A Parallel NIDS Pattern Matching Engine and Its Implementation on Network Processor

Jianming Yu
Research Institute of Information Technology
Tsinghua University, Beijing, China
yujm03@mails.tsinghua.edu.cn

Jun Li
Research Institute of Information Technology
Tsinghua University, Beijing, China

Abstract

At the heart of almost every modern Network Intrusion Detection System (NIDS), there is a pattern matching engine (PME). As pattern matching is the most time consuming operation in NIDS, it is highly desired to reduce the pattern matching time of each packet or flow. This paper proposed a parallel pattern matching algorithm based on Aho-Corasick (AC) algorithm and an efficient load balance policy for it. The method is implemented on Intel's IXP2850 Network Processor (NP). Experimental results show that when using eight processors, the pattern matching time of each packet or flow can decrease to 60.44%~14.42% of using only one processor. Based on the parallel algorithm, a PME utilizing parallel processing on three levels is proposed. Experimental results on IXP2850 show that the throughput speedup of pattern matching is 13.34~55.48 times.

Keywords: NIDS, pattern matching, parallel processing, network processor

1.0 Introduction

Network Intrusion Detection System (NIDS) are designed to identify attacks or intrusions against networks. As these threats can be invisible to firewalls, NIDS provides an additional layer of security and is being widely deployed in various network environments.

At the heart of almost every modern NIDS, there is a PME (pattern matching engine). Essentially, the pattern matching algorithm compares the set of patterns in the rule set (also called signature database) to the payloads of the packets. Pattern matching is computationally intensive. The pattern matching routines in Snort, a famous open source lightweight NIDS [5], account for up to 70% of total execution time and 80% of instructions executed on real traces [1].

An efficient PME is crucial to NIDS. If the capacity of NIDS cannot matching the speed of network, a passive NIDS will drop packets and thus miss attacks, while an inline NIDS will create a bottleneck for network performance. On the other hand, as the number of potential threats and their associated signatures is expected to grow, the cost of pattern matching is likely to increase further. Therefore, the pattern matching algorithm needs to be highly efficient to keep up with the increasing volume of network traffic, as well as the increasing number of patterns.

The Aho-Corasick (AC) algorithm proposed by A. Aho and M. Corasick [6] is the classic algorithm for searching multiple patterns simultaneously. Its time complexity is *O(n)*, where *n* is the length of the text which is compared with the patterns. This means that its time complexity is independent with the number of patterns in the rule set. This property makes it suitable for searching against a large set of rules.

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

This paper presents a parallel AC algorithm which can greatly decrease the pattern matching time of each packet or flow. It is implemented on Intel's NP (IXP2850) and the experimental results show the benefit of utilizing parallel processing and the hardware characteristics of NP. The paper also presented the design of a load balance policy for the parallel AC algorithm and a strategy to increase the throughput of pattern matching engine. The effects on pattern matching algorithm of using multi-processors as well as multi-threads technology are also analyzed.

The remainder of this paper is organized as follows. First section 2 reviews pervious works, after which section 3 describes the new parallel PME and its implementation on IXP2850. Section 4 then presents experiments and performance analysis of the algorithm and PME. Finally, section 5 summarizes the contribution of this research and presents conclusion and future work.

2. 0 Previous Works

The pattern matching problem (in NIDS it refers to string matching only) can be divided into two categories: single pattern matching algorithm and multiple pattern matching algorithm. The pattern matching algorithm can be described as: assume a string $D = d_0 d_1 ... d_{b-1}$, and a finite set of pattern strings $R = \{R_1, R_2, ..., R_a\}$, each composed of an ordered set of characters from an alphabet *A*. The pattern matching problem involves locating and identifying the substring of *D* which is identical to $R_j = r_0^j r_1^j ... r_{m-1}^j$, $1 \le j \le a$ where $d_s ... d_{s+m-1} = r_0^j ... r_{m-1}^j$, or to determine that *D* does not contain *R*. Here, *a* is the number of rules, *m* is the length of pattern string R_j, *s* is the starting point of the matching substring, and $0 \le s \le (b - m + 1)$.

BM algorithm (proposed by R. Boyer and J. Moore) [3] is the most well-known single pattern matching algorithm. The initial starting point is: $s = 0$. R_j is compared with $d_s ... d_{s+m-1}$ from the rightmost of R_j. BM algorithm utilizes two heuristics, bad character and good suffix, to reduce the number of comparisons (relative to brutal force pattern matching). The bad character heuristic works this way: when a mismatching character appears in R_j, R_j is shifted to right so that the mismatching character is aligned with the rightmost position at which the mismatching character appears in R_j. If the mismatching character dose not appears in R_j, R_j is shifted to the position that its leftmost character is one position past the mismatching character in *D*. The good suffixed heuristic is: when a mismatching occurs, there is a non-empty suffix that matches. Then R_j is shifted to the next occurrence of the suffix in R_j. BM takes the far most shift caused by the two heuristics. R. Horspool [4] improved the BM algorithm with a simpler and more efficient implementation (called BMH algorithm) that uses only the bad-character heuristic.

The AC algorithm is popular multiple pattern matching algorithm. It accepts all patterns in *R* to make up of a FSA (finite state automation) so that every prefix is represented by only one state, even if the prefix belongs to multiple patterns. The AC algorithm deals with the characters of *D* one by one and has proven linear performance to the length of *D*, regardless of the size of *R*..

Another widely used multiple pattern matching algorithm is the MWM algorithm designed by Wu and Manber[7]. The MWM algorithm uses bad character heuristic like the BM algorithm. But it utilizes two byte shift table. It also performs a hash on the two-byte prefix into a group of patterns. The MWM algorithm has shown its advantages to deal with large amounts of patterns efficiently. However, the performance of the MWM algorithm depends considerably on the length of the shortest pattern, because the maximum number of shifts equals to this value minus one.

G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis proposed an exclusion-based pattern matching algorithm E^2xB [8] based on the following observation: Suppose that one wants to check whether *D* contains R_j. If there is at least one character of R_j that is not contained in *D*, then R_j is not a substring of *D*. The E^2xB algorithm first checks *D* for missing fixed size sub-strings of R_j. If all the sub-strings of R_j can be found in *D*, standard string matching algorithms, such as the BM algorithm, is used to determine whether actual matching are occur. When mismatches are by far more common than matches, E^2xB could achieve a high performance.

All the above described algorithms are implemented and ever used in Snort.

3.0 Design and Implementation

3.1 Terms used in this paper

Input text *T*: the payload of one packet or the reassembled data of the flow of a packet stream.

Fragment *F*: one snippet of *T*.

Pattern *P*: one option field of NIDS rule, pattern is used to do string matching with *T*.

Pattern matching engine PME: part of NIDS which performs pattern matching on *T*.

3.2 Description of algorithm

3.2.1 The parallel AC algorithm

In order to decrease the pattern matching time of each *T*, multiple processor technology is utilized. h processors are used to construct a PMC (pattern matching cluster). As shown by the flow chart in Figure 1, the procedure is: when one *T* is received, the load balance unit divides it evenly into *h* *F*s, and signals the *h* PMUs (pattern matching unit) to process the *h* *F*s in parallel. All PMUs run AC algorithm with the same pattern set. When pattern matching is over, the *h* PMUs will signal the load balance unit and transmit their results.

It is obvious that the fragmentation of *T* could introduce false negative if we are not careful. For example, *T* is: “abcdefghij”, *P* is: “def”. $h = 2$, so *T* is divided into two *F*s as follows: “abcde”, “fghij”. Both of the two substrings miss *P* and this causes a false negative. To avoid this problem, the algorithm carries out fragmentation as follows: Suppose $T = t_1 t_2 ... t_n$, the length of the longest pattern is *w*. If one *F* ends with t_k, the next *F* starts with t_{k-w+2}. Thus no possible occurrence of a pattern will be missed by all *F*s.

3.2.2 Load balance policy

The aim of the load balance policy is to let the processing time of each PMU to be equal. The time complexity of AC algorithm is $O(n)$, n is the length of *T*. So the load balance policy used in this work is: let the length of each *F* is equal.

In other words, T is divided as follows:

$t_1 \ldots\ldots t_x$	$t_{x+1} \ldots\ldots t_{2x}$	……	$t_{(h-2)x+1} \ldots\ldots t_{(h-1)x}$	$t_{(h-1)x+1} \ldots\ldots t_n$

The length of each *F* is calculated by equations (1):

$$\begin{cases} y = x + (w-1) \\ x \cdot (h-1) + y = n \end{cases} \tag{1}$$

Where y is the length of each *F*. The i_{th} *F* is started with $t_{(i-1)\cdot x+1}$ and ended with $t_{i\cdot x+w-1}, i = 1,2,...,h$.

3.3 Description of PME

In this work, parallel processing on three levels is utilized to improve the performance of NIDS PME., as shown in Figure 1. Firstly, AC algorithm is adopted to search all the patterns in the rule set simultaneously. Secondly, parallel AC algorithm employed on *F* level to decrease the latency of processing *T*. Thirdly, multi-PMC is utilized on *T* level to increase the throughput of PME.

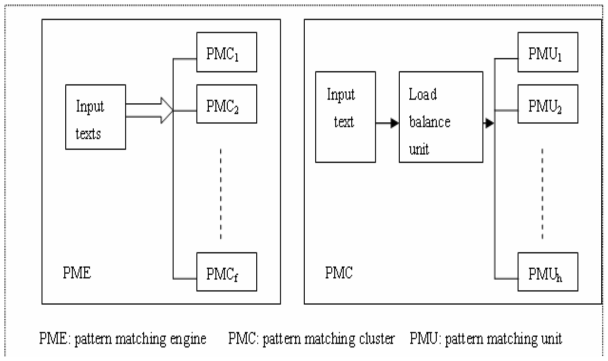
Input texts
PMC1
PMC2
PMCf
PME
Input text
Load balance unit
PMU1
PMU2
PMUh
PMC
PME: pattern matching engine
PMC: pattern matching cluster
PMU: pattern matching unit

Figure 1. The flow chart of parallel PME

3.4 Implementation on IXP2850

Intel's NP is a programmable chip tailored for network-specific applications. IXP2850 is the newest member of Intel IXP2XXX NP product line. It integrates a high-performance parallel processing design on a single chip for processing complex algorithms, deep packet inspection, traffic management and forwarding at wire speed. By combining a high-performance Intel XScale core with sixteen 32 bit independent multi-threaded MEs (microengine), IXP2850 provides more than 23.1 giga-operations per second. The detail diagram of IXP2850 is shown in Figure 2 [9].

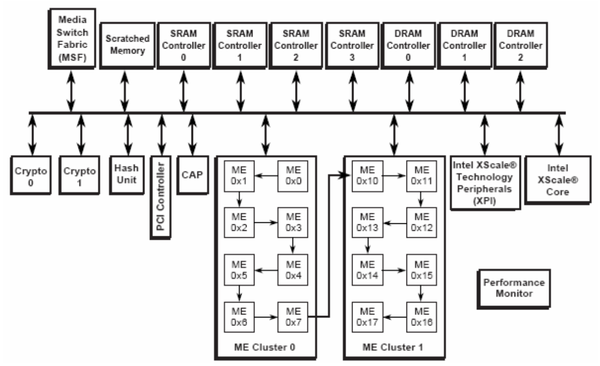

Figure 2. IXP2850 network processor [9]

The implementation of the parallel PME proposed in this work is shown in Figure 3.

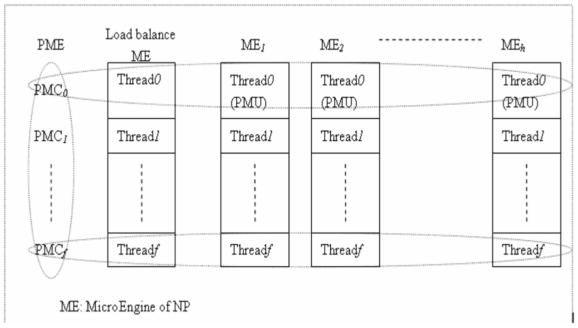

Figure 3. Implementation on IXP2850

We utilize *h+1* MEs with each ME run *f* threads to construct *f* PMCs. The threads are numbered from *0* to *f*. The *h+1* threads have the same thread number in different MEs make up of a PMC. For example, PMC$_0$ is constituted by all the Thread*0*. Load balance unit is implemented on thread*0* of load balance ME; every thread*0* of ME_1 to ME_h implements one PMU. All the threads share memory structure of AC in SRAM.

Furthermore, “defer tokens” characteristic of IXP2850 is utilized to improve the performance. Branch decision instruction in IXP2850 would cause one or more instructions in the execution pipeline to be aborted. By using “defer tokens”, one or more instructions that follow a branch decision instruction are allowed to execute before the branch takes effect. So the branch latency can be hidden if there are useful works to fill the wasted cycles after the branch instruction. With Intel’s SDK (software development kit), the deferred token can be inserted automatically by the assembler’s optimizer, or the programmer can do it manually [9].

4.0 Experiment and analysis

The added computations induced by the parallel AC are: the load balance, synchronization communication and the overlap fractions of the neighboring fragments. The load balance algorithm is very simple and its computation time is short compared with the pattern matching algorithm. On the other hand, it is in pipeline with pattern matching; this can hide the load balance computation time. The synchronization communication is mainly the signal exchange between load balance unit and PMUs. It’s execution time is also very short. The main added burden is the overlap fractions of the neighboring fragments.

Suppose the length of the input text is n. The length of the longest pattern is w. There are h PMUs in one PMC. The overlap fractions of all the fragments is: $(h-1)\cdot(w-1)$. The processing time of AC algorithm can be measured by the length of T. Ignore the time consumed on load balance; the processing time of the parallel AC algorithm can be represented as:

$$t_p = \frac{n+(h-1)\cdot(w-1)}{h} \qquad (2)$$

Obviously the latency of processing each T decreases if $t_p < n$, i.e.

$$\frac{n+(h-1)\cdot(w-1)}{h} < n \Rightarrow w < n+1 \qquad (3)$$

On the other hand, if $n < w-1$, the processing time will worsen, meaning the algorithm does not apply when the length of T is shorter than $w\text{-}1$.

The speedup of parallelism on fragment level is:

$$S_h = n/[\frac{n+(h-1)(w-1)}{h}] \qquad (4)$$

$$\frac{1}{S_h} = \frac{1}{h} + (1-\frac{1}{h})\cdot\frac{w-1}{n} \qquad (5)$$

We can find that the bigger n is, the bigger S_h is. This characteristic is very useful in NIDS, because we need to check the data flow in one session in some situations. In this scenario, n become very big. Therefore the speedup is high. When dealing with packets, according to the data in [8], the average lengths of different packet traces are all bigger than 300bytes. The length of longest pattern in snort is 37. So the pattern matching time will decrease in all the testing traces according to equation (3).

Equation (5) can also be translated into:

$$\frac{1}{S_h} = \frac{w-1}{n} + (1-\frac{w-1}{n})\cdot\frac{1}{h} \qquad (6)$$

So the bigger h is, the bigger S_h is. This means the pattern matching time decreases along with the increase of the number of PMU.

We did experiments on Intel’s IXDP2850 dual NPU platform. In the experiments the value of w is set to 41. The result is shown in Figure 4.

The horizontal axes represent h (i.e. the number of the PMU). The vertical axes represent $K_1 = 1/S_h$ and we let $K_2 = \frac{n}{w-1}$.

Figure 4 shows that the processing time decreases along with the increase of h or K_2. The results validate our analysis above. When $K_2 = 2$ and using eight PMUs, the processing time can be reduced to 60.44%. When $K_2 = 50$ and using eight PMUs, the processing time can be reduced to 14.42%.

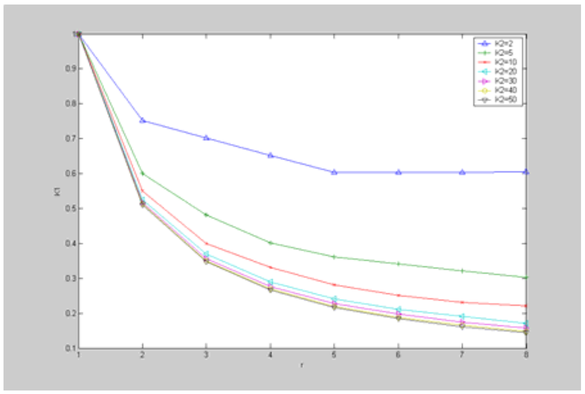

Figure 4. Performance of speedup vs. K_2 and h

Note that when $K_2 = 2$ and $h \geq 5$, K_1 increases a little bit. This is mainly caused by the synchronization communication. The synchronization time is increase along with the increase of PMUs. When n is small, the speedup of parallelism will decrease.

To analyze the effectiveness of parallelism on T level, we implemented multi-PMC with one ME. Each PMC deals with one T with the same length. All the PMCs use the same AC FSA in the SRAM. The result is shown in Figure5. The horizontal axes represent f (i.e. the number of the PMCs). The vertical axes represent K_3 = (the time of all f PMCs complete pattern matching) / (the time of running only one PMC). And we let $K_2 = \frac{n}{w-1} = 5$. We can find that, increasing the number of PMCs almost has no effect on the processing time of each PMC. In other word, the time of using f PMC to processing f different "input text" with the same length n is almost equal to the time of running one PMC to processing an input text of length n. Therefore the throughput of PME can increase f times when using f PMCs.

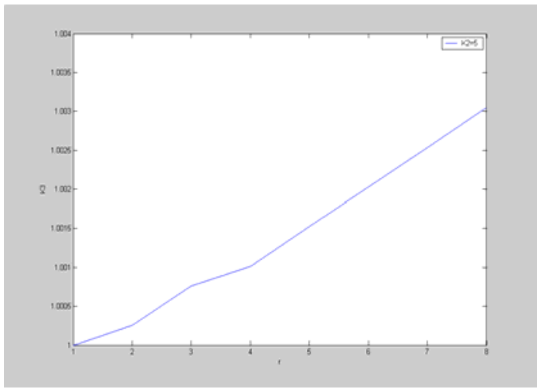

Figure 5. Performance of processing time vs. f

There are two reasons for this. First, there are many I/O operations (mainly SRAM and DRAM accesses) in pattern matching algorithm. The processor must wait for the I/O operation to complete. Hence, there are many idle cycles. Second, in the MEs of IXP2850, each context has its own register set, Program Counter, and Context Specific Local Registers. This eliminates the need to move context specific information to/from memory and ME registers when doing context swap. So the contexts waitting for I/O to complete can be swapped out with almost no cost, and allows other context to do computation.

Above all, the speedup of using all the three level parallelism is (using 8 PMUs and 8 PMCs):

$$S = \frac{1}{0.6044} \times 8 \sim \frac{1}{0.1442} \times 8 \Rightarrow 13.24 \sim 55.48 \quad (7)$$

5.0 Conclusions and future work

We have studied the performance of NIDS pattern matching algorithms, and presented the parallel AC algorithm. We implemented the parallel algorithm on NP and designed a simple and efficient load balance policy for it. Based on the parallel AC algorithm, a new NIDS PME utilizing parallel processing on three levels is proposed.

Experimental results show that the processing time of the parallel AC algorithm decreases along with the increasing of h and ratio $\frac{n}{w-1}$. When using eight PMUs, the processing time decrease to 60.44% ~14.42%, reflecting speedup of throughput of PME for 13.34~55.48 times when using eight PMUs and eight PMCs at the same time.

Furthermore, our results also allow for some more general observations to be made on the design and analysis of parallel NIDS pattern matching algorithms and the application of NP in NIDS.

Future work could include better load balance policy for parallel AC algorithm. Improve the performance of AC algorithm based on the characteristic of NP, such as increase the memory efficient. Design load balance policy for different pattern matching clusters. Apply network processor in other modules of NIDS.

Acknowledgments

This work was supported by the research grant provided to Tsinghua University by Juniper Networks, Inc.

Reference

[1] S. Antonatos, K. G.. Anagnostakis, and E. P. Markatos. “Generating realistic workloads for network intrusion detection systems.” Proc. ACM Workshop on Software and Performance, 2004.

[2] M. Fisk and G. Varghese. “An analysis of fast string matching applied to content-based forwarding and intrusion detection.” Technical Report CS2001-0670, University of California – San Diego, 2002.

[3] R. Boyer and J. Moore. “A fast string searching algorithm.” Communications of the ACM, 20(10): 762-772, 1977.

[4] R. Horspool. “Practical fast searching in strings.” Software Practice and Experience, 10(6): 501-506, 1980.

[5] M. Roesch. “Snort: Lightweight intrusion detection for networks.” Proc. 1999 USENIX LISA Systems Administration Conference, 1999.

[6] A. Aho and M. Corasick. “Efficient string matching: An aid to bibliographic search.” Communications of the ACM , 18(6): 333-343, 1975.

[7] S. Wu and U. Manber. “A fast algorithm for multi-pattern searching.” Technical Report TR94-17, University of Arizona, 1994.

[8] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis. “E^2xB : A domain-specific string matching algorithm for intrusion detection.” Proc. 18th IFIP International Information Security Conference (SEC2003), 2003.

[9] Intel Corporation. “Intel IXP2850 Network Processor Hardware Reference Manual.” http://www.intel.com/design/network/products/npfamily/ixp2850.htm