OPTIMIZING MULTI-THREAD STRING MATCHING FOR NETWORK PROCESSOR BASED INTRUSION MANAGEMENT SYSTEM

Jianming Yu,[1, 2] Quan Huang[1, 2] and Yibo Xue[2, 3]
1 Department of Automation, Tsinghua University, Beijing, China
2 Research Institute of Information Technology, Tsinghua University, Beijing, China
3 Tsinghua National Lab for Information Science and Technology, Beijing, China
yujm03@mails.tsinghua.edu.cn

ABSTRACT
String matching is the core algorithm and the most time consuming operation of almost every modern Network Intrusion Management System (NIMS). In this paper we aim at integrating string matching with multi-thread parallelism to dramatically improve the performance of NIMS. The string matching procedure under multi-thread parallelism situation is modeled and researched. The results are utilized to instruct the design of an improved Aho-Corasick (AC) algorithm, named as AC_MT, for network processor (NP) based NIMS. A simplified NIMS prototype and both the AC and AC_MT algorithms are implemented on Intel's NP platform IXDP2850. The evaluation results tested with SmartBits 600 reveals that the performance of the NIMS prototype is improved by 44.7%~148.8% depending on the different lengths of the input packets and different number of threads, under both algorithms using the same number of threads situation.

KEY WORDS
String matching, parallel processing, network processor.

1. Introduction

Along with the development of network intrusion techniques, Network Intrusion Management System (NIMS) has been widely deployed in various network environments.

NIMS uses string matching to compare the payload of the packet or flow against the known patterns of intrusions. The cost of string matching is significant. For instance, in the famous open source lightweight NIDS Snort [1, 2], the string matching routines account for up to 70% of total execution time and 80% of instructions executed on real traces [3]. Furthermore, as the number of potential threats and their associated patterns do keep growing, the cost of NIMS is likely to increase as well.

These challenges motivate the researches of string matching algorithm [4, 5, 6, 7], and introducing parallel processing in NIMS [8, 9, 10, 11, 12, 13]. But they are not combined together well. The researches of parallel NIMS are focused on load balance strategies. The researches of string matching algorithm are either taking no account of parallelism, or studying based on the perfect parallel computing model, such as parallel random access machine (PRAM) [14, 15, 16, 17]. The fruits cannot fit to the chips utilized to construct NIMS: commodity processor, embedded system, network processor et al. Moreover, they are almost all single pattern matching algorithms. Considering the huge number of patterns in current NIMS, single pattern matching algorithm is not efficient.

Network processor (NP) is a special-purpose, programmable chip tailored to construct network devices. It combines the low cost and flexibility of commodity processor with the speed and scalability of custom silicon (ASIC chips).

In this work, we focus on integrating string matching with multi-thread parallelism on NP. We first modeled and studied the multi-thread string matching procedure. Then the results are used to instruct the design of an improved Aho-Corasick (AC) [18] algorithm, named as AC_MT, for NP-based NIMS. The AC and AC_MT algorithms and a simplified NIMS prototype are implemented and tested on Intel's network processor IXP2850 [19].

The rest of the paper is organized as follows. In section 2 we give a review of previous works. In section 3, we address the model and analysis of the multi-thread string matching. In section 4, we present the improved AC algorithm, AC_MT. In section 5, we give the evaluations on IXP2850. And we summarize our contributions in section 6; the open issues for further investigation are outlined as well.

2. Previous Works

String matching is finding one, or all the occurrences of a search string in an input string. In NIMS application, pattern is the search string, while payload is the input string. If only one search string is matched against the input string, it is called single pattern matching. Otherwise, it is called multiple pattern matching.

AC is one of the most classical and popular multiple pattern matching algorithms. All search strings are accepted to make up of a finite state automaton (FSA) so that the prefix of each search string is represented by only one state. AC deals with the input string character by character and has proven linear performance to the length of input string, regardless of the number and lengths of search strings. So AC is a very robust algorithm for intrusion management system.

Suppose the pattern set is P = {hers}, the alphabet is Σ. Its corresponding deterministic automaton is shown in Figure 1. There are totally five states represented as {0, 1, 2, 3, 4}. The symbol '^{}' means all the characters of Σ except the characters in {}.

Generally, the AC automaton is implemented as two tables as shown in Figure 2(a). Each table has five items corresponding to the five states. The item of MatchListTable contains pointers to a list of the matching patterns at current state. In this example, the MatchListPointer 0 to 3 are equal to zero. And MatchListPointer4 is equal to the address of pattern "hers".

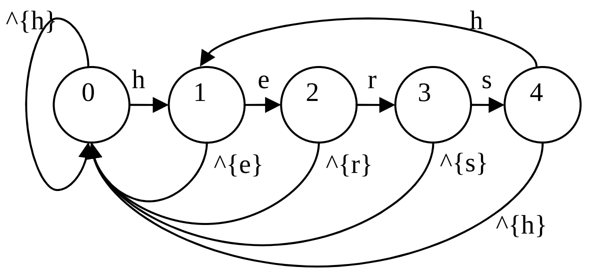
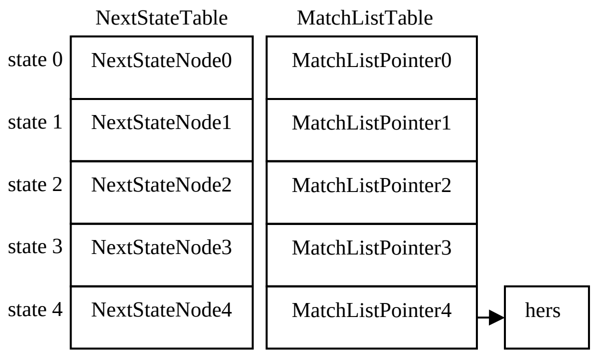


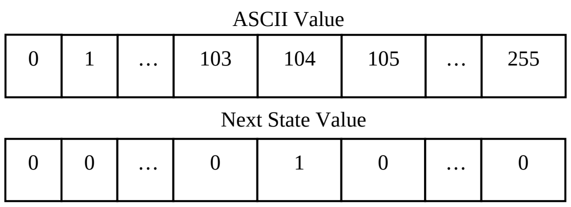
Fig. 1. The automaton of pattern set P

The item of NextStateTable contains 255 entries when using ASCII code. The implementation of NextStateNode0 is shown in Figure 2(b). The value of each entry means the state number transiting to when the input is the corresponding ASCII code number. For example, the ASCII number of character 'h' is 104, and from Figure 1 we find that from state 0 with the input character 'h', the next state is state 1. So the value of the entry corresponding to ASCII number 104 is equal to 1. From state 0, input all the other characters except 'h' will transit to itself. So the values of all the other entries except the one corresponding to 104 are equal to zero.



	NextStateTable	MatchListTable	
state 0	NextStateNode0	MatchListPointer0	
state 1	NextStateNode1	MatchListPointer1	
state 2	NextStateNode2	MatchListPointer2	
state 3	NextStateNode3	MatchListPointer3	
state 4	NextStateNode4	MatchListPointer4	hers

(a) The Tables to implement an AC automaton



ASCII Value

0	1	…	103	104	105	…	255

Next State Value

0	0	…	0	1	0	…	0

(b) The implementation of NextStateNode 0
Fig. 2. The implementation of the AC automaton

The search procedure of the AC automaton is:

```
ST = 0;  //The initial state number
i = 0;
while (i<n) //n is the length of input string
{
//T represents the input text
        // Memory Operation 1
NS = NextStateNode ST [T[i]];
        // Memory Operation 2
if(MatchListPointer NS != 0)
{
// The matching procssing
}
ST = NS;
}
```

3. Theoretical Analysis of Multi-thread String Matching

The search procedure of string matching could be abstracted as two kinds of operations: "memory operation" (reading or writing SRAM) and "calculation operation" (execution of the instructions which are not SRAM operation). When there is only one thread, CPU is idle during "memory operation". In this work, we neglect the cost of thread exchange.

3.1 Variable Definition

T: The input string.

t_s (second): The average CPU idle time due to “memory operation” for processing one byte of the input string when running only one thread.

t_c (second): The average CPU running time corresponding to “calculation operation” for processing one byte of the input string when running only one thread.

n_{TS} : The maximum number of threads could be used.

n_{TR} : The real number of threads used.

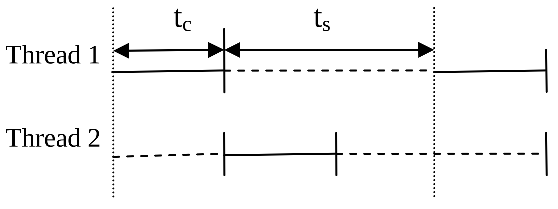
TP_n (bytes/second): The total throughput when running *n* threads.

r_n : The CPU utilization rate when running *n* threads.

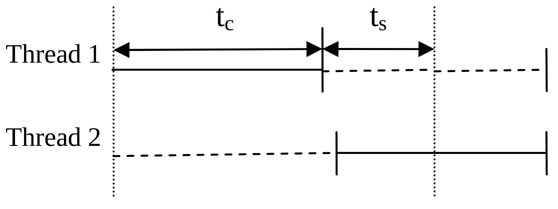
3.2 The Total Throughput of Multi-thread

When running only one thread, the time to process one byte of the input string is $t_c + t_s$. The CPU utilization rate is $r_1 = t_c/(t_c + t_s)$. The throughput is $TP_1 = 1/(t_c + t_s)$.

When running multi-thread, for example two threads, under $2t_c < (t_s + t_c)$ situation $r_2 = 2t_c/(t_c + t_s)$. The total throughput is $TP_2 = 2TP_1$. The time sequence is shown in Figure 3 (a).



(a) The time sequence under $2t_c < (t_s + t_c)$ situation



(b) The time sequence under $2t_c > (t_s + t_c)$ situation

Fig. 3. The time sequence when running two threads

Under $2t_c > (t_s + t_c)$ situation, $r_2 = 1$. The total throughput is $TP_2 = \frac{t_c + t_s}{t_c} \cdot TP_1 = \frac{1}{t_c}$. Under this situation, increasing the number of threads will not increase the throughput anymore. The time sequence is shown in Figure 3 (b).

The maximum total throughput of multi-thread string matching is calculated below. Define variable $n_T = [1/r_1]$.

(1) Under $n_T < n_{TS}$ situation:

(a) Under $n_{TR} < n_T$ situation:

$$TP_{n_{TR}} = n_{TR}/(t_c + t_s) < \frac{t_c + t_s}{t_c} \cdot \frac{1}{t_c + t_s} = 1/t_c$$

(b) Under $n_T < n_{TR} \le n_{TS}$ situation:

$$TP_{n_{TR}} = 1/t_c \text{ .}$$

Therefore, $TP_{\max} = 1/t_c$.

(2) Under $n_T \ge n_{TS}$ situation:

$$TP_{\max} = n_{TS}/(t_c + t_s) .$$

Overall, $TP_{\max} = \begin{cases} 1/t_c, n_T < n_{TS} \\ n_{TS}/(t_c + t_s), n_T \ge n_{TS} \end{cases}$.

If knowing t_c and t_s , the corresponding $TP_{\max}$ could be calculated. To any initial t_c and t_s , how to alter the values of them to improve the $TP_{\max}$ is discussed in Section 3.3.

3.3 The Effects to $TP_{\max}$ of Altering t_c or t_s

In this section, we thoroughly discussed the effects to $TP_{\max}$ of changing t_c and t_s to t'_c and t'_s . The results are shown in Table 1 and Table 2. The detailed computing course is shown in the appendix.

In Table 1 and 2, the symbol ‘-’ means invariable, the symbol ‘$\uparrow$’ means increased, and the symbol ‘$\downarrow$’ means decreased.

Table 1. The results under $n_T < n_{TS}$ situation

	$t_s \uparrow t_c \uparrow$	$t_s \uparrow t_c -$	$t_s \uparrow t_c \downarrow$	$t_s - t_c \uparrow$
$n'_T < n_{TS}$	$\downarrow$	–	$\uparrow$	$\downarrow$
$n'_T \ge n_{TS}$	$\downarrow$	$\downarrow$	①	cannot appear
	$t_s - t_c \downarrow$	$t_s \downarrow t_c \uparrow$	$t_s \downarrow t_c -$	$t_s \downarrow t_c \downarrow$
$n'_T < n_{TS}$	$\uparrow$	$\downarrow$	–	$\uparrow$
$n'_T \ge n_{TS}$	$\uparrow$	cannot appear	cannot appear	$\uparrow$

① $TP'_1 > TP_{\max}/n_{TS}$, $\uparrow$. Otherwise $\downarrow$.

Table 2. The results under initial $n_T \geq n_{TS}$ situation

	$t_s \uparrow t_c \uparrow$	$t_s \uparrow t_c -$	$t_s \uparrow t_c \downarrow$	$t_s - t_c \uparrow$
$n'_T < n_{TS}$	$\downarrow$	cannot appear	cannot appear	$\downarrow$
$n'_T \geq n_{TS}$	$\downarrow$	$\downarrow$	①	$\downarrow$
	$t_s - t_c \downarrow$	$t_s \downarrow t_c \uparrow$	$t_s \downarrow t_c -$	$t_s \downarrow t_c \downarrow$
$n'_T < n_{TS}$	cannot appear	②	$\uparrow$	$\uparrow$
$n'_T \geq n_{TS}$	$\uparrow$	①	$\uparrow$	$\uparrow$

① while $TP'_1 > TP_1$, $\uparrow$ Otherwise, $\downarrow$

② while $t'_c < 1/TP_{\max}$,$\uparrow$ Otherwise, $\downarrow$

4. The AC_MT Algorithm

The theoretical results of section 3 are utilized to instruct the design of an improved AC algorithm for NP-based NIMS. The NP utilized in this work is Intel's IXP2850. It has sixteen multi-thread process units, called MicroEngine (ME). The ME sustains hardware multi-thread, and the thread swap cost is almost zero. The maximum number of threads running in each ME is eight, i.e. $n_{TS} = 8$.

The AC algorithm is implemented and tested on IXP2850 to get the value of initial CPU utilization rate. The input string is a 32 Mbytes uniformly random text. In this paper, the pattern set used which contains 464 patterns is a sub-pattern set randomly selected from Snort (rule distributed in July 27, 2005). Table 3 shows the results.

Table 3. The CPU utilization rate of the AC

Thread number	CPU utilization rate
1	11.07%
8	88.35%

So $n_T=1/0.1107=9.03>n_{TS}=8$. According to Table 2, there are five methods to improve the maximum throughput. Among these methods, three could improve the performance without conditions: $t_s \downarrow t_c \downarrow$, $t_s \downarrow t_c -$, and $t_s - t_c \downarrow$. Other two are conditional: $t_s \downarrow t_c \uparrow$, $t_s \uparrow t_c \downarrow$.

The procedure of the AC algorithm is analyzed thoroughly. It is very hard to decrease t_c, the most feasible methods is $t_s \downarrow t_c -$ and $t_s \downarrow t_c \uparrow$. Both need to decrease t_s.

In the search procedure of AC, there are two memory operations (read *NextStateNode ST[T[i]]* and *MatchListPointer NS*) for each byte of input string. If we can decrease the number of memory operations, t_s will be decreased as well.

In this work, we utilize the highest bit of each entry in the NextStateNode to indicate whether there is a matching at this state. '1' indicates that a matching occurs. Otherwise, the bit is set to '0'. The other bits of each entry represent the original value. The number of total states could be represented decreases, but it is still enough. In the searching, after reading the *NextStateNode ST [T[i]]*, we can determine whether there are matching patterns at current state. If there are no matching patterns, we need not to read the pointer *MatchListPointer NS*. Suppose the length of each entry is 16 bits, the modification of the entry corresponding to 'h' in Figure 2(b) is shown in Figure 4. Considering the fact that a majority of the states do not have matching patterns, so the number of memory reading operation is effectively decreased.

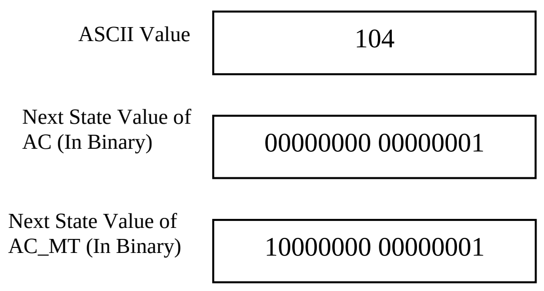

ASCII Value
104
Next State Value of AC (In Binary)
00000000 00000001
Next State Value of AC_MT (In Binary)
10000000 00000001

Fig. 4. The NextStateNode entry of AC_MT

This method incurs additional cost of bit computing, so t_c will increase. In order to validate the effectiveness, the method in Figure 4, named as the AC_MT algorithm, is implemented on IXP2850. The performance comparison of the AC and AC_MT algorithms is shown in Figure 5.

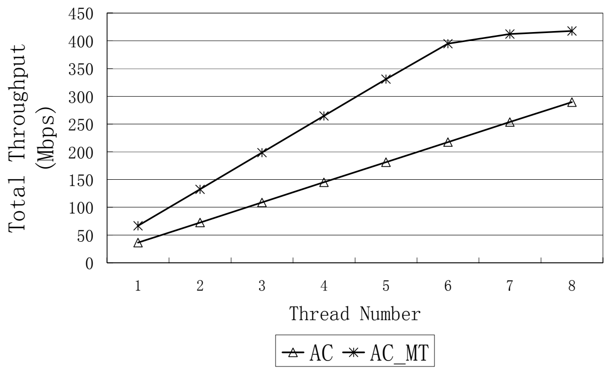


Fig. 5. The performance comparison of AC and AC_MT

In Figure 5, the horizontal axes represent the number of threads, and the vertical axes represent the total throughput of multi-thread. The input string is a 32 Mbytes uniformly random text. The performance of AC is improved by 81% when running one thread and by 44.3% when running eight threads, under both algorithms running same number of threads situation.

To the AC_MT algorithm, CPU utilization rate r_1 measured is 15.84%. So $n_T = [1/0.1574] = 6$. When the thread number is bigger than seven, $TP_{\max}$ will not increase along with the thread number increasing anymore.

5. NIMS Prototype and Evaluation

In order to evaluate the performance of AC and AC_MT in NIMS application, a simplified NIMS prototype, named as NIMS_V1, is implemented on IXDP2850 [20]. IXDP2850 is a dual-2850 NP platform released by Intel®. There are totally 32 MEs in the two NPs, and 15 of them are used in NIMS_V1. The flow chart of NIMS_V1 is shown in Figure 6. String matching algorithms, AC and AC_MT, are implemented in Detection Engine module on only one ME now. The processing after finding matching patterns is not included in NIMS_V1.

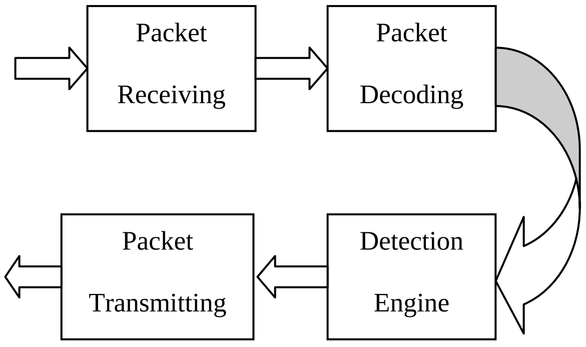


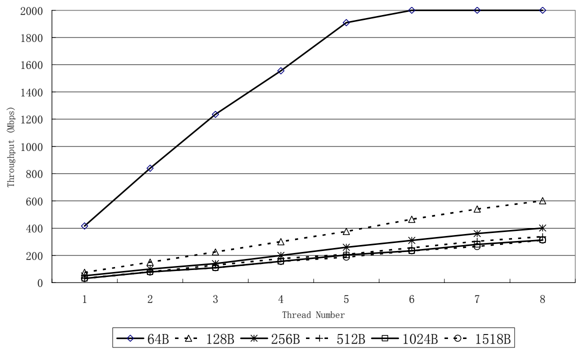
Fig. 6. The flow chart of the simplified NIMS prototype on IXDP2850

In the evaluation, SmartBits 600 (Spirent®) with two LAN-3201A (SmartMetrics 1000Base-x) cards is utilized. The software used to measure the throughput is SmartAapplication version 3.00 (Spirent®). Six different lengths of packet are tested: 64B, 128B, 256B, 512B, 1024B, and 1518B. The results are shown in Figure 7 (a) and (b). Because the SmartBits 600 we used can only provide maximum 2000 Mbps flow, we mark the value as "2000 Mbps" and do not give the accurate values when the performance is higher than it. All the values are the speed of input flow.

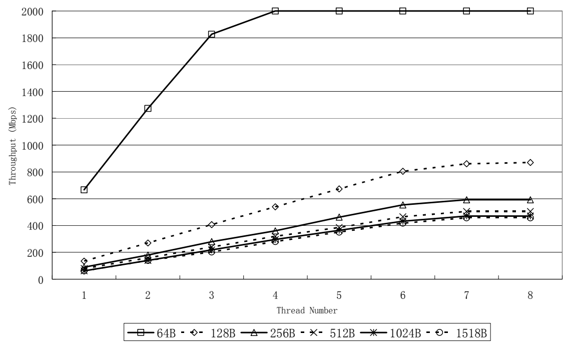
In NIMS_V1 the bottleneck lies in the Detection Engine module, so the throughput is determined by the Detection Engine. Define the throughput as *Thu*, the number of packets is *Num*, the length of packets is *Len*, and the length of packet headers is *Len_h*. $Thu = Num \cdot Len$.

Suppose the Detection Engine can process *C* bytes input string per second. So the throughput can be calculated as $c \cdot \frac{Len}{Len - Len\_h}$. This reveals that the throughput will decrease as the length of input packets increase. Because $\frac{Len}{Len - Len\_h} \to 1$ along with the increase of *Len*. The throughputs of input lengths 1024B and 1518B are close.

The performance of NIMS_V1 is improved by 60.3%~148.8% with one thread, and by 44.7~50.8% with eight threads depending on the different length of input packets.



(a) The throughput of NIMS_V1 with AC



(b) The throughput of NIMS_V1 with AC_MT
Fig. 7. The performance of NIMS_V1 with AC_MT and AC

6. Conclusions and Future Work

Designing string matching algorithms specific to NIMS application and introducing parallel processing are effective methods to improve the performance of NIMS. But they are not combined together well.

In this work we first model the procedure of multi-thread string matching. Then we utilize the results to instruct the design of an improved AC algorithm, AC_MT, for NP-based NIMS. Both AC and AC_MT are implemented and evaluated on Intel's network processor IXP2850. The experiments indicate that the performance of the algorithm is improved by 81% when running one thread and by 44.3% when running eight threads, under both running the same number of threads situation.

We also implemented one simplified NIMS prototype, NIMS_V1, on IXDP2850, and insert the AC_MT and AC in its detection engine. SmartBits 600 with two 1Gbps LAN-3201A cards is used to test the throughput of NIMS_V1. The performance of NIMS_V1 is improved by 60.3%~148.8% when using one thread and by 44.7~50.8% when using eight threads depending on the different length of input packets, under both running the same number of threads situation.

Future work could include: Apply the parallelization method to other string matching algorithms and compared with AC_MT. Do more evaluations with other testing traces such as real flow. Construct more integrated NIMS prototype on NP, so the algorithm performance could be evaluated more precisely.

Acknowledgements

We would like to thank the anonymous reviewers for providing useful comments on this paper. We would thank all the colleagues of our laboratory for their comments and enlightening discussions on draft of the paper. This work was funded by Juniper Research Grant and Intel IXA University Program.

References

[1] M. Roesch, Snort: lightweight intrusion detection for networks. *Proc. 13th System Administration Conference and Exhibition*, Seattle, Washington, USA, 1999, 229-238.
[2] M. Norton, & D. Roelker, The new Snort, *Computer Security Journal*, *19*(3), 2003, 37-47.
[3] S. Antonatos, K.G. Anagnostakis, & E.P. Markatos, Generating realistic workloads for network intrusion detection systems, *Software Engineering Notes*, *29*(1), 2004, 207-215.
[4] M. Fisk, & G. Varghese, *An analysis of fast string matching applied to content-based forwarding and intrusion detection* (Technical Report CS2001-0670, San Diego: University of California, 2002).
[5] C.J. Coit, S. Staniford, & J. McAlerney, Towards faster string matching for intrusion detection or exceeding the speed of Snort. *Proc. the DARPA Information Survivability Conference and Exposition II*, Anaheim, CA, USA, 2001, vol. 1, 367-373.
[6] K.G. Anagnostakis, E.P. Markatos, S. Antonatos, & M. Polychronakis, E^2xB: a domain-specific string matching algorithm for intrusion detection. *Proc. 18th IFIP International Information Security Conference*, Athens, Greece, 2003, 217-228.
[7] N. Tuck, T. Sherwood, B. Calder, & G. Varghese, Deterministic memory efficient string matching algorithms for intrusion detection. *Proc. IEEE Infocom 2004*, Hong Kong, China, 2004, vol. 4. 2628-2639.
[8] X. Zhao, & J. Sun, A parallel scheme for IDS. *Proc. the International Conference on Machine Learning and Cybernetics*, Xi'an, China, 2003, vol. 4, 2379-2383.
[9] I. Charitakis, K. Anagnostakis, & E. Markatos, An active traffic splitter architecture for intrusion detection. *Proc. 11th IEEE/ACM International Symposium on Modeling Analysis and Simulation of Computer Telecommunications Systems*, Orlando, Florida, USA, 2003, 238-241.
[10] H. Lai, S. Cai, H. Huang, J. Xie, & H. Li, A parallel intrusion detection system for high-speed networks. *Proc. the 2nd International Conference of Applied Cryptography and Network Security*, Yellow Mountain, China, 2004, 439-451.
[11] W. Jiang, H. Song, & Y. Dai, Real-time intrusion detection for high-speed networks. *Computers and Security*, *24*(4), 2005, 287-294.
[12] H. Cho, D. Kim, J. Kim, Y. Doh, & J. Jang, Network processor based network intrusion detection system. *Proc. the 2004 International Conference on Information Networking*, Busan, Korea, 2004, 973-982.
[13] F. Li, H. Zhang, K. Yang, Researches on parallel intrusion detection methods based-on network processor. *Proc. the 2004 International Symposium on Distributed Computing and Applications to Business, Engineering and Sciences*, Wu'han, China, 2004, 1040-1044.
[14] U. Vishkin, Optimal parallel matching in strings, *Information and Control*, *67*(1-3), 1985, 91-113.
[15] Z. Galil, A constant-time optimal parallel string-matching algorithm, *Journal of the Association for Computing Machinery*, *42*(4), 1995, 908-918.
[16] M. Crochemore, Z. Galil, L. Gasieniec, K. Park, & W. Rytter, Constant-time randomized parallel string matching, *SIAM Journal on Computing*, *26*(4), 1997, 950-960.
[17] A. Czumail, Z. Galil, L. Gasieniec, P. Kunsoo, & W. Plandowski, Work-time-optimal parallel algorithms for string problems. *Proc. 27th Annual ACM Symposium on Theory of Computing*, Las Vegas, Nevada, USA, 1995, 713-722.
[18] A. Aho, & M. Corasick, Fast pattern matching: an aid to bibliographic search, *Communications of the ACM*, *18*(6), 1975, 333-340.
[19] Intel Corporation, Intel IXP2850 network processor hardware reference manual, http://www.intel.com/design/network/products/npfamily/ixp2850.htm, 2004.
[20] Intel Corporation, Intel IXDP2800 advanced development platform system user's manual, http://www.intel.com/design/network/products/npfamily/ixp2850.htm, 2004.