IEICE TRANS. COMMUN., VOL.E93–B, NO.9 SEPTEMBER 2010

LETTER

A Hardware-Efficient Pattern Matching Architecture Using Process Element Tree for Deep Packet Inspection

Seongyong AHN[†], **Hyejeong HONG**[†], **HyunJin KIM**[†], **Jin-Ho AHN**[††], **Dongmyong BAEK**[†††], ***Nonmembers***, ***and*** **Sungho KANG**[†a)], ***Member***

SUMMARY This paper proposes a new pattern matching architecture with multi-character processing for deep packet inspection. The proposed pattern matching architecture detects the start point of pattern matching from multi-character input using input text alignment. By eliminating duplicate hardware components using process element tree, hardware cost is greatly reduced in the proposed pattern matching architecture.

key words: *network intrusion detection system, deep packet inspection, pattern matching, brute-force algorithm*

1. Introduction

Due to the growth of network environment complexity and malicious network attacks, efficient and effective implementation of DPI (Deep Packet Inspection) is important. Pattern matching that searches pre-defined patterns for packet payloads determines the throughput of DPI. Because software-based pattern matchings cannot support the network wire speed, hardware-based pattern matching architectures have been researched.

Chang et al. [1] proposes a hardware-based multi-character pattern matching architecture for high throughput. Because of multi-character processing of input text per step, throughput of the previous architecture is higher than an architecture which can process one character per step. However, hardware cost is greatly high because comparators are required to search multiple characters of input text for all possible matchings.

Cho et al. [2] proposes a hardware-optimized architecture for multi-character pattern matching. Using this architecture, duplicate comparators for common substrings are removed. Although hardware cost is reduced, hardware cost is still high owing to comparators for all possible matchings in multiple characters.

The architecture in [3] reduces hardware cost using input text alignment. Because input text alignment aligns all possible matchings, comparators for only one possible matching are required. Therefore, hardware cost is reduced in comparison with the previous architectures [1], [2]. Nevertheless, duplicate hardware components still exist.

This paper proposes a pattern matching architecture that adopts process element tree. By organizing process element tree, the number of required comparators is reduced because duplicate hardware components are shared. The proposed pattern matching architecture reduces hardware cost using process element tree and input text alignment and processes multi-character per step.

2. Input Text Alignment

In multi-character processing, pattern matching architecture should find not only full matching, such as (*case*1) in Fig. 1(a), but also partial matchings, such as (*case*2) and (*case*3) in Fig. 1(a). Therefore, when process width, the number of input characters per one step, increases, hardware cost increases to cover all of the possible matchings.

Therefore, comparators are required to find both full matching and partial matching. To reduce the required number of comparators, input text alignment is proposed in the previous work [3]. By applying input text alignment, partial matchings in the input text are found and aligned to full matching.

When the first substring of a pattern is $s_1=\{c_1,c_2,\cdots,c_n\}$ with process width n, the substring is searched for in the input text $T=\{t_1,t_2,\cdots,t_x,x\rightarrow\ \infty\}$. c_x and t_x represent a character. The condition of full matching is $c_1=t_1,c_2=t_2,\cdots,c_n=t_n$. Partial matching occurs when one of the input texts $\{t_2,t_3,\cdots,t_{n+1}\}$,$\{t_3,t_4,\cdots,t_{n+2}\}$, $\cdots$,$\{t_n,t_{n+1},\cdots,t_{2n-1}\}$ is equal to substring s_1. In the case of $\{t_2,t_3,\cdots,t_{n+1}\}$, the input text T is aligned to $T'=\{t_2,t_3,\cdots,t_x,x\rightarrow\infty\}$ by one character shift. Similarly, the input text that has a partial matching is aligned to full matching by two $\sim n-1$ character shifts.

In other words, if the suffix of the input text is matched to the prefix of the pattern, input text alignment aligns the

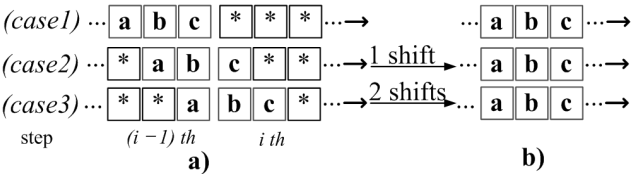

Fig. 1 All of the possible matchings and input text alignment for substring '*abc*.'

Manuscript received January 5, 2010.
Manuscript revised April 28, 2010.

[†]The authors are with the School of Electrical & Electronic Eng., Yonsei University, Seoul, Korea.
[††]The author is with the Dep. of Electronic Eng., Hoseo University, Asan, Korea.
[†††]The author is with the Next Generation Ethernet Research Team, ETRI, Deajon, Korea.
a) E-mail: shkang@yonsei.ac.kr

DOI: 10.1587/transcom.E93.B.2440

input text. Consider the example in Fig. 1.

• (case 1): The input text is exactly matched to substring '*abc*.' In the next step, the next input text is compared to the next substring.
• (case 2): The suffix of the input text '*ab*' is matched to the prefix of substring '*ab*.' Because the matching is determined by the matching results of the next character of input text, a shift of one character occurs
• (case 3): The suffix of the input text '*a*' is matched to the prefix of substring '*a*.' Because the matching is determined by the matching results of the next two characters of input text, a shift of two character occurs.

3. Proposed Pattern Matching Architecture

The proposed pattern matching architecture based on the previous work [3] consists of *process elements*(PEs) and *alignment elements*(AEs). To implement a pattern, $\lceil \frac{\textit{the pattern length}}{\textit{process width}} \rceil$, chained PEs and one AE are needed. When process width is n, a PE has n comparators and an AE has $\frac{n(n-1)}{2}$ comparators. When substring in the first PE is s_1={c_1,c_2,$\cdots$,c_n} and process width is n, the AE consists of the sets of the comparators for {c_2,c_3,$\cdots$,c_n},{c_3,$\cdots$,c_n},$\cdots$,{c_n} in order to handle partial matching.

In the previous pattern matching architecture [3], PE chains are implemented independently of each pattern. Independent PE chains take advantage of the management of the patterns, but the AE is needed to implement each PE chain. Also, independent PE chains have duplicate PEs that have identical substrings. Therefore, if duplicate AEs and PEs are shared, hardware cost decreases more than the previous pattern matching architecture with independent PE chains.

When substring-prefix is shared, duplicate AEs and PEs can be shared. Let us suppose that two patterns in a set of patterns P, p_1={s_1,s_2,$\cdots$,s_i} and p_2={s'_1,s'_2,$\cdots$,s'_j}, are implemented by two PE chains, E_1={e_1,e_2,$\cdots$,e_i} and E_2={e'_1,e'_2,$\cdots$,e'_j}. s_x represents a substring which has n characters and e_x represents a PE. And when process width is n, s_1={c_1,c_2,$\cdots$,c_n} and s'_1={c'_1,c'_2,$\cdots$,c'_n} are implemented by e_1 and e'_1. If s_1 and s'_1 are identical, which means c_1=c'_1, c_2=c'_2, $\cdots$, c_n=c'_n, e_1 and e'_1 can be shared. AE for s_1 and s'_1 can also be shared. When s_{k-1} and s'_{k-1}($1 < k \leq \min(i, j)$) are identical and, if s_k and s'_k are identical, e_k and e'_k also can be shared. By eliminating PEs and AE for the substring-prefix, process element tree is organized.

The example of the proposed pattern matching architecture for the set of patterns P={*expection*, *expectant*, *expand*}, when process width is 3, is Fig. 2(b). Figure 2(a) is the previous pattern matching architecture that does not adopt process element tree. Each pattern splits into multiple substrings which have the length of process width, p_1={$exp(s_1)$, $and(s_2)$}, p_2={$exp(s'_1)$, $ect(s'_2)$, $ati(s'_3)$, $on(s'_4)$}, p_3={$exp(s''_1)$, $ect(s''_2)$, $ant(s''_3)$}, and is implemented by PE chains and AEs. In Fig. 2(a), s_1, s'_1 and s''_1 are identical. Also, three AEs handle partial matching of the substring '*exp*.' Therefore, duplicate PEs and AEs can be shared. Because s'_1 and s''_1 are identical and s'_2 and s''_2 are also identical, duplicate PEs for the substring '*ect*' can be shared. In Fig. 2(b), by sharing PEs and AEs for substring '*exp*' and '*ect*,' process element tree is organized. The previous pattern matching architecture in Fig. 2(a) requires nine PEs and three AEs. Whereas, the proposed pattern matching architecture requires six PEs and one AE. Using process element tree, three PEs and two AEs are eliminated.

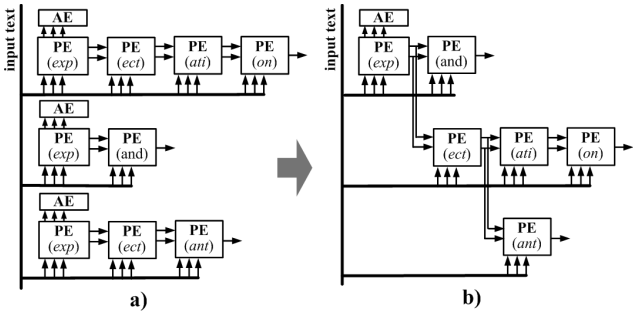

Fig. 2 The process element tree for the set of patterns P = {*expection*, *expectant*, *expand*}, when process width is 3.

When duplicate PEs for s_k=s'_k are shared, if s_{k-1} and s'_{k-1} are not identical, it causes wrong pattern matching results. For example, s_2 and s'_2 in p_1={s_1, s_2, s_3}, p_2={s'_1, s'_2, s'_3} are identical and shared. When s_1 and s'_1 are not identical, the architecture causes wrong pattern matching results, such as $wrong\_pattern_1$={s_1, s'_2, s_3} and $wrong\_pattern_2$={s'_1, s_2, s'_3}. Therefore, only substring-prefix can be shared.

4. Experimental Results

In our experiments, we compute the hardware cost of the pattern matching architecture using the number of comparators which are major components of the PE and the AE. Experiments use four rule sets of *Snort* version 2.6 [4]. The number of AEs and PEs is shown in Table 1. In addition, reduction ratios of the number of comparators in comparison with previous works are listed in Table 1.

When process width is n, the number of required comparators is $(\#\ PE)\times n+(\#\ AE)\times\frac{n(n-1)}{2}$. When process width increases, because the length of substring becomes longer, the number of PEs decreases. However, when substring length increases, the number of AEs increases because the number of identical substring-prefixes decreases.

RR in Table 1 which is computed by Eq. (1) represents the reduction ratio of the number of comparators. In [1] and [2], when process width increases, reduction ratio increases. For example, reduction ratio for the *deleted* rule increases by 66.4–70.8%, compared with [1]. The main reason is that the required number of comparators in the proposed architecture increases linearly, but the required number of comparators in previous works increases exponentially. However, in the architecture that does not adopt process element

Table 1 Comparison of hardware cost for *Snort* rule sets.

Rule set	*process width* = 3					*process width* = 4				
	# AEs	# PEs	*RR of* [1]	*RR of* [2]	*RR of* [3]	# AEs	# PEs	*RR of* [1]	*RR of* [2]	*RR of* [3]
chat	44	145	**61.8%**	**40.8%**	**11.7%**	44	114	**64.8%**	**47.9%**	**10.7%**
sql	37	333	**69.5%**	**23.8%**	**22.8%**	43	260	**73.6%**	**50.0%**	**22.4%**
backdoor	625	2,810	**65.1%**	**17.3%**	**10.5%**	744	2,298	**66.7%**	**37.2%**	**14.6%**
deleted	440	2,251	**66.4%**	**28.5%**	**18.2%**	486	1,802	**67.0%**	**47.1%**	**16.8%**
Rule set	*process width* = 5					*process width* = 6				
	# AEs	# PEs	*RR of* [1]	*RR of* [2]	*RR of* [3]	# AEs	# PEs	*RR of* [1]	*RR of* [2]	*RR of* [3]
chat	44	95	**65.5%**	**48.8%**	**10.3%**	44	87	**65.8%**	**45.9%**	**9.8%**
sql	47	227	**75.6%**	**57.7%**	**21.9%**	51	192	**75.7%**	**62.9%**	**20.9%**
backdoor	823	1,989	**66.2%**	**40.9%**	**10.5%**	864	1,772	**65.0%**	**40.6%**	**7.8%**
deleted	494	1,531	**71.1%**	**51.7%**	**15.3%**	510	1,352	**70.8%**	**52.7%**	**13.5%**

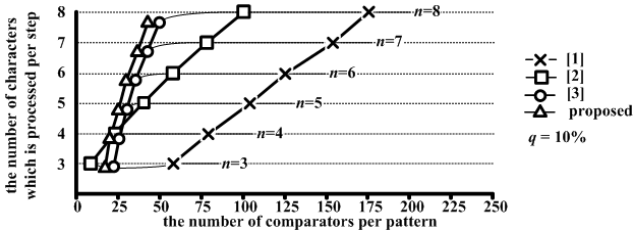

Fig. 3 Comparison of the number of comparators per pattern and performance for each process width.

tree [3], reduction ratio decreases when process width increases because the number of AEs increases. And if many patterns which have common substring-prefix are existed in rule, reduction ratio is high, such as the *sql* rule.

$$RR = \frac{C_{previous} - C_{proposed}}{C_{previous}} \times 100\% \quad (1)$$

$$(C_k : \#\ of\ comparators\ in\ k)$$

When input text alignment is used, throughput can be degraded. In other words, when partial matching occurs, AE consumes an additional step to align input text. Therefore, throughput degradation depends on process width n and the ratio of patterns in input text q. Although the proposed method reduces the hardware cost, the additional throughput degradation compared with [3] does not occur. The reason is that the proposed method shares only AEs and PEs for substring-prefix.

Figure 3 using the all rule sets in the *Snort* shows both throughput degradation and hardware cost. The hardware cost is estimated by the required comparators for one pattern. And throughput is estimated by the number of characters that is processed per step. In Fig. 3, throughput of the proposed architecture is nearly equal to that of previous works. Also, the hardware cost of the proposed architecture is lower than that of the previous works except when process width is 3. The main reason is that the common substrings, whose length is 3, are increased when individual rule sets are merged into a single set. Therefore, the proposed architecture is hardware-efficient while producing the same throughput and shows higher throughput with the same hardware cost.

5. Conclusion

In this paper, we proposed a hardware-efficient pattern matching architecture. By sharing substring-prefix, duplicate elements are removed. Therefore, the proposed pattern matching architecture reduces hardware cost while performing at the same level of throughput of previous works.

Acknowledgement

This work was supported by the IT R&D program of MKE/IITA. [2009-S-043-01, Development of Scalable Micro Flow Processing Technology]

References

[1] Y.K. Chang, M.L. Tsai, and Y.R. Chung, "Multi-character processor array for pattern matching in network intrusion detection system," Proc. AINA, pp.991–996, 2008.
[2] Y.H. Cho and W.H. Mangione-Smith, "Deep network packet filter design for reconfigurable devices," ACM Trans. ECS, vol.7, no.2, article no.21, 2008.
[3] S. Ahn, H. Hong, H. Kim, J. Ahn, D. Baek, and S. Kang, "A hardware-efficient multi-character string matching architecture using brute-force algorithm," Proc. ISOCC, pp.464–467, 2009.
[4] Snort, intrusion detection system, Visit www.snort.org