have to duplicate memory banks to simulate the multiport SRAM when the number of ports becomes larger than 2. In this experiment, we choose 256 bits for the size of the character position vector. Therefore a PMU can handle the total pattern length of 256 bytes. The average size of patterns in Snort is about 12 bytes and hence a PMU can process more than 20 patterns in average.

Figure 6 shows the processing speed of a PMU as the number of Shift-OR stage is increased. The number of Shift-OR stages is shown in logarithmic scale. We can process the same number of input characters together as the number of Shift-OR stages. There are two different versions of the designs. The one labeled as *'pipeline'* has a pipeline stage between the memory bank and Shift-OR computation circuits, and the other one labeled as *'non-pipeline'* does not have a pipeline stage. The graph shows that the processing speeds of both versions of a PMU can reach up to 14 and 14.5 Gbps, respectively, with 64 Shift-OR stages.

The pipelined design shows up to 58 % improvement over the non-pipelined design. The performance difference is more noticeable when the number of Shift-OR stage is small. This is because the effective memory access time is reduced by overlapping memory accesses and a few stages of Shift-OR computations. This shows that memory access time is critical when the number of Shift-OR stages is small. As the number of Shift-OR stages is increased, the total Shift-OR computation time for all stages becomes the dominant performance factor.

Figure 7 shows the resource count for different hardware resources, such as RAM banks, LUT, flip-flops for the non-pipelined design and flip-flop for the pipelined design. They are labeled as *RAM, LUT, FF(NP), FF(P),* respectively in the figure. The resource count counts the number of resources needed per one pattern character. The increase in the number of RAM banks is due to the simulation of multiport memory. As the number of Shift-OR stages is increased, we need more memory read ports. Duplicated memory banks are used instead of memory ports, and this leads to the large number of memory banks for a

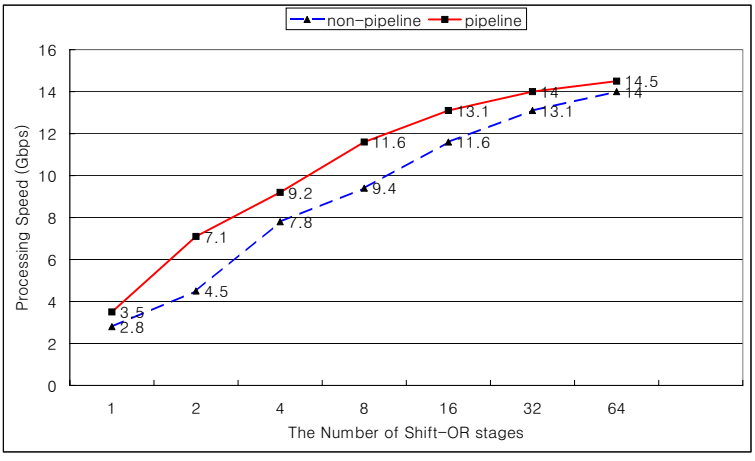

Fig. 6. PMU Processing Speed vs. The Number of Shift-OR Stages

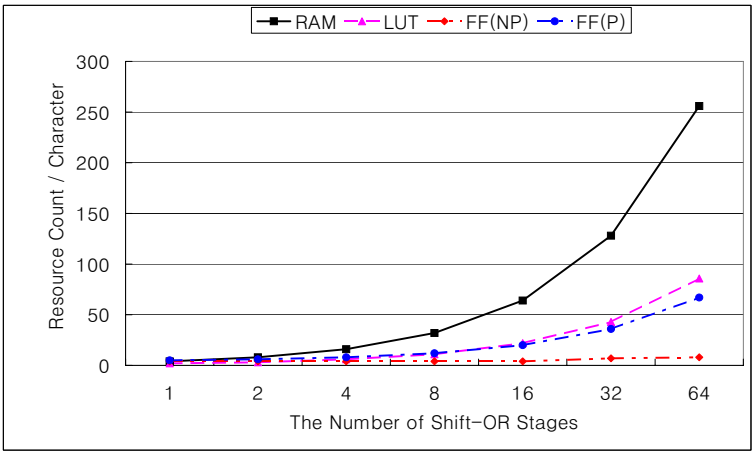

Fig. 7. Resource Count per Pattern Character vs. The Number of Shift-OR Stages

large number of Shift-OR stages. The number of LUT and flip-flops for the pipelined design is increased as well. This is because Shift-OR stages use LUTs for the logic, and the number of flip-flops for memory pipeline is increased due to memory port increase for more Shift-OR stages. Note that the number of flip-flops for the non-pipeline design is almost constant. This implies that non-pipeline is a good choice when memory is fast enough and many Shift-OR stages are needed.

Overall, there are trade-offs between the processing speed and the amount of hardware resources available for Shift-OR pattern matching architecture. As we add more Shift-OR stages, we need more read ports or memory banks and LUTs even for the resource-efficient non-pipeline design. However, note that

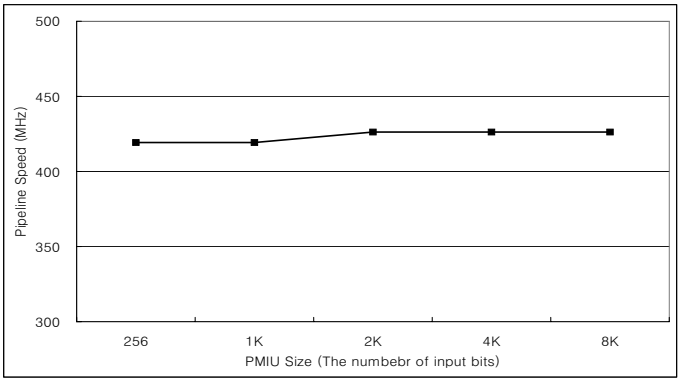

Fig. 8. PMIU Pipeline Speed vs. PMIU sizes

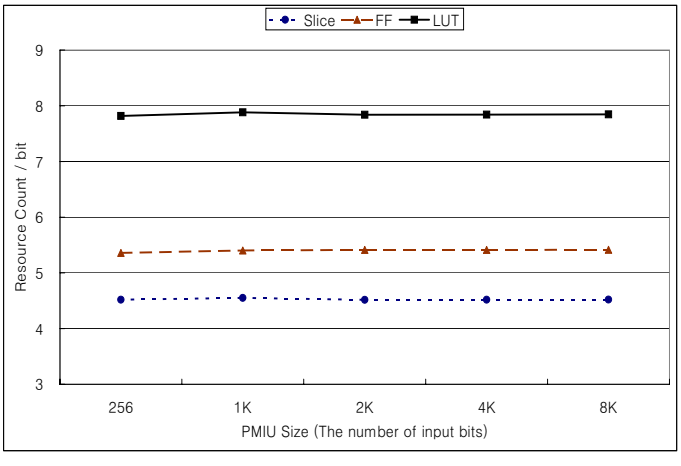

Fig. 9. Resource counter per one PMIU bit vs. PMIU sizes

when more hardware resource is available, it is relatively easy to improve the processing speed of a PMU by adding more Shift-OR stages. This is an important advantage of Shift-OR pattern matching architecture over other previous pattern matching hardware.

We implement PMIUs with 256 inputs through with 8K inputs. Figure 8 shows the pipeline speed of PMIU as its input size increases. The pipeline speed remains almost constant for all PMIU sizes we test and successfully matches the speed of all the non-pipelined PMUs and the speed of pipelined PMU except those with 1 or 2 Shift-OR stages. Figure 9 shows the resource count for different hardware resources, slices, LUTs, flip-flops for different sizes of PMIUs. The resource count counts the number of resources per one input bit of a PMIU. The hardware resource used to construct a PMIU increases linearly as the size

increases. These data show that the PMIU architecture uses a reasonable amount of hardware resources and can produce a pattern index and the location information at least every clock cycle of 420 MHz.

Finally we put all system components together on one single Xilinx Virtex-4 chip and measure the resource utilization. The designed architecture has one PGU of size 512 x 32 bit wide, 32 PMUs with 4 Shift-OR stages, one 8K-bit PMIU. The architecture can support total pattern length of 8K bytes. The slice is the most constraint resource in the design, reaching 94 % utilization. A slice consists of two LUTs and flip-flops. There are many flip-flops and LUTs left unused (54 % and 88 % utilization, respectively). This implies that a custom design may improve the balance of the resource utilization. However, we do not further investigate the issue in this paper.

5 Conclusion

In Network Intrusion Prevention Systems (NIPS), pattern matching is extensively used to find attack signatures in a payload and is the most computationally intensive part of the execution. In this paper, we proposed a pattern matching system architecture that satisfies four important requirements for NIPS: pattern match information generation at wire speed, pattern grouping support, constant worst-case performance, and fast non-interrupting pattern update. These requirements are as important as finding matching attack patterns at wire speed for NIPS.

We evaluated the proposed architecture using Xilinx FPGA tools and showed that the system scaled well to achieve a high speed over 10Gbps. The pipeline speed of PMIU matched most of PMU operation speeds and could generate a pattern index and match location information at every clock cycle of 420 MHz.

References

1. Code Red worm exploiting buffer overflow in IIS indexing service DLL. CERT Advisory CA-2001-19 (2002)
2. MS-SQL Server Worm. CERT Advisory CA-2003-04 (2003)
3. Xinyou Zhang, Chengzhong Li, Wenbin Zheng: Intrusion Prevention System Design. In: The Fourth International Conference on Computer and Information Technology (CIT'04). (2004)
4. Snort. (http://www.snort.org)
5. S. Antonatos, K. G. Anagnostakis, E. P. Markatos: Generating Realistic Workloads for Network Intrusion Detection Systems. In: Proceedings of ACM Workshop on Software and Performance. (2004)
6. N. Tuck, T. Sherwood, B. Calder, G. Varghese: Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In: Proceedings of the 23rd Conference of the IEEE Communication Society (INFOCOM04). (2004)
7. R. Liu, N. Huang, C. Chen, C. Kao: A Fast String Matching Algorithm for Network Processor Based Intrusion Detection System. ACM Transaction on Embedded Computing Systems **3** (2004) 614–633

8. Sunil Kim: Pattern Matching Acceleration for Network Intrusion Detection Systems. In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS V). (2005)
9. Lin Tan, Timothy Sherwood: A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In: The 32nd Annual International Symposium on Computer Architecture. (2005)
10. David Moore, Colleen Shannon, Geoffrey M. Voelker, Stefan Savage: Internet Quarantine: Requirements for Containing Self-Propagating Code. In: IEEE INFOCOM. (2003)
11. Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, John Lockwood: Deep Packet Inspection Using Parallel Bloom Filters. In: Proceedings of the Symposium on High Performance Interconnects (HotI). (2003) 44–51
12. B. L. Hutchings, R. Franklin, D. Carver: Assisting Network Intrusion Detection with Reconfigurable Hardware. In: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. (2002)
13. J. Moscola, J. Lockwood, R. P. Loui, M. Pachos: Implementation of a Content-Scanning Module for an Internet Firewall. In: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. (2003)
14. M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, V. Hogsett: Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. In: Proceedings of International Conference on Field-Programmable Logic and Applications. (2002)
15. I. Sourdis, D. Pnevmatikatos: Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In: Proceeding of the 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines. (2004)
16. Young H. Cho, W. H. Mangione-Smith: Programmable Hardware for Deep Packet Filtering on a Large Signature Set. In: Workshop on Architectural Support for Security and Anti-Virus. (2004)
17. Young H. Cho, W. H. Mangione-Smith: Deep Packet Filter with Dedicated Logic and Read Only Memories. In: Proceedings of the 12th IEEE Symposium of Field-Programmable Custom Computing Machines. (2004)
18. A. V. Aho, M. J. Corasick: Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM **18** (1975) 333–340
19. R. A. Baeza-Yates, G. H. Gonnet: A New Approach to Text Searching. In: Proceedings of ACM 12th International Conference on Research and Development in Information Retrieval. (1989)
20. Xilinx, Inc. (http://www.xilinx.com)