國立清華大學

碩士論文

題目：網路處理器上快速字串比對方法之研究

A Fast String Matching Algorithm Based On Network Processor

所別：資訊工程研究所

學號姓名：904397 周智杰

指導教授：黃能富教授

中華民國九十三年一月九日

摘　要

隨著網路速度不斷地增快，單一處理器已無法滿足高速網路設備的需求，網路處理器的出現解決了此一問題。然而，對於大部份需要檢視封包內容的網路設備來說－如入侵偵測系統，如何對封包做更有效率的比對，將成為影響網路設備效能的重要因素。

以目前知名的入侵偵測系統 Snort 為例，其對於封包比對所用最新的方法為 Sun Wu 及 Udi Manber 於 1994 年所提出的 A Fast Algorithm For Multi-Pattern Searching 演算法(簡稱為 WuM)。此演算法可以同時快速比對 Multi-Pattern，但前題是 Pattern 的最小長度必需大於 1，若 Pattern 的最小長度為 1，則將只能使用最原始的 Single-Pattern 比對方法(Boyer R.S., and J.S. Moore 1977－簡稱為 BM)，一個個比對完所有的 Pattern，如此將使得封包的處理皆被限制在 Pattern 的比對上，大大降低其效能。

本論文的主要目的，在於改良 Sun Wu 及 Udi Manber 的演算法(快速 WuM，簡稱為 FWM)，有效增加 Pattern Search 的速度，並使得其可以處理最短 Pattern 長度(Length of Shortest Pattern)為 1 的情形。如此，便不用因為 LSP=1，而使得比對的方式需改用 Single Pattern 的比對方式逐一比對了。同時針對 Network Processor 提出增進效能的方法，增加設備的 Throughput。

為了驗證 FWM 演算法的效能，我們利用 Snort 裡所定義的 Signatures 當做 Pattern，同時由 DEFCON 取得攻擊封包做為 Input，與傳統的 Multi-Pattern Searching 演算法 WuM、E2xB 比較，得到了 15%到 25%的效能提升。FWM 主要貢獻有三：(1)解決當 LSP=1 時，便無法做 Multi-Pattern 的 Search 問題。(2)提升 WuM 演算法的效能。(3)適合用於 Network Processor 平台上。

ABSTRACT

With the development of high-speed network, uni-processor incapable of affording a large number of traffic can not satisfy what is required by high performance network equipments. It can help improve the performances. However, most network equipments such as network intrusion detection or protection systems need to inspect the packet content and compare with its own signatures, and react appropriately. Thus, the need for a faster algorithm for multi-pattern searching becomes more and more urgent. It is the most crucial factor concerned with the network performance.

Take Snort [18], a popular open-source network intrusion detection system as an example, it uses the algorithm called “A fast Algorithm For Multi-Pattern Searching” proposed by Sun Wu, and Udi Manber 1994(denoted as WuM) [6]. The WuM algorithm can compare the input text with the whole patterns concurrently, but the length of the shortest pattern (denoted as LSP) can not be less than the block-size usually equal to 2. If LSP is less than the block-size, in snort, it compares the input text with its own signatures sequentially using the BM algorithm which is proposed by Boyer R. S., and J. S. Moore 1977 [1]. Consequently, the throughput is limited by matching patterns, and has the poor performance.

The purpose of the thesis is to improve the performance of WuM algorithm, and to handle the length of the shortest pattern less than the block size. Therefore, we can use the only one algorithm to perform multi-pattern searching even the LSP is equal to 1. We concentrate on typical searches rather than on worst-case behavior. It makes sense in most network devices which need to compare incoming packets to its own patterns. Malicious packets in network are always less than legal packets.

To verify the performance of FWM, we use the signatures defined by snort as the patterns, and use the packets downloaded from DEFCON [17] as the input to run the simulation. Finally, we got 15% - 25% performance improvement.

CONTENTS

FIGURES AND TABLES

1. INTRODUCTION

1.1 The Use of Multi-Pattern Searching

Typically, many of the network devices such as network intrusion detection system (NIDS) try to detect anomalous behavior by inspecting the incoming packets. Rapid growth of network traffic has made NIDS become more important. In general, there are two main techniques used for detecting the intrusion – one based on statistical analysis and the other on signature. The statistical analysis based technique usually determines whether an incoming packet is anomalous by gathering protocol header information and comparing it with the known attacks such as SYN Flooding. The signature-based technique usually has its own rules or signatures which are defined in advance to represent known intrusive attacks such as MS.Blaster.Worm virus. When a packet comes, NIDS has to compare it with all the signatures and determine whether or not it is an intrusion. Unfortunately, successful detection is increasingly difficult due to more and more fresh viruses and the modification of an old virus detection. The performance of signature-based NIDS is seriously constrained by the speed of pattern matching algorithms. For example, as mentioned in abstract, Snort uses the WuM algorithm [6] to compare incoming packets with its signatures while the length of shortest pattern is greater than or equal to 2. If there exists one signature whose length is equal to 1, Snort will use only BM algorithm [1] to compare the incoming packet with its all signatures sequentially. Thus, the number of signatures is the critical determinant of system performances. Only by implementing an algorithm which can search pattern concurrently, unconstrained by pattern number or length can upgrade the effectiveness of the NIDS system.

1.2 Effect of Pattern Matching Algorithm

As mentioned in 1.1, the performance of pattern matching algorithm affects the throughput of network device basing on incoming packets examination such as Network Intrusion Detect System. A certain set of signatures defines how a Network Intrusion Detect System functions. It examines incoming packets and determines whether it is an intrusion.

Recent measurements of the snort NIDS show that as much as 31% of total processing is used for pattern matching, as shown in Table 1. Thus, without better efficiency, it is hard for the NIDS system to keep up with soaring linking speed. In other words, more effective pattern matching results in an increased throughput for the NIDS.

Prupose	Routine	Portion
String Match	mSearch	31%
Packet Classification	EvalHeader	8.5%
Packet Classification	CheckSrcIPNotEq	6.7%
Other Matching	EvalOpts	5.8%

Table 1.1 Profile of Snort

1.3 Additional Applications

In fact, pattern matching technique has extended its applications beyond network intrusion detection system. Web site and advertisement e-mail filtering device are only two examples where pattern matching technique is employed to prevent users receiving. The web site filtering device parses the URL and compares with its own pattern by some user-defined keywords usually found in a pornographic site. And the

advertisement e-mil filtering device parses the sender or subject to filter the garbage or advertisement e-mail.

Grep [8] is a well-known tool in UNIX capable of searching the whole file quichly and reporting the lines once a pattern is matched. It can not only input the single pattern but also assign the multi-pattern from a file with the parameter ‘f’.

Match-and-replace utility is used in many editors. Each pattern is associated with a replacement pattern. When a pattern is matched, it is replaced. But how long we must shift to avoid overlapping replacements, it is also a multi-pattern searching problem.

Another application is search engine in World Wide Web such as Google, Openfind. Users may input some key words and the search engine must find all the possible pages which contains the key words.

2. Previous Work

Aho and Corasick presented a linear-time algorithm (AC) [2] for matching multiple strings. The algorithm is based on an automata approach that accepts all strings in the set. It processes the input characters, and follows the state transition diagram. If it reaches the final state, the input text makes a match. So, the performance depends on the length of the input text rather than pattern length. The AC algorithm has proven linear-time performance, and it's optimal in the worst case.

Boyer-Moore is another powerful algorithm (BM) in single-pattern searching. It can skip a large portion of the input text while searching. At first, it builds a table called bad character shift table, and then compares the string with the input text starting from the right most character of the string. If the mismatching character is in the search string, the search string can be shifted to align with the rightmost position at which it appears in the search string. If the mismatching character is not in the search string, we can safely shift the maximum distance – the pattern length. In average case, the BM algorithm is faster than linear algorithm.

Among the single pattern searching algorithm mentioned above, BM algorithm is the fastest, but its worst case is slower than AC algorithm in some worst cases. In sum, BM and AC algorithm are two best methods in single pattern search.

K. G. Anagnostakis, M. Polychronakis, E. P. Markatos, and S. Antonatos have designed an informal algorithm called exclusion-based string matching denoted as ExB [10]. ExB is based on a simple logic. If the input text 'I' contains a string 'S', then, if there is at least one character which is in the string 'S' and is not in input text 'I'. Then 'S' is not in 'I'. Moreover, if every character of 'S' belongs to 'I', it still needs another algorithm such as BM to confirm whether 'S' is a substring of 'I'.

E^2XB [9] is based on ExB algorithm. The difference between ExB and E^2XB is the method used to denote a match. So the performance of both ExB and E^2XB decrease rapidly because of the increasing false matching rates.

Wu-Manber is another widely used multi-pattern algorithm. In Unix, the words searching tools such as grep [8], agrep using WuM algorithm to reach the goal. The WuM algorithm is also based on bad character heuristic similar to Boyer-Moore. But the WuM algorithm covers a concept called block. It uses one or two-byte bad shift table by pre-processing the all patterns, and performs a hash on the two-character prefix of the input text to compute an index which is the location on the bad shift table, as in Boyer-Moore. The performance of the WuM algorithm also depends on the shortest length of the pattern (LSP). Its maximum shift number equals to LSP minus one.

3. The Algorithm

3.1 Background

3.1.1 Network Processor

Network processor is a programmable device that has been designed and highly optimized to perform networking functions. Because network processors implement all packets processing in software, they are more flexible. An NP based platform can be used a variety of packet processing function, such as table lookup, parsing, classification, modification, and forwarding. Network processors use multiple execution engines, each of which contains multiple contexts.

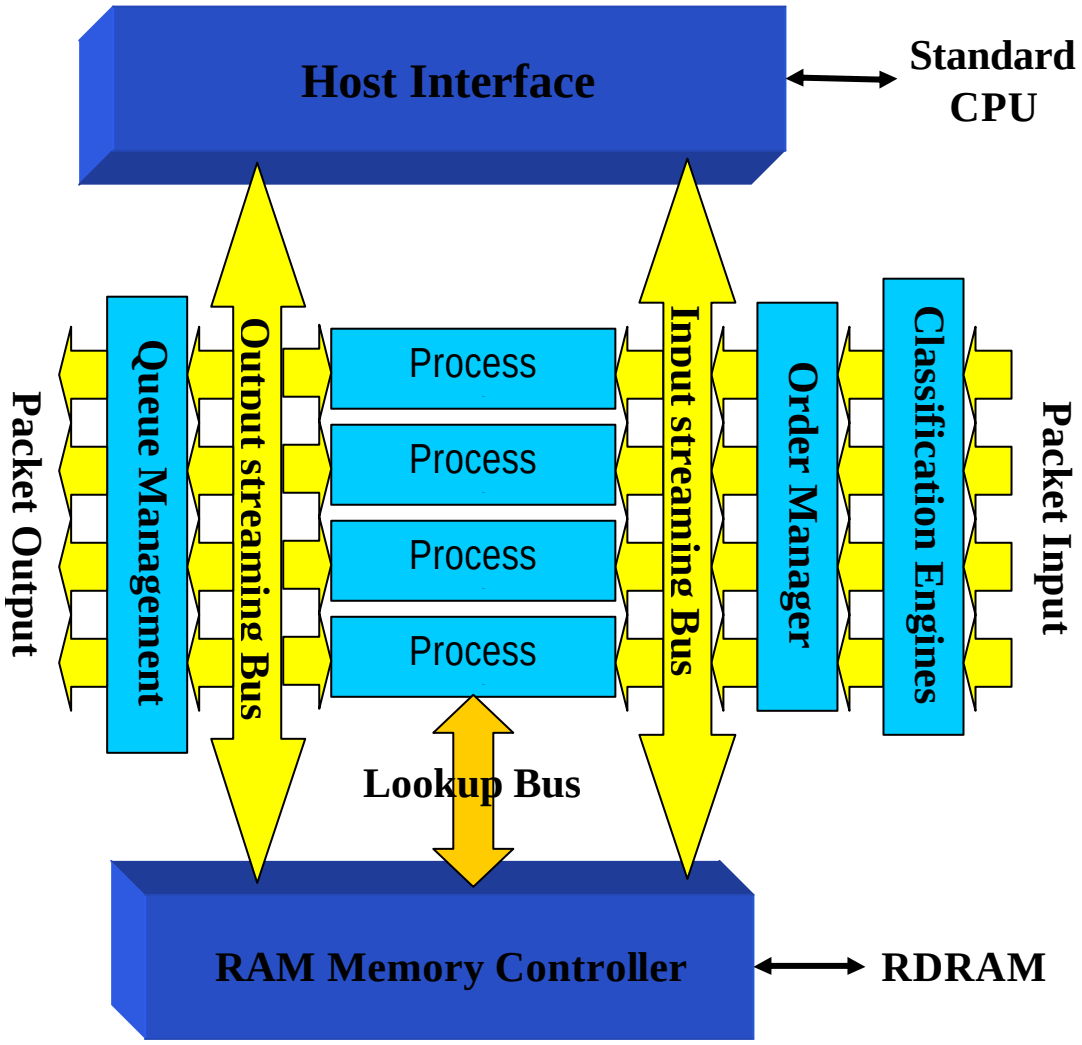


Figure 3.1 Network Processor Architecture

For example, Vitesse IQ2000 [19] contains four packet processing engines (PPE), each of which runs at 200MHz, and is RISC architecture. Each packet processing engine contains DMA co-processors, five stage pipelines, lookup co-processors, header buffers, and so on.

Network processors usually use pipelining, parallelism and multi-context to reduce latency. Network processors also exploit hardware accelerators for hashing, table lookup and forwarding. But the most important the network processor differs greatly from general purpose CPU in L1, L2 cache. We all know that general purpose CPU architecture usually exploits both the data and program locality, and has cache architecture called L1, L2 cache. It makes the program run smoothly. But in network processor architecture, it is not really true. Network processors usually have no L1, L2 cache. It just exploits other techniques such as mentioned above to increase its performance. We also use the special characteristic to increase the performance for pattern matching on network processor.

3.1.2 Wu Manber Algorithm

Sun Wu and Udi Manber proposed a fast algorithm for multi-pattern searching on May 1994. We refer to it as WuM algorithm.

WuM algorithm also uses the ideas developed Boyer-Moore and are summarized as followed. Suppose that the pattern length is m, we can compare the last character of the pattern against t_m which is the m'th character of the input text. If there is a mismatch, then we can shift according to the rightmost occurrence of t_m. For example, if t_m matches the 1th character of the pattern, we can shift m-1. If t_m matches the 2th character of the pattern, we can shift m-2. If there is no any match of the pattern, we can safely shift the maximum m characters and so on.

Because BM algorithm can shift greater than one character, it can perform well than linear time algorithm.

3.1.2.1 The Preprocessing Stage

BM algorithm just can handle the single pattern. WuM algorithm designs to handle multi-pattern searching. Suppose Let $P = \{p_1, p_2, p_3, ..., p_k\}$ be a set of patterns and Let $T = t_1, t_2, t_3 ..., t_n$ be an input text. The first step by WuM method is to compute the minimum length of a pattern (denoted as m). For each pattern we just consider only the fist m characters and assume that all patterns have the same length m. Thus, in case of there exists a very short pattern, say of length 2. We can not shift more than 2. Therefore, WuM algorithm can not handle the case in which m is equal to 1.

Secondly, WuM algorithm is characterized by the idea called blocks of size B, and usually uses B=2 or B=3. Moreover, WuM algorithm also includes a shift table which plays the same role as that in BM algorithm. The only difference between WuM and BM is that WuM determines the shift based on the last B characters rather than just one character. Thus, the maximum shift character is equal to m-B+1. We can say that BM is a special case of WuM algorithm.

Thirdly, we need to determine the shift table. If the shift table contains all the possible string of size B, it needs to allocate $2^{8 \cdot B}$ bytes to store shift values. WuM actually uses a compressed table with several strings mapped into the same entry (hash) to save space. To speed up the performance, we hope that the hash value which is computed from the characters in blocks can also be taken as an index of shift table. The values in the shift table determine how far we can shift while we scan the input text. Suppose let $T = t_1, t_2, t_3, ..., t_n$ be the input text,

$P = p_1, p_2, ..., p_m$ be the pattern whose length is equal to m. We just consider the first B characters $\{t_1, t_2, ..., t_B\}$ in the block. There are two cases:

1. $\{t_1, t_2, ..., t_B\}$ appears in some patterns:

   In this case, if we find the rightmost occurrence of T in any of the patterns, and it ends at position q of P_j. We may have a lot of q, but we can just store the minimum q in the corresponding entry to avoid false matching.

2. $\{t_1, t_2, ..., t_B\}$ doesn't appear in any patterns:

   In this case, we can store m-B+1 in the corresponding entry.

Let's take an example as follows:

Assume that $P_1 = \{abcdef\}$
$P_2 = \{xyabz\}$, and B = 2
$P_3 = \{uxyv\}$

Considering P_1: In case of 'cd', we can shift 0
In case of 'bc', we can shift 1
In case of 'ab', we can shift 2
Others, we can shift 3

Considering P_2: In case of 'ab', we can shift 0
In case of 'ya', we can shift 1
In case of 'xy', we can shift 2
Others, we can shift 3

Considering P_3: In case of 'yv', we can shift 0
In case of 'xy', we can shift 1
In case of 'ux', we can shift 2
Others, we can shift 3

In this case, if we find 'ab', we can shift 2 according to P_1, but we just can shift 0 according to P_2. Thus, we must shift the smaller one – '0' to avoid false matching. P_2 and P_3 belong to the same case. If we find

'xy', we can shift 2 according to P_2, but we should shift 1 according to P_3. If we find any string which is not a substring in any pattern, we can shift 3 that is m-B+1 $\Leftrightarrow$ 4-2+1=3.

As long as the shift value is greater than 0, we can shift and continue the scan while we start to scan the input text. In general case, it happens most of time, especially in NIDS. Illegal packets are usually less than legal packets. If the shift value is 0, it is possible that the current text contains a substring in perfect match with some pattern in pattern list. Thus, we need to compare the substring to suspected patterns. In WuM algorithm , it uses hash function to classify all the patterns. As mentioned above, WuM uses a key which is computed with the substring in the block to be an index of shift table. Thus, if the value is equal to 0, we use the same key to be a hash value for patterns classification. Patterns whose suffixes are the same will have the same key. And they will be accommodated to the same list.

We use shift table and hash table to speed up the matching. And we use the suffix matching to represent a matching. But it is not random in natural language texts. The suffixes such as 'tive', 'tion' or 'ing' are very common. They also cause collisions in the hash table and increase the number of patterns which we need to compare the text against the pattern directly and the performance is thus unfavorably affected.

To avoid collisions, WuM algorithm uses another table called prefix table to reduce the probability of collisions. After scanning the text, computing the index and looking up the shift table, if the shift value is equal to zero (An indication that the text has the same suffix as some pattern), we need to traverse pattern list to determine whether there is a matching. Before traversing, we can check the prefix table. If there is still a matching (Indicating that the text has the same prefix as some pattern), then we compare the text against the pattern directly.

```
Initially: ptr ← text start + block-size
           end ←text end
While ptr < end
Begin
    hash_value ← (*ptr << hash_bit) + (*(ptr-1))
    if block-size == 3 then
    hash value ← (hash_value << hash_bit) + (*(ptr-2))
    shift_value ← SHIFT[hash_value]
    if shift_value == 0
    Begin
        prefix_hash ← (*(ptr-m+1)<<8) + (*(ptr-m+2))
        /* we shift 8 bits to avoid collision in prefix table */
        pat_ptr ← Hash[hash_value]
        while pat_ptr != NULL
        Begin
            if PREFIX[prefix_hash] != text_prefix continue
            p ← pat_ptr[0]
            p ← text-m+1
            while *(p++) == *(q++)
            if *(p-1) == 0   /* C-String */ → Match
            pat_ptr ← pat_ptr->next
        End
        shift_value ← 1
    End
    ptr ← ptr + shift_value
End
```

Figure 3.2 WuM pseudo code

3.1.2.2 The Scanning Stage

In scanning stage, we first compute a hash value h based on the current B characters from the input text. (B is the block-size here). Secondly, check the value of SHIFT[h]. If SHIFT[h] > 0, shift the text and repeat scanning. Otherwise, we need to compute the hash value of the prefix of the text. Thirdly, if there is a matching in prefix, we can traverse the pattern list which has the same hash value to determine whether matching or not. WuM algorithm pseudo code is shown in the Figure 3.2.

3.2 FWM algorithm

3.2.1 The Principle

This work addresses the string matching problem: Let $P = \{p_0, p_1, p_2, ..., p_k\}$ be a set of patterns, which are strings of characters. Let $T = t_0, t_1, t_2, ..., t_n$ be an input text. Patterns and text consists of either ascii or binary strings. The string matching problem identifies the substring of T which is identical to P_j.

As mentioned in section 3.1.2.1, we know that the WuM algorithm runs faster than linear time algorithm by skipping a large portion of the text while searching, and the maximum distance which it can skip is equal to m-B+1. In other words, it equals to the length of the shortest pattern minus block-size and plus one. For example, if the length of the shortest pattern is equal to 3 and the block-size is equal to 2. Thus, the maximum distance that we can skip is equal to 3-2+1 = 2.

For the above reason, we know that WuM algorithm is sensitive to the pattern length. If the length of the shortest pattern 'm' is larger, then we can have the larger shift value. In other words, if there exists a pattern whose length is equal to 2, we just can have the maximum shift value 1 in spite of the rest of patterns whose length are very long.

As long as an algorithm can shift larger distance than other algorithms, equally, it can run faster than others. According to maximum shift distance of the WuM algorithm, there may be two method to improve the shift distance. Clearly, we can increase the length of the shortest pattern 'm' or use a smaller block size 'B'.

In WuM algorithm, suppose that M be the total size of all patterns and c be the size of the alphabet. A good value of B is in the order of $\log_c 2M$. In fact, the implementation of WuM algorithm in agrep [8] uses

only B=2 and B=3. If we decrease the B value, in other words, we set B=1, there must be a number of collisions in shift table, and we must more often check the text against the pattern directly, and reduce the performance. Thus, decreasing B value is impossible.

A possible solution is increasing the length of the shortest pattern 'm'. Of course, we can eliminate the pattern whose length is short. But we need to keep two algorithms at hand to search patterns. Here we propose a technique called look ahead to increase the length of the shortest pattern. Example 1:

Assume that $P_1 = \{abcdef\}$
$P_2 = \{xyabz\}$ and B=2
$P_3 = \{uxyv\}$

We can shift maximum distance 4-2+1=3, and have the shift table shown as table 3.1

P1

Case	Shift value
cd	0
bc	1
ab	2

P1 & P2

Case	Shift value
cd	0
bc	1
ab	0
ya	1
xy	2

P1 & P2 & P3

Case	Shift value
cd	0
bc	1
ab	0
ya	1
xy	1
yv	0
ux	2

Table 3.1 Shift Table of Example 1

Now, we use example 1 and consider it with a look-ahead character. Firstly, when we process P_1, substring ‘ab’ is the maximum shift value if the substring in block is identical to ‘ab’ originally. In other words, we can safely shift 2 characters, and continue checking the next substring in block. While considering a look-ahead character, we need to add an entry ‘*a’ to shift table where * is a wild card. Similarly, we process P_2 and P_3 in the same way. Therefore, while using a look-ahead character, we can change the pattern length of the shortest pattern from 3 into 4. The newest shift table with a look-ahead character is shown in Table 3.2

P1

Case	Shift value
cd	0
bc	1
ab	2
*a	3

P1 & P2

Case	Shift value
cd	0
bc	1
ab	0
ya	1
xy	2
*a	3
*x	3

P1 & P2 & P3

Case	Shift value
cd	0
bc	1
ab	0
ya	1
xy	1
yv	0
ux	2
*a	3
*x	3
*u	3

Table 3.2 Shift Table of Example 1 with a look-ahead character

Comparing Table 3.2 to Table 3.1, we can find that the maximum value is 2 in Table 3.1, but is 3 in Table 3.2. Moreover, the wild card ‘*’

means that we must enumerate all the possible ones in the character set. For example, we have a look-ahead character on P_1, then we must fill all the entries whose suffix are 'a' with the value 3. When we process P_2, there is a collision on 'ya'. Similiarly, we must choose the smaller one to avoid false-matching. Thus, we fill the entry of 'ya' with the value 1.

When the length of the shortest pattern is very small, it would be more advantageous with a look-ahead. If the length of the shortest pattern is equal to 2, then the rate of performance improvement is equal to $\frac{(3-2+1)-(2-2+1)}{2-2+1}=\frac{1}{1}=100\%$. If the LSP is equal to 3, the rate of the performance improvement is equal to $\frac{(4-2+1)-(3-2+1)}{3-2+1}=\frac{1}{2}=50\%$, and $\frac{(5-2+1)-(4-2+1)}{4-2+1}=\frac{1}{3}=33.33\%$ with LSP = 4. Thus, we know that the performance improvement is inverse proportion to the LSP.

How about increasing the block-size? Originally, we can shift the maximum distance (m-B+1). If we fix the block-size to 2, then we can shift the maximum distance (m-1). If including a look-ahead character, we can shift the maximum distance 'm'. Now, let's increase the block-size to 3 and take two look-ahead characters. We will have the same maximum distance 'm' as the block-size fixed to 2.

We need another technique to raise the performance that makes it insensitive to LSP. Thus, we build a table called occurrence table to handle patterns whose length are less than block-size. Occurrence table is extremely intuitive. We merge the shift table with the occurrence table. In scanning stage, we compute a hash value based on the current B characters from the text, and check the value of shift table. We can determine whether we match some pattern whose length is less than block-size with the shift value for some pattern whose length is greater than block-size in the same time.

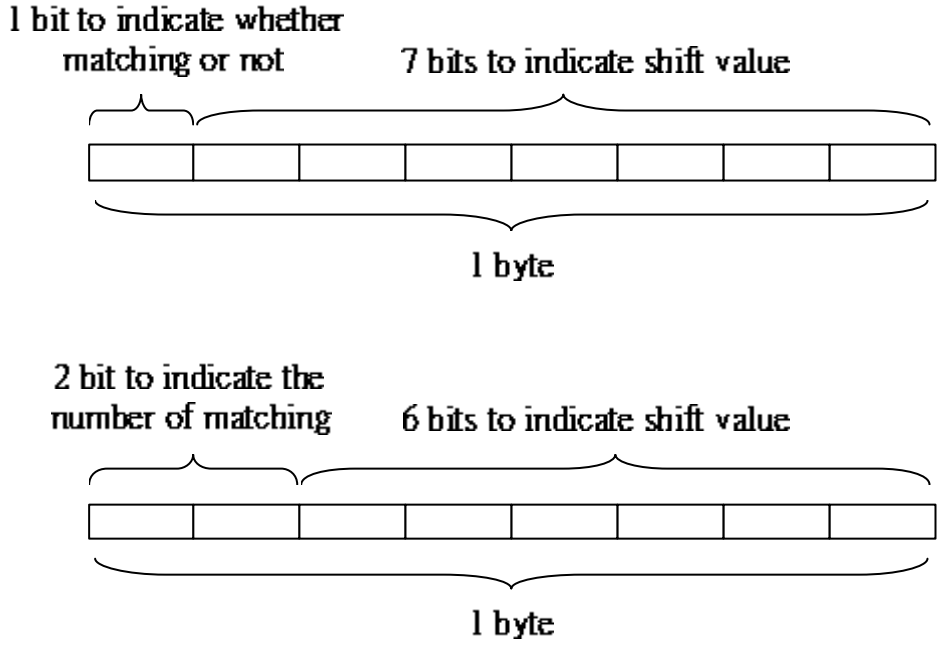
On general purpose CPU architecture such as x86, the size of occurrence table will affect the performance. If we use the block-size is equal to 2, we need $2^{8*2} = 32K$ bytes to store the table. But when the block-size increases to 3, we will need $2^{3*8} = 16M$ bytes to store the table. We all know that CPU uses cache architecture to store the most important data to avoid the access to external memory and make the performance better. The capacity of cache is usually less than 1M bytes. If we use a very large table like 16M bytes, we may probably have inferior performance. Fortunately, as mentioned in section 3.1.1, network processors usually don't have cache system, and the cost of memory is much cheaper than others on the platform of network processor. Thus, we can use masses of spaces to reduce the searching time.

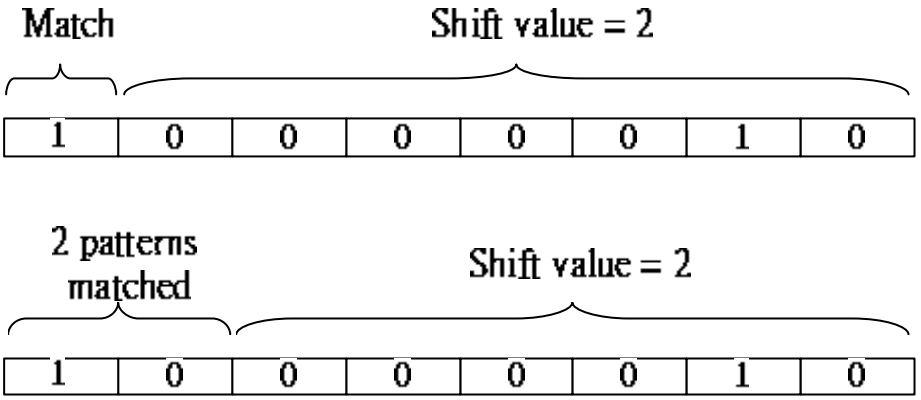
3.2.2 Design

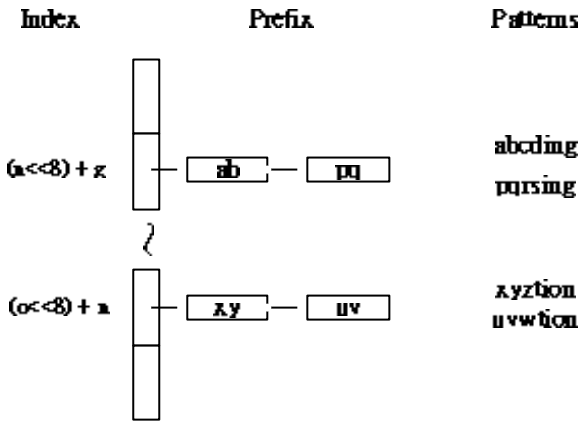
3.2.2.1 Shift Table

We use the shift table to determine shift value and whether matching the short pattern while scanning the text. In order to reduce the access time to memory, and use an occurrence table, we use a straightforward hash function. For example, suppose let $P_j = abcde$, and block-size be 2. We can store the shift value of 'ab' in entry $(a << 8) + b$, and store the shift value of 'bc' in entry $(b << 8) + c$ to avoid collisions. Thus, we can get the shift value in one memory access. Again, we must emphasize that the trick is only work on network processor platform.

We use the straightforward hash function to avoid collisions. Another purpose is to point out whether it is matched or not for the small patterns. Therefore, we split one byte into two parts. One is the shift value, and the other is matching-flag as shown in Figure 3.3.




Match
Shift value = 2
1
0
0
0
0
0
1
0
2 patterns matched
Shift value = 2
1
0
0
0
0
0
1
0


Index
Prefix
Patterns
(a<<8) + z
ab
pq
abcding
pqrsing
(o<<8) + a
xy
uv
xyztion
uvwtion

3.2.3 Implementation

During initialization, we must acquire the length of the shortest pattern, and use it to compute the maximum shift value. If the block-size is equal to 2, we will build a 32K bytes shift table, and hash table. Similarly, if the block-size is equal to 3, we will build a 16M bytes shift table, and hash table. All entries are set to the maximum shift value 'm-B+1' in the shift table and set to zero in the hash table. Initialization pseudo code is shown as Figure 3.6.

Initialization:

```
if(FLAG_LOOK_AHEAD)
    LSP += (BlockSize == 3) ? 2 : 1;
if(BlockSize == 2) {
    MAXSHIFT = MAXHASH = 0x10000;
    HASH = (uchar*)malloc(MAXHASH*sizeof(char));
    SHIFT = (uchar*)malloc(MAXSHIFT*sizeof(char));
}
else
{
    MAXSHIFT = MAXHASH = 0x1000000;
    HASH = (uchar*)malloc(MAXHASH*sizeof(char));
    SHIFT = (uchar*)malloc(MAXSHIFT*sizeof(char));
}
for(i=0; i< MAXSHIFT; i++) SHIFT[i] = LSP - BlockSize + 1;
for(i=0; i< MAXHASH; i++) HASH[i] = 0;
for(i=1; i<=PatNum; i++) prep_hash_shift(&Pattern[i][1]);
accumulate();
for(i=1; i<=PatNum; i++) prep_hash_shift2(i, Pattern[i][0],
&Pattern[i][1]);
for(i=1; i<=PatNumFNP; i++) prep_enumerate(i, PatFNP[i][0],
&PatFNP[i][1]);
```

Figure 3.6 FWM Initialization

After initializing, we only handle the patterns whose lengths are greater than block-size and fill the corresponding entry with individual shift value. If a look-ahead character is considered, we need to compute the

hash key with any characters and the prefix of a pattern. When a collision happens, we always fill the smaller one to avoid false matching. Then, we start to handle the patterns whose lengths are less than block-size. For these patterns, we need to enumerate all the possible. If the block-size is equal to 2 and we have a pattern ‘a’, we need to turn on the bit indicating the match as long as the index of the entry is corresponding to ‘*a’ or ‘a*’. Preprocessing pseudo code is shown in Figure 3.7.

Preprocess:

```
if(FLAG_LOOK_AHEAD)   m = (BlockSize == 3) ? LSP-2 : LSP-1;
else m = LSP;
for(i=m-1; i>=BlockSize-1; i--) {
      hash = (int)pat[i];
      hash = (hash << Hbits) + (int)pat[i-1];
      if( BlockSize == 3 ) hash = (hash << Hbits) + (int)pat[i-2];
      if(SHIFT[hash] > m-i-1) SHIFT[hash] = m-i-1;
}
if (FLAG_LOOK_AHEAD) {
      if(BlockSize == 2)
            for(i=0; i<=255; i++) {
                  hash = (int)pat[0];
                  hash = (hash << Hbits) + i;
                  if(SHIFT[hash] > m-1) SHIFT[hash] = m-1;
            }
      else /* BlockSize == 3 */ {
            same as BlockSize == 2, expand the * and fill the table
      }
}
i = m - 1;
hash = (int)pat[i];
hash = (hash << Hbits) + (int)pat[i-1];
if( BlockSize == 3 ) hash = (hash << Hbits) + (int)pat[i-2];
HASH[hash]++;
```

Figure 3.7 FWM Preprocessing

After preprocessing, we start to scan the text. While scanning, we compute the index and get the shift value. Concurrently, we check the

matching bit to determine whether a matching occurs. If the shift value is not zero, we shift and continue to scan. Otherwise, there may be a match to occur. We check the prefix. If there is still a match, we traverse the list to match a pattern. Scanning stage pseudo code is shown in Figure 3.8

Scanning Stage:

```
while(text <= textend)
{
    hash = *text;
    hash = (hash << Hbits) + *(text-1);
    if( BlockSize == 3 ) hash = (hash << Hbits) + *(text-2);
    shift = SHIFT[hash];
    if((shift & 0x80) != 0) {
        num_match++; shift &= 0x7F;
    }
    if( shift == 0 ) {
        hash2 = (*(text-m1) << Hbits) + *(text-m1+1);
        p = HASH[hash];
        p_end = HASH[hash+1];
        while(p++ < p_end) {
            if (prefix is not match) continue;
            px = PatPtr[p];
            qx = text - m1;
            for(i=0; i<patternlen; i++) {
                if(*px == *qx) {px++; qx++;}
                else break;
            }
            if(i == plen)    MATCHED++;
        }/* end of while p++ < p_end */
        if(!MATCHED) shift = 1;
        else {
            MATCHED = 0;
            shift = (m1-1) > 0 ? (m1-1) : 1;
        }
    }/* end of if shift == 0 */
    shift = shift > BlockSize ? BlockSize : shift;
    text += shift;
}/* end of while text <= textend */
```

Figure 3.8 FWM Scanning Stage

4. Experiments

4.1 Environment

To evaluate the performance of our algorithm, we compare our algorithms to E^2XB on a Linux RedHat 7.3 workstation (Pentium 4 CPU 2.0GHz) with 512MB memory, 20KB L1 cache, and 512KB L2 cache. We perform two different kinds of experiments with the same input. One is to simulate the behavior on network processor. The other is to run normally. For simulating the behavior on network processor, we turn off the L1, L2 cache.

Each environment, we evaluate the performance by comparing the time which is measured by running the simulation of individual algorithm. In our simulations, we get the real traffic from DEFCON [17] to be the input, and the signatures from snort [18] to be the patterns.

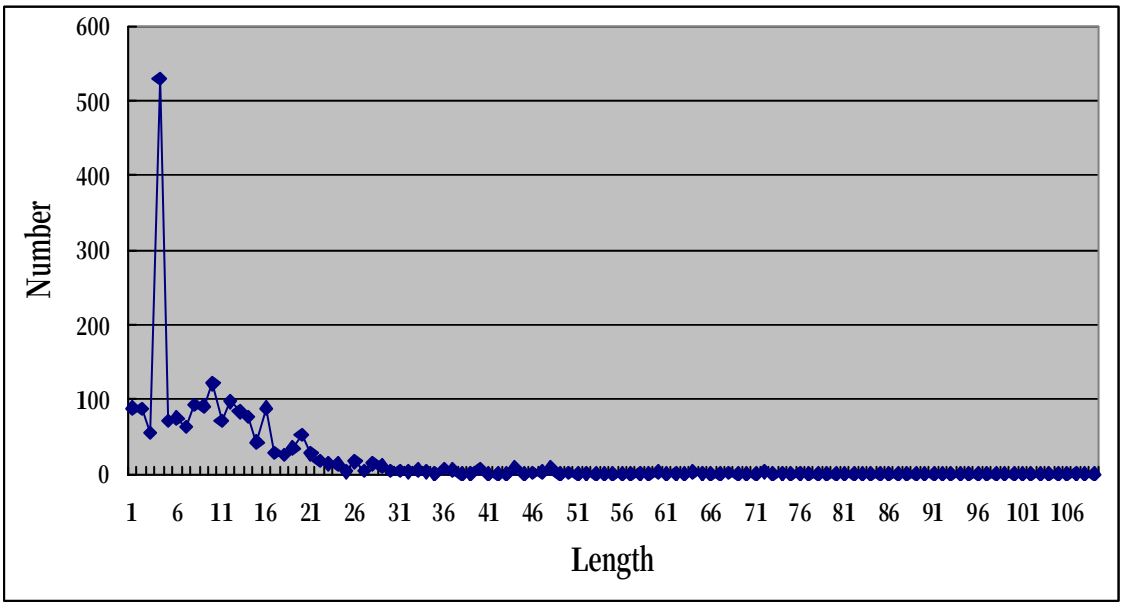


Figure 4.1 Distribution of the signature length

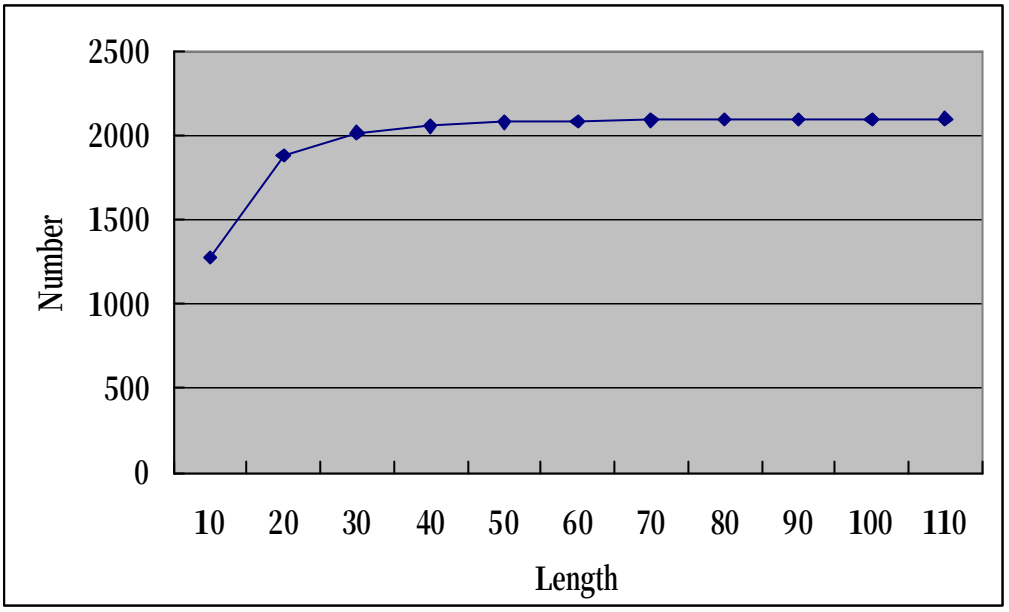


Figure 4.2 Accumulation of the pattern length

Figure 4.1 and Figure 4.2 show the distribution of the patterns. We get the snort rules and transform these rules into our own patterns. We use 2100 patterns to run the simulation. As Figure 4.1, we find that the pattern whose length is equal to 4 is the most frequent. Moreover, there are 177 patterns whose lengths are equal to or less than 2. There are 233 patterns whose lengths are equal to or less than 3. These all affect the performance of the algorithm. The text file we used for all experiments was a collection of attacking traffic from DEFCON [17]. The file size of the traffic is about 900Mbytes, and contains total 575635 packets.

4.2 Performance Evaluation

4.2.1 Network Processor Simulation

To verify the effectiveness of the proposed FWM, we present several experiments in which our algorithm is compared to several algorithms and the effects of block-size, LSP, look-ahead and the

number of rules on the performance are also evaluated.

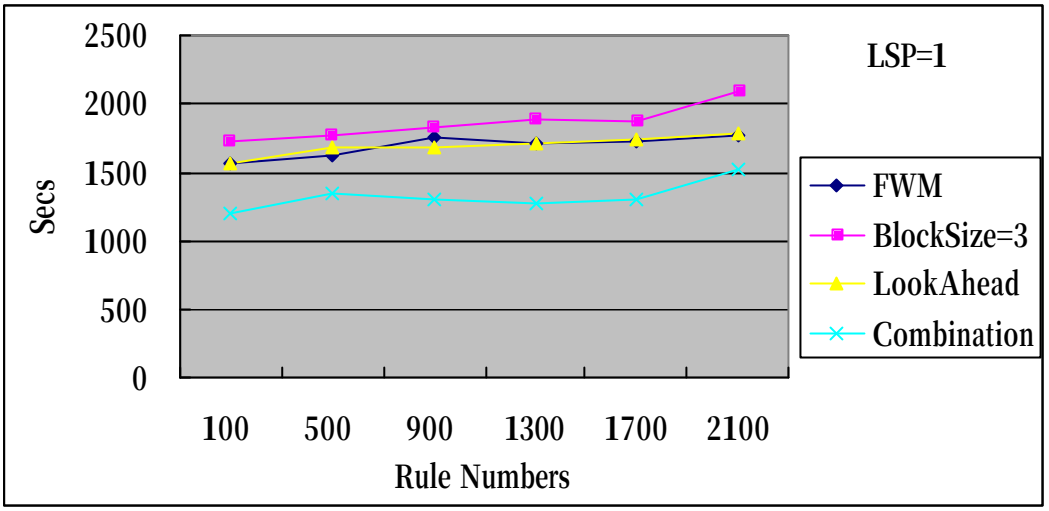


Figure 4.3 Completion time comparison using LSP=1

Figure 4.3 reveals that the processing time different processing techniques require. FWM algorithm, which is WuM algorithm with occurrence table, takes about 1700 seconds. Typical WuM algorithm can't run on LSP=1. The running time of enlarging block-size to 3, labeled BlockSize=3, is greater than FWM. Although enlarging block-size increases the number of matching to the smaller patterns. But it also makes the maximum shift value decrease. As shown in Figure 4.1, most patterns whose lengths are greater than 3. Therefore, FWM that use the block-size = 2 has better performance than block-size = 3.

FWM with a look-ahead character, labeled LookAhead, has the similar performance to the FWM. Although the LookAhead enlarges the LSP, it may probably result in more collisions. It depends on what the patterns we have. Another reason having lower performance is occurrence table. As mentioned in section 3.2.1, the performance of LookAhead is the reverse proportion to the LSP. But patterns whose lengths are smaller than block-size are determined by the occurrence table. LookAhead works to the patterns whose lengths are greater than 2. So the rate of

performance improvement of LookAhead is $\frac{(4-2+1)-(3-2+1)}{(3-2+1)}=\frac{1}{2}=50\%$. (Because of the occurrence table, the original LSP is equal to 3).

We combine the BlockSize=3 and LookAhead, labeled Combination, and have the best performance. As mentioned in section 3.2.1, when we enlarge the block-size, we will have two look-ahead characters. The combination of two techniques confer many advantages. It can increase the maximum shift value and let more patterns be matched in block-size. Having two look-ahead characters runs better than one look-ahead character. LookAhead with two look-ahead characters may also have collisions. But because of the bigger block-size, it also increases the matching probability. Moreover, it changes the LSP into (LSP+2), then the performance improvement will be $\frac{(6-3+1)-(4-3+1)}{(4-3+1)}=\frac{2}{2}=100\%$.

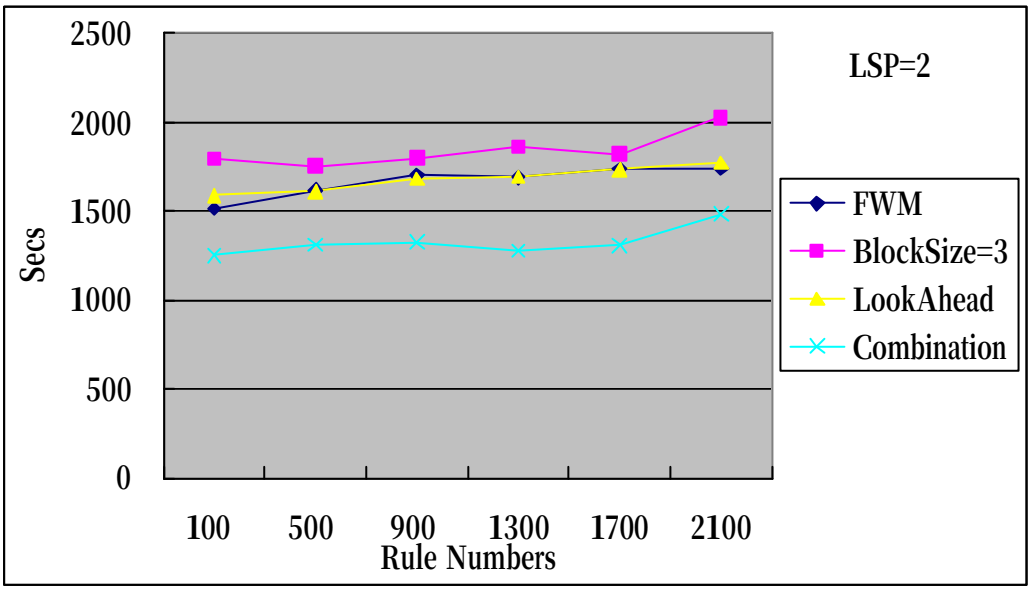


Figure 4.4 Completion time comparison using LSP=2

Figure 4.4 shows the processing time with LSP=2. We have similar result to the simulation with LSP=1.

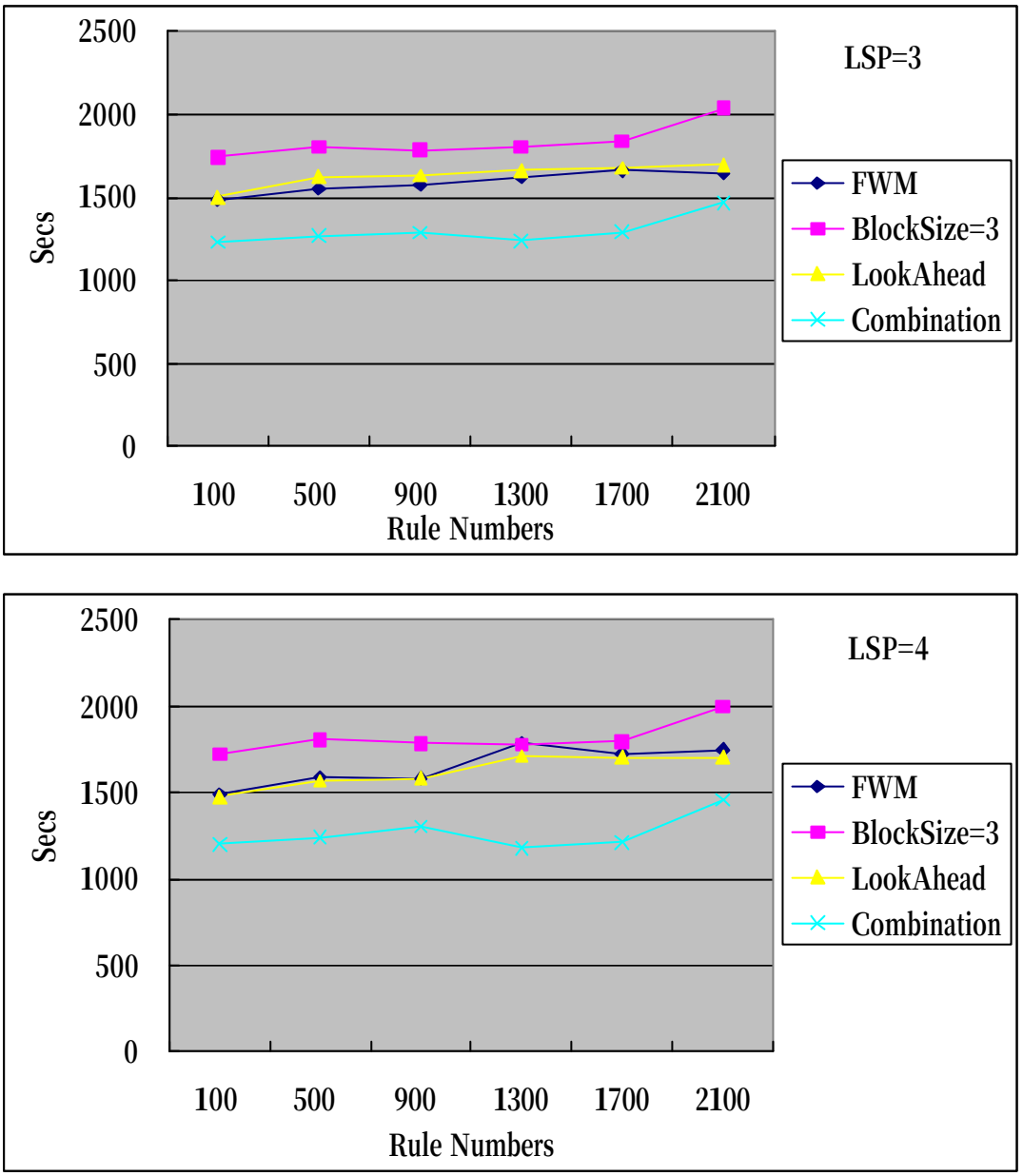


Figure 4.5 Completion time comparison using LSP=3, 4

Figure 4.5 shows the processing time with LSP=3 and LSP=4. In this experiment, FWM is equivalent to WuM algorithm. Because we only use the occurrence table to match the patterns whose lengths are equal to or less than the block-size. And the LSP is equal to 3 in this experiment. Therefore, FWM algorithm takes less time while LSP=3 than LSP=2 because of no collisions.

Figure 4.6 shows the processing time of FWM algorithm under different LSP. We find that each of the processing time of FWM algorithms seems to be very close.

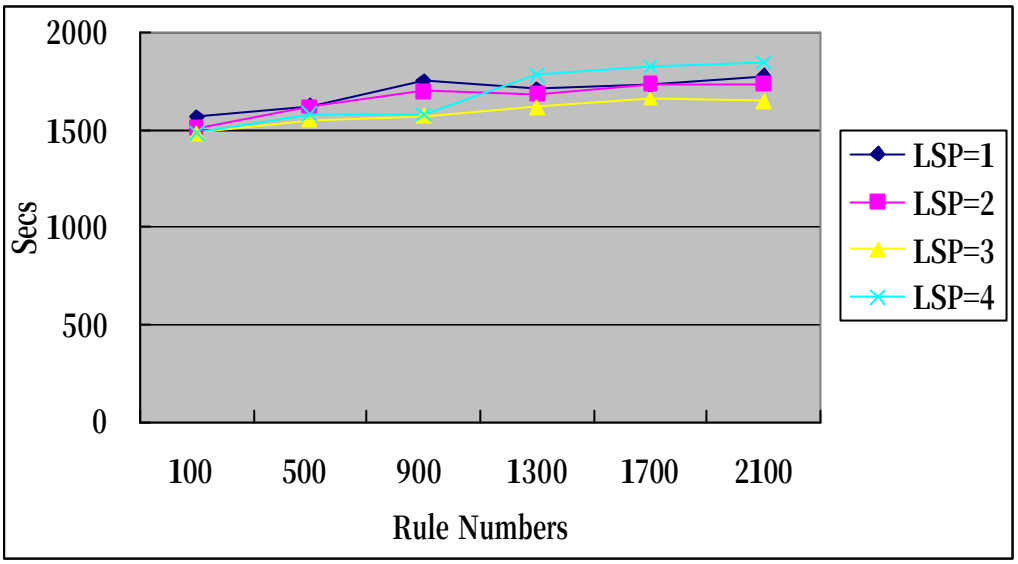


Figure 4.6 Completion time comparison on various LSP

4.2.2 CPU-Based Simulation

In section 4.2.1, we run the simulation to simulate the behavior of network processor. As mentioned in section 3.2.1, whether FWM algorithm improve the performance rely on occurrence table. If the character set is too big, we will need mass of memory spaces to store the table. In network processor architecture, we can build occurrence table directly in external memory because there is no L1-L2 cache in it. But in general purpose CPU architecture, if the table size is too big, it may not be stored in L1-L2 cache and will have poor performance. We run the simulation on x86 architecture and compare it to the simulation based on network processor.

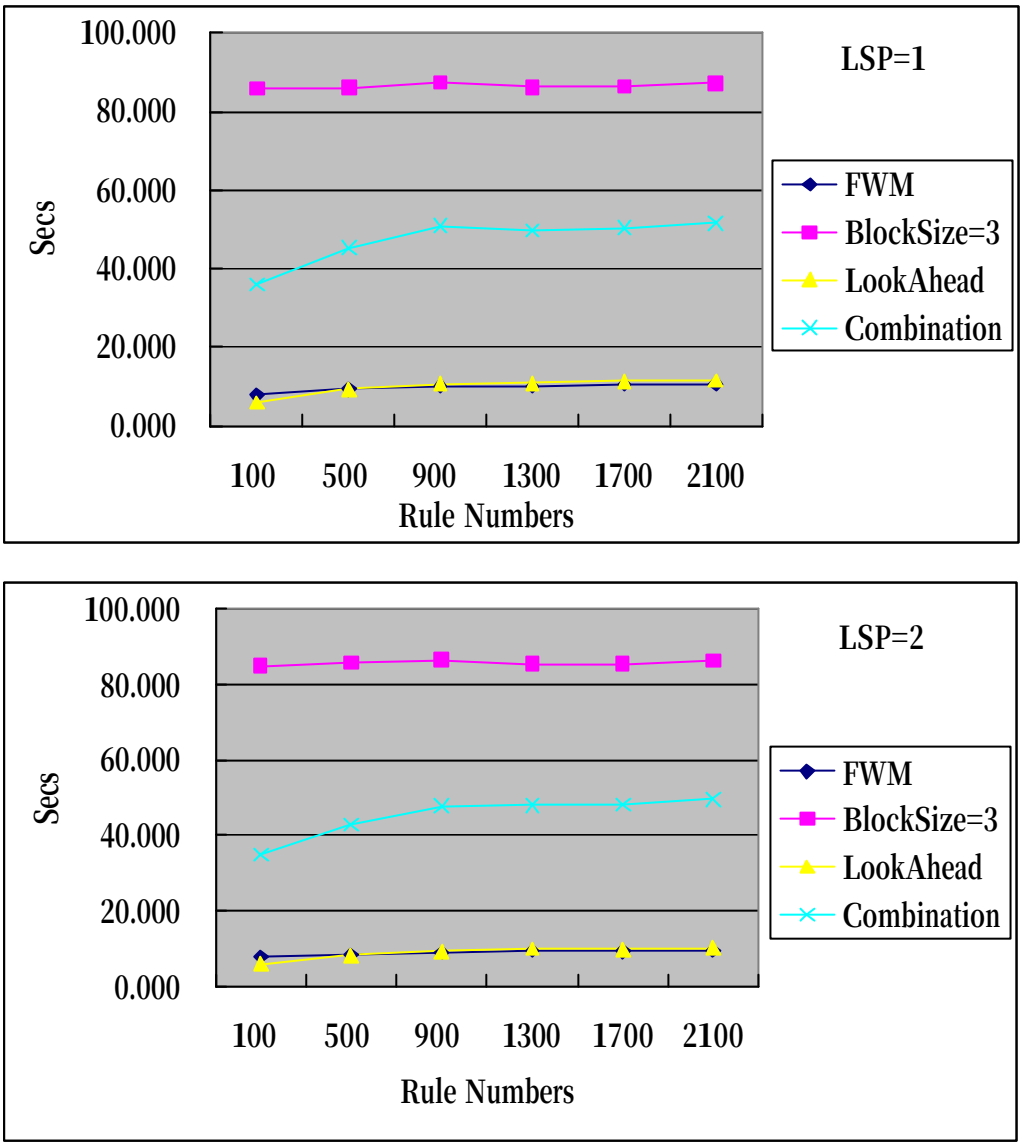


Figure 4.7 Completion time comparison using LSP=1, 2 with cache

Figure 4.7 shows the processing time using LSP=1 and LSP=2 with cache. Obviously, we can know that if the block-size is equal to 3, it must have poor performance. As mentioned in section 4.1, the size of L1 cache is 20Kbytes, and 512Kbytes to L2 cache. If the block-size is equal to 2, the size of shift table is $2^{8\cdot2} = 2^{16} = 64Kbytes$ and the size is less than L2 cache. Thus, it has the better performance. If the block-size is equal to 3, the size of shift table is $2^{24} = 16Mbytes$ and the size is greater than 512Kbytes. Thus it has poor performance.

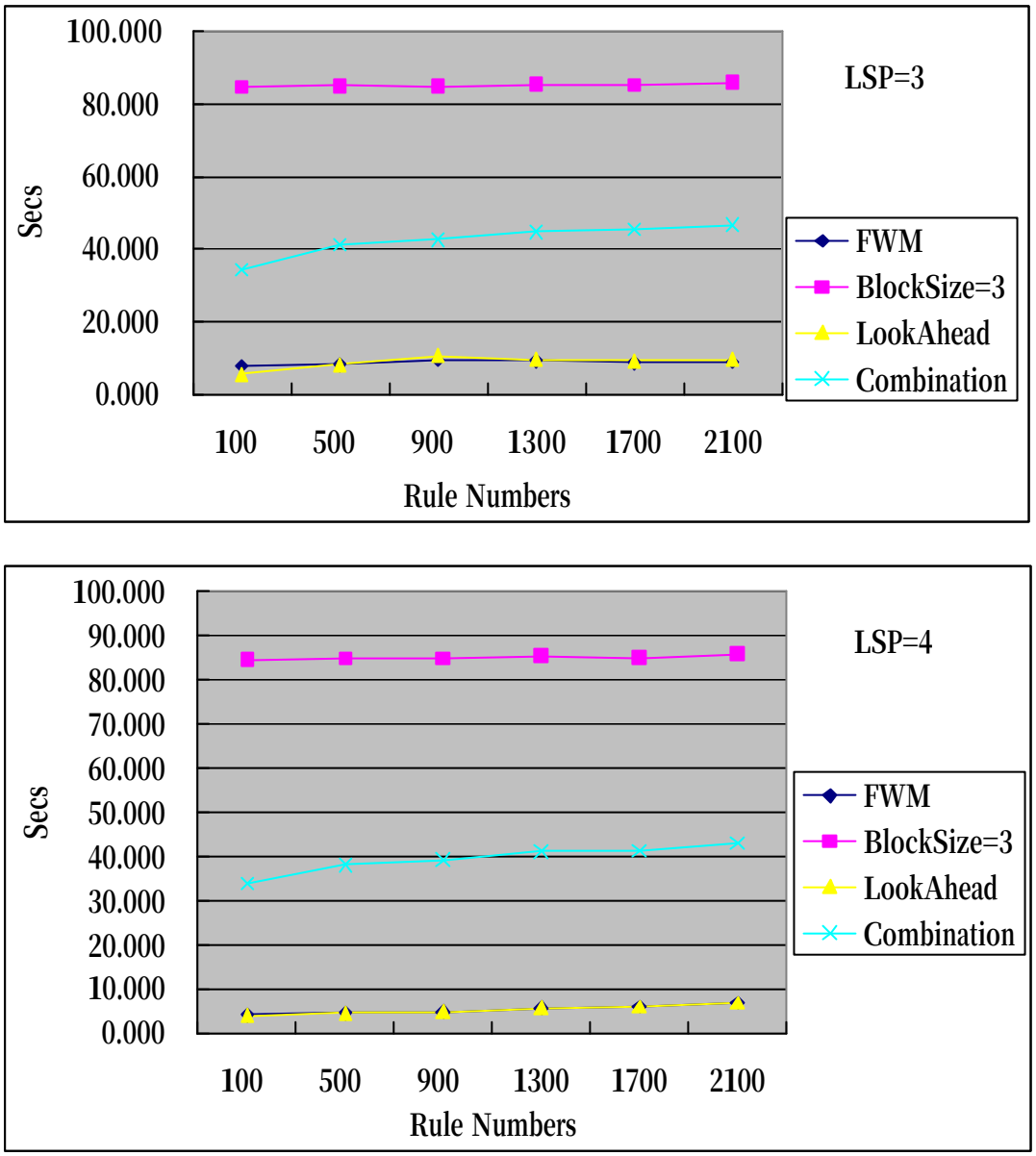


Figure 4.8 Completion time comparison using LSP=3, 4 with cache

Figure 4.8 shows the processing time using LSP=3 and LSP=4 with cache. The running time of FWM is less than that of others in Figure 4.6. FWM algorithm in Figure 4.8 is equivalent to typical WuM algorithm because the LSP is greater than block-size. We know WuM algorithm have better performance while having bigger LSP. Similarly, if the block-size is equal to 3, it must have poor performance.

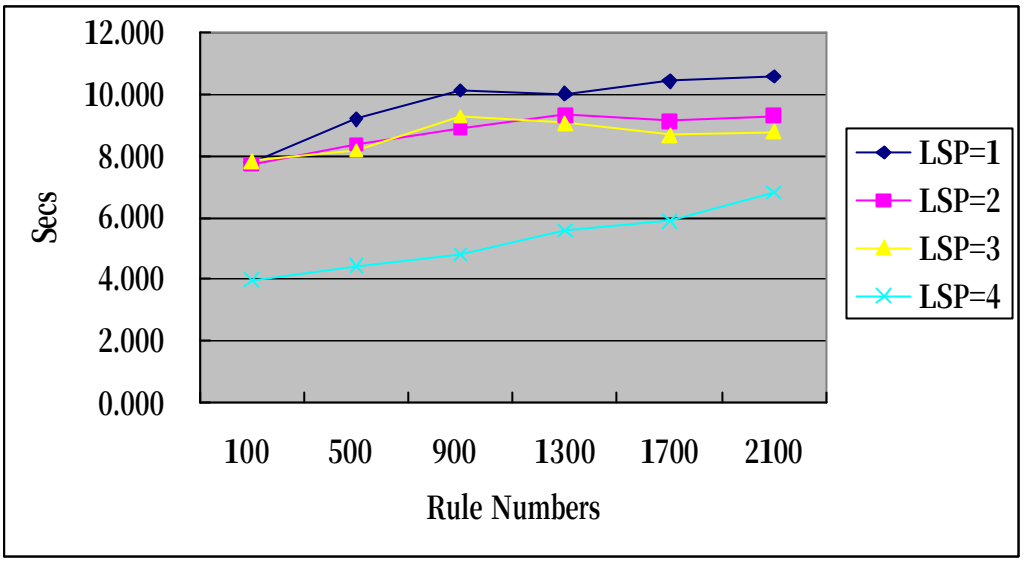


Figure 4.9 Completion time comparison on various LSP with cache

Figure 4.9 shows the processing time of FWM algorithm under different LSP. When LSP=4, it can shift the maximum characters and it doesn't have any collisions by occurrence table, it has the best performance.

4.2.2 Comparison

We compare our algorithm against E^2XB [9], The paper "Performance Analysis of Content Matching Intrusion Detection Systems" [11] runs a simulation to compare the performance between WuM and E^2XB. According to the paper, E^2XB algorithm has better performance than does WuM algorithm. We run the simulation shown as Figure 4.10.

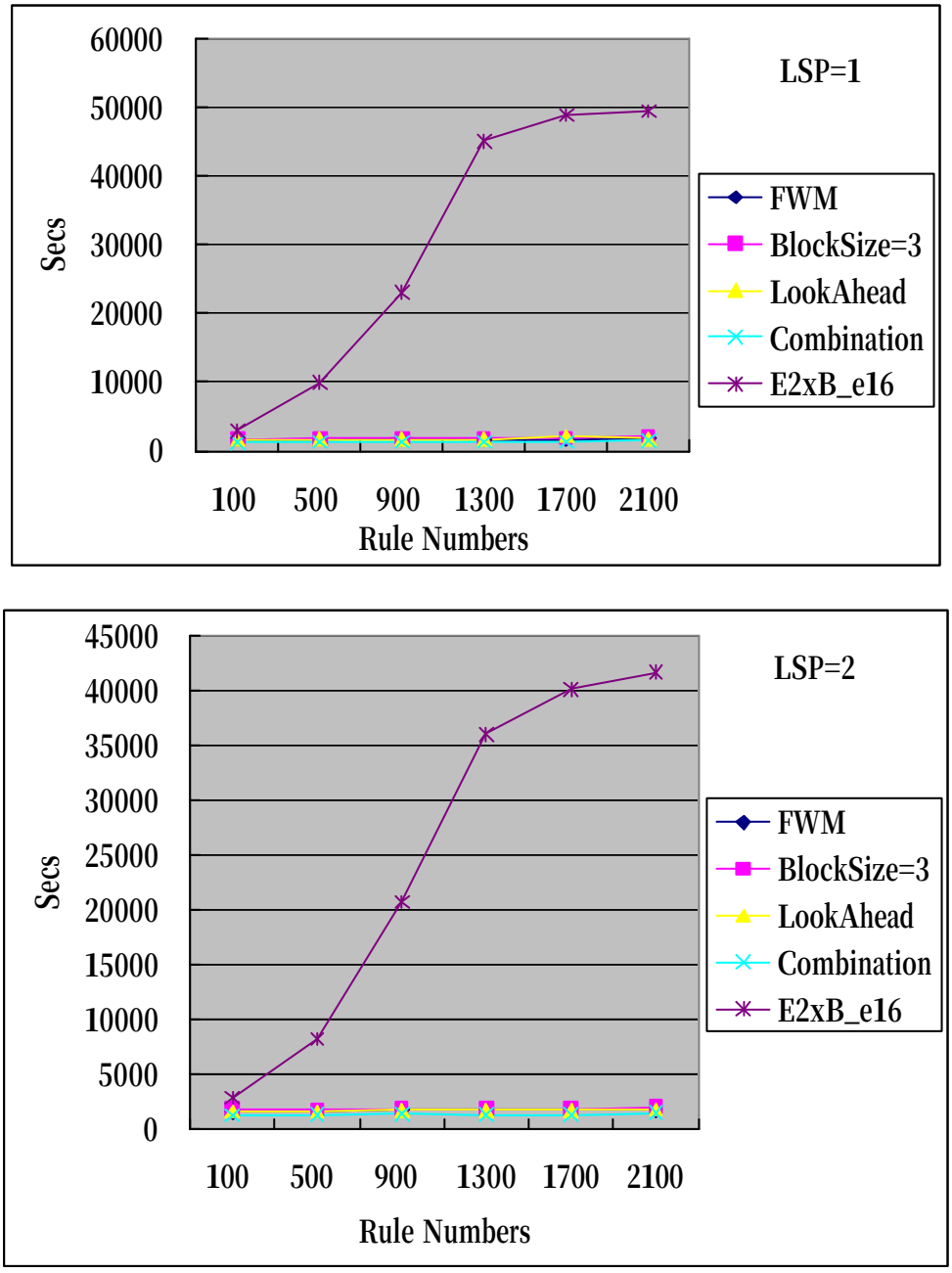


Figure 4.10 Comparing to E^2XB

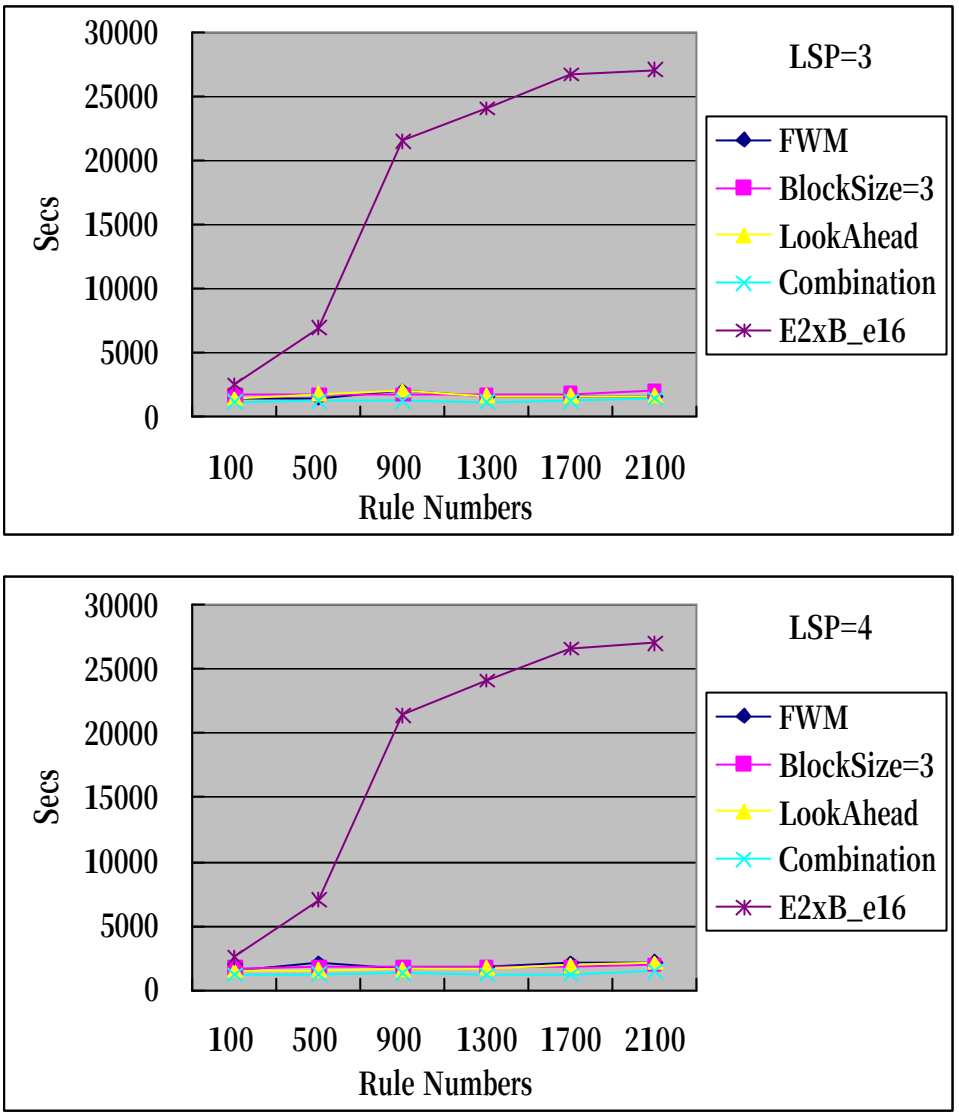


Figure 4.11 Comparing to E^2XB

According to reference [11], the performance of pattern matching algorithm depends on the number of rules. Both WuM and E^2XB algorithms, the performance of each is proportion to the number of rules. But we get almost constant time in our experiment. According to the principle of WuM, all patterns are processed to compute the shift value. The size of shift table is insensitive to the number of rules. In other words, no matter how many patterns, the size of shift table is fixed. This implies the running time is irrelevant to the number of rules. E^2XB algorithm is a searching algorithm for single pattern. If it would like to process

multi-pattern, it must compare the text with all patterns sequentially. Therefore, E^2XB algorithm must be sensitive to the number of rules.

Moreover, E^2XB is also sensitive to LSP. Figure 4.10 and Figure 4.11 show the relation under different LSP. When LSP increasing, the running time of E^2XB decreases. E^2XB also rely on BM algorithm to check whether matching or not. Thus, bigger LSP will increase the performance. This is why E^2XB takes the least time while LSP=4 than others.

5. Conclusion

This work shows how an improved pattern matching algorithm can do for network devices based on inspecting incoming packets. We know that while some algorithms such as BM, WuM move up the searching speed by shifting to reduce the comparison, these algorithms can't run when LSP is equal to 1. We proposed our algorithm that could further enhance the performance of WuM algorithm.

We exploit the feature of NPU architecture which is no L1-L2 cache. Thus, we can use occurrence table to match smaller patterns. Moreover, we improve the performance of WuM by increasing the length of the shortest pattern and compare the performance with other search routines: Big Block-size, LookAhead, and Combination.

Our experiments so far have led to several observations. First, we have found that WuM algorithm is insensitive to the number of rules. That's a marked difference from the finding listed in the reference [11]. According to our experiments, WuM algorithm spent almost the same time with different LSP and the number of rules. Searching algorithm for single pattern may be sensitive to the number of rules because it must compare the text to all patterns sequentially.

Second, the size of occurrence table is limited. We can't build an occurrence table whose size is greater than the size of L1-L2 cache. If the character set is large, we may not use occurrence table unless there is no L1-L2 cache.

Third, we can increase the length of the shortest pattern to improve the performance by shifting more characters. But using look-ahead may result in collisions and make no effect. This is because we need to handle * character. When we parse a * character, we need to enumerate all the possible once and use a loop to expand * character.

Fourth, we set block-size to 3 to let more patterns be matched in block-size, and take two look-ahead characters to increase the length of the shortest pattern. Thus, we can shift more characters.

Fifth, we speed up the performance by 15%-20% in terms of the time spent in simulations. As illustrated in Table 1.1, the better performance of pattern matching will also improve the throughput in NIDS.

Reference

[1] Boyer R. S., and J. S. Moore, "A Fast String Searching Algorithm" Communications of the ACM20 (October 977), pp. 762-772

[2] Aho, A. V., and M. J. Corasick, "Efficient String Matching: an aid to bibliographic search" Communications of the ACM 18 (June 1975), pp. 333-340

[3] Commentz-Walter, B, "A String Matching Algorithm Fast on The Average" Proc. 6th International Colloquium on Automata, Languages, and Programming (1979), pp. 118-132.

[4] U. Manber and S. Wu, "GLIMPSE: A Tool to Search Through Entire File System" Usenix Winter 1994 Technical Conference (January 1994), pp. 23-32.

[5] U. Manber, "Finding Similar Files in a Large File System" Usenix Winter 1994 Technical Conference, pp. 1-10.

[6] Sun Wu and Udi Manber, "A Fast Algorithm for Multi-Pattern Searching" (May 1994), Tech. Rep. TR94-17 Department of Computer Science, University of Arizona, May 1994.

[7] Wu S., and U. Manber, "Fast Test Searching Allowing Errors" Communications of the ACM 35 (October 1992), pp. 83-91

[8] Wu S., and U. Manber, "Agrep – A Fast Approximate Pattern-Matching Tool," Usenix Winter 1992 Technical Conference, (January 1992), pp. 153-162.

[9] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis. "E2xB: A domainspecific string matching algorithm for intrusion detection" Proceedings of the 18th IFIP International Information Security Conference, (May 2003).

[10] E. P. Markatos, S. Antonatos, M. Polychronakis and K. G Anagnostakis. "ExB: Exclusion-based signature matching for intrusion

detection" Proceedings of the IASTED International Conference on Communications and Computer Networks, Cambridge, USA, (November 2002), pp. 146-152.

[11] S. Antonatos, K. G. Anagnostakisy, E. P. Markatos, and M. Polychronakis, "Performance Analysis of Content Matching Intrusion Detection Systems", Proceedings of the International Symposium on Applications and the Internet (SAINT), 2004.

[12] C. Jason Coit, Stuart Staniford, and Joseph McAlerney, "Towards faster pattern matching for intrusion detection or exceeding the speed of snort", Proceedings of the 2nd DARPA Information Survivability Conference and Exposition, (June 2001).

[13] Sun Kim and Yanggon Km, "A fast multiple string-pattern matching algorithm", Proceedings of the 17th AoM/IAoM International Conference on Computer Science, (May 1999).

[14] P. Paulin, F. Karim, P. Bromley, "Network Processors: A perspective on Market Requirements, Processor Architectures and Embedded S/W Tools", Proceedings of the DATE 2001 on Design, automation and test in Europe, IEEE Press, 2001, pp. 420-429.

[15] Mike Fisk and George Varghese, "Fast Content-Based Packet Handling for Intrusion Detection", UCSD Technical Report CS200-067D, (May 2001).

[16] Nen-Fu Huang and Rong-Tai Liu "A Fast String Matching Algorithm for Network Processor-Based Intrusion Detection System", ACM TECS special issue on Embedded Systems and Security, (Febrary 2003).

[17] DEFCON, http://www.shmoo.com/cctf/

[18] Snort.org, http://www.snort.org/

[19] Vitesse.com, http://www.vitesse.com