TECHNICAL REPORT 04-02

Correctness Proof of the SBT method

Yutaka Sugawara and Kei Hiraki

October 20, 2004

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TITLE

Correctness Proof of the SBT method

AUTHORS

Yutaka Sugawara, Mary Inaba, and Kei Hiraki

KEY WORDS AND PHRASES

string matching, FPGA, SBT, correctness proof

ABSTRACT

In this paper, we prove the correctness of the SBT method, which we proposed in a previous paper [1]. SBT is a string matching method optimized for high-speed multi-stream packet scanning on FPGA. The SBT method is capable of lightweight switching between TCP streams, and enables easy implementation of multi-stream scanners. In addition, we achieved over 10Gbps string matching bandwidth using a Xilinx XC2V6000 FPGA. In this paper, we present a correctness proof of the SBT method, which we could not presented in the previous paper [1] due to the space limitation.

REPORT DATE October 20, 2004	**WRITTEN LANGUAGE** English
TOTAL NO. OF PAGES 11	**NO. OF REFERENCES** 12

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Correctness Proof of the SBT method

Yutaka Sugawara, Mary Inaba, and Kei Hiraki

1 Introduction

Recent technology has realized fast networks such as 10Gbit Ethernet and OC-768. To take advantage of such high-speed networks, it is necessary to accelerate network applications. Part of the applications requires intensive packet payload scanning. For example, in network intrusion detection systems (NIDS), payload scanning to find suspicious patterns occupies the major part of the computation time[2]. Other examples include Content-based billing, SPAM filtering, and HTTP request distribution. To accelerate them, fast string matching is necessary. In the matching, a set of byte strings (*rules*) is statically given, and occurrences of the strings in input packet streams are checked.

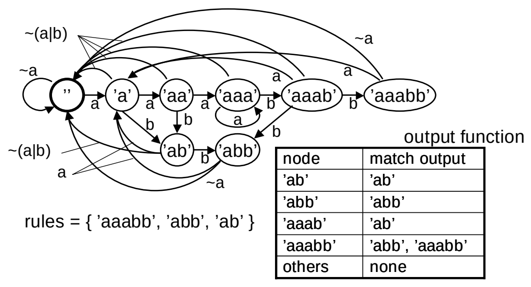
When matching speed alone is important, hardware implementation is better than software implementation. However, it is also necessary to update pattern rules. For example, in NIDS, rules are frequently updated to cope with the patterns found in new intrusion methods. Therefore, we cannot hard-wire rules because the rule update is impossible without replacing the hardware. FPGA based implementation is a solution for string matching in hardware while allowing the rules to be updated. When FPGAs are used, the rules can be changed by reconfiguration. There are studies of string matching using FPGAs such as comparator based methods[3][4], CAM based methods[5], non-deterministic automaton(NFA) based methods[6][7][8], deterministic finite automaton(DFA) based methods[9], Bloom Filter based methods[10], and Knuth-Moris-Pratt(KMP) algorithm based methods[11]. They enabled high-speed string matching while allowing the rules to be changed. For a high throughput under a limited clock speed, it is important to process multiple input characters at once[3][4][8].

In TCP, a communication data stream is split into packets. Therefore, a target pattern may span multiple packets. Such fragmented pattern cannot be discovered by a per-packet scan. This is a serious problem in applications such as NIDS that require a complete scan. To avoid the problem, it is necessary to scan TCP streams. However, existing FPGA based methods cannot be used for scanning multiple streams because of the difficulty in switching between streams.

In [1], we proposed a string matching method called suffix based traversing (SBT). SBT is an extension of the Aho-Corasick algorithm[12], which uses table lookup in state transition. Since a small number of state bits are used in the SBT method, lightweight stream switching is enabled. The main points of extension for SBT are (1)processing multiple input bytes at once, and (2)table size reduction. When the Aho-Corasick algorithm is naively extended to process multiple bytes, a large lookup table is necessary. We solved this problem by reducing the number of table input patterns. As a result, the large table is converted into smaller tables and over 10Gbps scanning is enabled.

In the previous paper [1], we did not present the correctness proof of the SBT method because of the space limitation, though it is mandatory for a complete algorithm description. In this paper, we present the correctness proof.

In Section 2 we describe the SBT method. Section 4 presents the correctness proof of the SBT method. We conclude this paper in Section 5.



node	match output
'ab'	'ab'
'abb'	'abb'
'aaab'	'ab'
'aaabb'	'abb', 'aaabb'
others	none

Figure 1: An example FSM of Aho-Corasick algorithm

2 Outline of SBT Method

2.1 Aho-Corasick Algorithm

In Aho-Corasick algorithm[12], a trie is built that contains all the rule strings. Then, the string matching is performed using a state machine whose state is a node position of the trie. The initial state is the root node of the trie. Each time a character of the string is input, the state transits from the current node to the next node through the transition edge corresponds to the input character[1]. The current state node corresponds to the longest suffix of the input string up to this time.

The output function is defined for each state node. The output function indicates what rule strings are matched when each state node is reached. Note that the output format of the match results depends on the application[2]. We must change the output function according to the output format. In this paper, we discuss only the state transition, and we do not discuss the matching result output. Figure 1 is an example FSM used in Aho-Corasick algorithm.

2.2 Optimization of FSM Based Method

In the description of this paper, all the strings consist of 1 octet (8 bit) characters. When string matching is performed using an FSM based method with state transition table lookup, we must optimize the implementation in the following respects:

- Parallel processing of multiple input characters.
- Memory size reduction of the state transition tables.

The former is important to realize high matching throughput under a limited clock speed. In an FSM based method, when the state transition table lookup is performed in an sequential manner, it is impossible to process multiple characters in parallel. To process multiple characters in parallel, it is necessary to change the implementation of state lookup mechanism.

The later is necessary for implementing the state transition tables using FPGA on-chip memories. Since off-chip memories are slower than off-chip ones, high performance cannot be achieved by off-chip memory implementations. In addition, when multiple characters are processed in parallel, off-chip implementations are very difficult because many independent memory chips are necessary.

[1]In Aho-Corasick algorithm, it is possible not to enumerate all the transition edges by using the failure function. However, we use an implementation without the failure function because it is suitable for hardware implementation since the processing amount of each cycle is constant.

[2]For example, in an application, just the longest matching rule string is output.

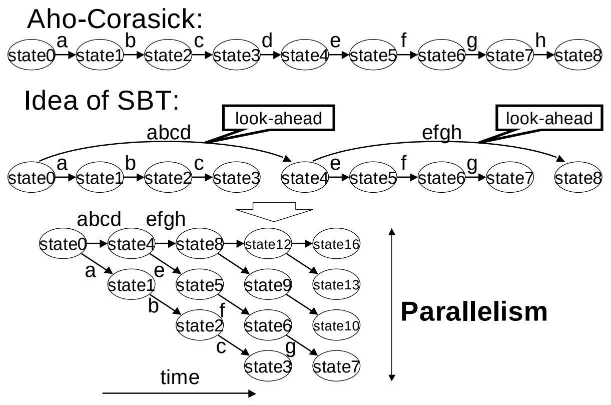


Figure 2: Parallel processing of multiple characters using look-ahead tables

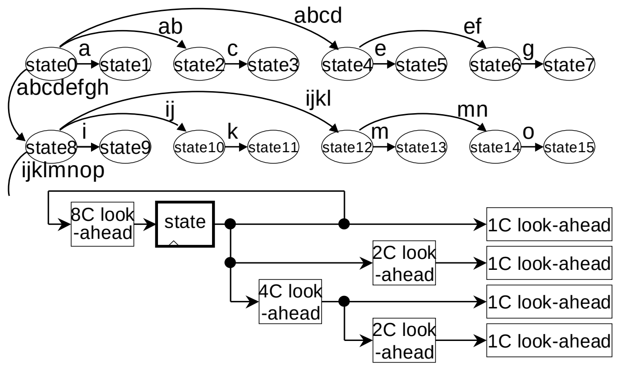


Figure 3: Tree-wise state look-ahead

2.3 Parallel Processing of Characters using Look-ahead Tables

In SBT method, **state look-ahead table** is used to get a state after a specified number of characters. We call a state look-ahead table that tells a state after n characters as n-characters look-ahead table. The n-characters look-ahead table receives the current state and the succeeding n characters, and returns the state after the n characters[3]. Using the look-ahead tables, multiple state transitions can be calculated in parallel. For example, in Figure 2, 4 state transitions are calculated in parallel using a 4-characters look-ahead table.

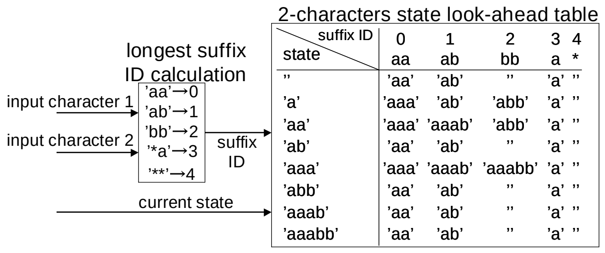
In an implementation like Figure 2, the calculation latency is $O(w)$ when w characters are processed in parallel. To reduce the latency to $O(\log w)$, a tree-wise state look-ahead is performed in SBT method, as shown in Figure 3. Though the w can be an positive integer that is not a power of 2, in this paper, we limit the w to a power of 2 for simplicity.

2.4 Memory Usage Reduction

2.4.1 Input pattern classification

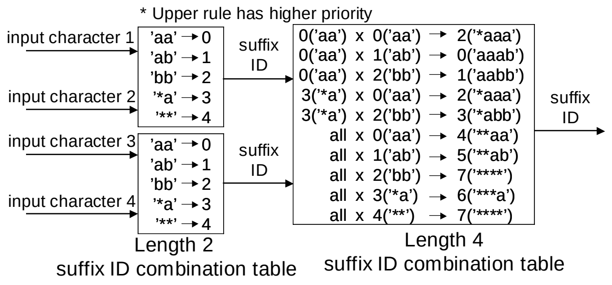
One naive implementation of n-characters state look-ahead table is to lookup a table of 2^{b+8w} entries using a $b + 8w$ bits value that is the concatenation of the current state ID and the w

[3]1-characters look-ahead table is state transition table itself.



state \ suffix ID	0 aa	1 ab	2 bb	3 a	4 *
''	'aa'	'ab'	''	'a'	''
'a'	'aaa'	'ab'	'abb'	'a'	''
'aa'	'aaa'	'aaab'	'abb'	'a'	''
'ab'	'aa'	'ab'	''	'a'	''
'aaa'	'aaa'	'aaab'	'aaabb'	'a'	''
'abb'	'aa'	'ab'	''	'a'	''
'aaab'	'aa'	'ab'	''	'a'	''
'aaabb'	'aa'	'ab'	''	'a'	''

Figure 4: State look-ahead table lookup using suffix pattern ID



0('aa')	x	0('aa')	→	2('*aaa')
0('aa')	x	1('ab')	→	0('aaab')
0('aa')	x	2('bb')	→	1('aabb')
3('*a')	x	0('aa')	→	2('*aaa')
3('*a')	x	2('bb')	→	3('*abb')
all	x	0('aa')	→	4('**aa')
all	x	1('ab')	→	5('**ab')
all	x	2('bb')	→	7('****')
all	x	3('*a')	→	6('***a')
all	x	4('**')	→	7('****')

Figure 5: Hierarchical calculation of suffix ID

input characters (b is the bit width of the state ID). However, the drawback of this method is that the memory usage increases rapidly as the w increases. For example, when $w = 8$, the number of the table entries is more than 2^{64}. Therefore, the implementation is difficult.

However, lookup tables have a redundancy that it returns a non-initial state only when the input characters have limited suffix patterns. We can reduce the memory size using the redundancy. For example, in the 2-characters lookup table of the rules shown in Figure 1, though the total number of 2-character input patterns is 2^{16}, the table returns a non-initial state only when a suffix of the input character is `'aa'`, `'ab'`, `'bb'`, or `'a'`.

Therefore, we can reduce the size of the lookup tables by classifying the input character patterns based on its suffix pattern and looking up the look-ahead table using the **suffix ID**, as shown in Figure 4. In the example of Figure 4, the number of the columns is reduced from 2^{16} to 5.

The suffix ID calculation is performed in an hierarchical manner using **suffix ID combination tables**, as shown in Figure 5. First, IDs are assigned to the necessary suffix patterns of length k strings for $k = 2, 4, 8, \ldots, w$. A k-characters suffix ID combination table receives two suffix IDs of length $k/2$ strings, and returns the suffix ID of the length k string that is the concatenation of the two $k/2$ length strings. Using the hierarchical suffix ID calculation, the table size is reduced.

2.4.2 Table size reduction using indirect pointers

Even when the number of columns of the look-ahead tables are reduced using suffix IDs, the look-ahead tables have a redundancy that each row has a value that appears frequently. For example, in Figure 4, `'aa'` appears frequently in the column 0(`'aa'`), and `''` appears frequently in the column 2(`'bb'`). The suffix ID combination tables have a similar redundancy. We can reduce the table size using the redundancy, as shown in Figure 6. First, the **default value**

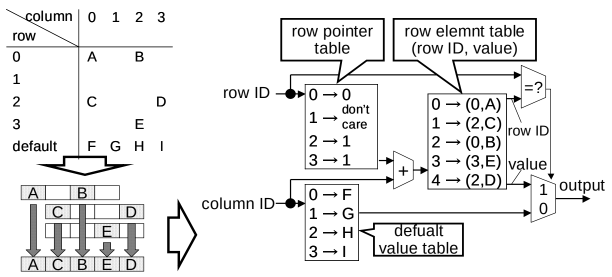


Figure 6: Table size reduction using row pointers

table is built that holds the value that appears most frequently for each row. Next, the original table is decomposed into rows, and the column elements that differ from the default table values are stored in the **row element table**. The owner row ID is stored for each element in the row element table. Finally, the row pointer table is built that holds the starting index of each row in the row element table. Each empty element of the upside-left table in Figure 6 is identical to the default value of the column.

In a table lookup, the row pointer table is looked up using the row index. Simultaneously, the default value table is looked up using the column ID. Then the column ID is added to the row pointer to make the index of the row element table. The row element table is looked up using the index. The returned row ID is compared with the input row ID. When they matched, the value in the row element table is output. Otherwise, since the value of the row element table is invalid, the value of the default value table is output.

3 Formal Algorithm Description of SBT Method

We denote the w input bytes as a w bytes string p_{in}. Let P be the set consisting of all the rule strings p_i. The character indices of strings start from 0. The length of the string p is written as $|p|$. We write the concatenation of strings p and p' as $p :: p'$. $\texttt{substr}(p, i, j)$ is the j characters substring of p starting from position i, and $\texttt{prefix}(p, k) = \texttt{substr}(p, 0, k)$. We write $p \preceq p'$ when the string p is p' itself or a suffix of p'. $\texttt{IN}(X, p)$ is the IN of the longest string p' in X which satisfies $p' \preceq p$.

Variables and Tables $T = \{p \,|\, p \preceq p_i(p_i \in P)\}$.

$$I_k = \begin{cases} \text{all the byte characters} & \text{when } k = 1 \\ \bigcup_{i,j,l(l<k)}(\texttt{substr}(p_i, j, k) \cup \texttt{prefix}(p_i, l)) & \text{when } k = 2, 4, 8, \cdots, w \end{cases}$$

for $k = 2, 4, 8, \cdots, w$ and $p, p' \in I_{k/2}$,

$$C_k(\texttt{IN}(I_{k/2}, p), \texttt{IN}(I_{k/2}, p')) = \begin{cases} \texttt{IN}(I_k, p') & \text{when } |p'| < k/2 \\ \texttt{IN}(I_k, p :: p') & \text{when } |p'| = k/2 \end{cases}$$

for $k = 1, 2, 4 \cdots, w$ and $i = 0, k, 2k, \cdots, w - k$ and $p \in I_k$ and $s \in T$,

$$m(k, i) = \begin{cases} \texttt{IN}(I_1, \texttt{substr}(p_{in}, i, 1)) & \text{when } k = 1 \\ C_k(m(k/2, i), m(k/2, i + k/2)) & \text{when } k \geq 2 \end{cases}$$

$$NS_k(\texttt{IN}(T, s), \texttt{IN}(I_k, p)) = \begin{cases} \texttt{IN}(T, p) & \text{when } |p| < k \\ \texttt{IN}(T, s :: p)) & \text{when } |p| = k \end{cases}$$

T is the nodes of the rule string trie. I_k is the input suffixes for NS_k table. Table C_k tells the IN of the longest suffix of given 2 suffixes' concatenation. $m(k, i)$ is $\texttt{IN}(I_k, \texttt{substr}(p_{in}, i, k))$, the IN of the longest suffix of a portion of p_{in}. Table NS_k tells the state change after k characters.

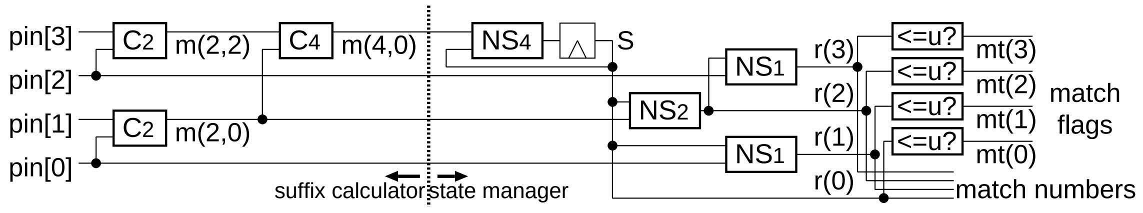


Figure 7: Structure of the pattern matcher when $w = 4$

Static Preparation Make T, I_k from given P and assign INs. The INs of the elements in I_1 are their character codes. In the IN assignment of T, the elements which match some p_i (i.e. $\{p \mid p \in T \text{ and } p_i \preceq p \text{ for some } p_i\}$) are numbered first, and then the other elements are numbered, in increasing order. Let u be the maximum IN of the elements in T which match some p_i. Then, calculate lookup tables C_k and NS_k.

Input In each cycle, w *characters of stream data* are input. Let $p_{in}(t)$ be the w input characters of the t-th clock cycle.

Output Let $q_i = p_{in}(0) :: \cdots :: p_{in}(t-1) :: \texttt{prefix}(p_{in}(t), i)$. In each cycle, for each q_i, the following are output: (1) $r(i) : \texttt{IN}(T, q_i)$, i.e. *the IN of the longest string in* T *that matches* q_i, and (2) $mt(i)$: A *match flag* which indicates whether q_i matches some p_j.

Algorithm Procedure Let S be a state variable which holds $\texttt{IN}(T, p_{in}(0) :: \cdots :: p_{in}(t-1))$. In each clock, do following using lookup tables:

1. Calculate each $m(k, i)$ from the input $p_{in}(t)$.
2. Calculate $r(i)$ $(i = 0, ..., w-1)$ recursively: $r(0) = S$,
   $r(k(2l+1)) = NS_k(r(2kl), m(k, 2kl)) \qquad k = 1, 2, 4, \cdots, wk(2l+1) < w$
3. $mt(i) = true$ if $r(i) \leq u$. Otherwise, $mt(i) = false$.
4. Assign the next state $NS_w(S, m(w, 0))$ to S

4 Correctness Proof

In the following description, p, p', q, q', r, s are strings, and X is a set of strings.

Definition 1
$\mathrm{LS}(X, p)$ is the longest string p' in X which satisfies $p' \preceq p$.

Following corollaries are immediately derived from the definition of the LS function.

Corollary 1
$p' \preceq p \rightarrow |p'| \leq |p|$

Corollary 2
$\mathrm{LS}(X, p) \in X$

Corollary 3
$p \preceq q \wedge q \preceq r \rightarrow p \preceq r$

Corollary 4
$p \succeq q \leftrightarrow p :: r \succeq q :: r$

Corollary 5
$\mathrm{LS}(X, p) \preceq p$

Proposition 1
$p' \preceq p \wedge p \preceq p' \to p = p'$

Proof
From Corollary 1, $|p| = |p'|$ is derived. By the definition of the $\preceq$ operator, $|p| = |p'|$ is satisfied only when $p = p'$ ∎

Proposition 2
$p \succeq p', p' \in X \to \mathrm{LS}(X, p) \succeq p'$

Proof
Since $p \succeq p'$, $\mathrm{LS}(X, p)$ is p' itself or a string that contains p' as a suffix. Therefore, $\mathrm{LS}(X, p) \succeq p'$ ∎

Proposition 3
$p \in X \to \mathrm{LS}(X, p) = p$

Proof
By putting $p = p'$ in Proposition 3, $\mathrm{LS}(X, p) \succeq p$ is derived. By Corollary 5, $\mathrm{LS}(X, p) \preceq p$ stands. Therefore, using Proposition 1, $\mathrm{LS}(X, p) = p$ is derived ∎

Corollary 6
$\mathrm{LS}(X, \mathrm{LS}(X, p)) = \mathrm{LS}(X, p)$

Proof
Because of Corollary 2, $\mathrm{LS}(X, p) \in X$. Therefore, from Proposition 3, $\mathrm{LS}(X, \mathrm{LS}(X, p)) = \mathrm{LS}(X, p)$ is derived ∎

Proposition 4
$p \succeq p' \to \mathrm{LS}(X, p) \succeq \mathrm{LS}(X, p')$

Proof
By Corollary 5, $p' \succeq \mathrm{LS}(X, p')$ stands. Therefore, from Corollary 3, $p \succeq \mathrm{LS}(X, p')$ is derived. By Corollary 2, $\mathrm{LS}(X, p') \in X$ stands. Therefore, from Proposition 2, $\mathrm{LS}(X, p) \succeq \mathrm{LS}(X, p')$ is derived ∎

Proposition 5
$p \succeq p' \succeq \mathrm{LS}(X, p) \to \mathrm{LS}(X, p') = \mathrm{LS}(X, p)$

Proof
When $p \succeq p' \succeq \mathrm{LS}(X, p)$, using Proposition 4, $\mathrm{LS}(X, p) \succeq \mathrm{LS}(X, p') \succeq \mathrm{LS}(X, \mathrm{LS}(X, p))$ is derived. Therefore, by Corollary 6, $\mathrm{LS}(X, p) \succeq \mathrm{LS}(X, p') \succeq \mathrm{LS}(X, p)$ stands. Therefore, from Proposition 1, $\mathrm{LS}(X, p) = \mathrm{LS}(X, p')$ is derived. ∎

Lemma 1
For $k = 2, 4, 8, \cdots, w$,
$C_k(\mathtt{IN}(I_{k/2}, p), \mathtt{IN}(I_{k/2}, p')) = \mathtt{IN}(I_k, p :: p')$ where $|p| = |p'| = k/2$.

Proof
Let $q = \mathrm{LS}(I_{k/2}, p)$, $q' = \mathrm{LS}(I_{k/2}, p')$, and $r = \mathrm{LS}(I_k, p :: p')$. By definition of $\mathtt{IN}$, $\mathtt{IN}(I_{k/2}, p) = \mathtt{IN}(I_{k/2}, q)$ and $\mathtt{IN}(I_{k/2}, p') = \mathtt{IN}(I_{k/2}, q')$.

- When $|r| \geq k/2$
  $r \succeq p'$ stands. Therefore we can put $r = r_p :: p'$, where $r_p = \mathtt{prefix}(r, |r| - k/2)$. Because of the definition of I_k and $I_{k/2}$, $r_p, p' \in I_{k/2}$ stands. Therefore, from Proposition 3, $q' = \mathrm{LS}(I_{k/2}, p') = p'$ is derived.

  Since $p \succeq r_p$ and $r_p \in I_{k/2}$, from Proposition 2, $q = \mathrm{LS}(I_{k/2}, p) \succeq r_p$ is derived. Therefore, by Corollary 4, $q :: q' = q :: p' \succeq r_p :: p' = r = \mathrm{LS}(I_k, p :: p')$. Since $p \succeq q$, by Corollary 4, $p :: p' = p :: q' \succeq q :: q'$ stands. Therefore, from Proposition 5, $\mathrm{LS}(I_k, p :: p') = \mathrm{LS}(I_k, q :: q')$ is derived.

  Therefore, since $|q'| = |p'| = k/2$, $C_k(\mathtt{IN}(I_{k/2}, p), \mathtt{IN}(I_{k/2}, p')) = C_k(\mathtt{IN}(I_{k/2}, q), \mathtt{IN}(I_{k/2}, q')) = \mathtt{IN}(I_k, q :: q') = \mathtt{IN}(I_k, p :: p')$.

- When $|r| < k/2$
  $p' \succeq r$ stands.

  - When $|q'| = k/2$
    Since $q' = p'$, 4, $p :: p' = p :: q' \succeq q :: q'$ stands. Furthermore, $q :: q' > q' = p' > r = \mathrm{LS}(I_k, p :: p')$. Therefore, by Corollary 3, $q :: q' > \mathrm{LS}(I_k, p :: p')$ stands. Therefore, from Proposition 5, $\mathrm{LS}(I_k, q :: q') = \mathrm{LS}(I_k, p :: p')$ is derived. Since $|q'| = k/2$, $C_k(\mathtt{IN}(I_{k/2}, p), \mathtt{IN}(I_{k/2}, p')) = C_k(\mathtt{IN}(I_{k/2}, q), \mathtt{IN}(I_{k/2}, q')) = \mathtt{IN}(I_k, q :: q') = \mathtt{IN}(I_k, p :: p')$.
  - When $|q'| < k/2$
    By the definition of I_k and $I_{k/2}$, $r \in I_{k/2}$ stands. Since $p' \succeq r$, by Proposition 2, $q' = \mathrm{LS}(I_{k/2}, p') \succeq r = \mathrm{LS}(I_k, p :: p')$ stands. Furthermore, since $p :: p' \succeq p' \succeq q'$, $p :: p' \succeq q'$ stands. Therefore, by Proposition 5, $\mathrm{LS}(I_k, q) = \mathrm{LS}(I_k, p :: p')$ stands. Since $|q'| < k/2$, $C_k(\mathtt{IN}(I_{k/2}, p), \mathtt{IN}(I_{k/2}, p')) = C_k(\mathtt{IN}(I_{k/2}, q), \mathtt{IN}(I_{k/2}, q')) = \mathtt{IN}(I_k, q') = \mathtt{IN}(I_k, p :: p')$ ∎

Lemma 2
For $k = 1, 2, 4, \cdots, w$,
$NS_k(\mathtt{IN}(T, s), \mathtt{IN}(I_k, p)) = \mathtt{IN}(T, s :: p)$.

Proof
The Lemma is proved in a way similar to Lemma 1.

Lemma 3
For $k = 1, 2, 4, \cdots, w$ and $i = 0, k, 2k, \cdots, w - k$, $m(k, i) = \mathtt{IN}(I_k, \mathtt{substr}(p_{in}, i, k))$.

Proof
We prove the Lemma by a derivation on k.

- When $k = 1$
  By the definition, $m(k, i) = m(1, i) = \mathtt{IN}(I_1, substr(p_{in}, i, 1)$. Therefore, the Lemma stands.

- When the Lemma stands for $k = k'/2$ $(k' \geq 2)$
  By the assumption of the derivation, $m(k, i) = C_{k'}(m(k'/2, i), m(k'/2, i{+}k'/2)) = C_{k'}(\mathtt{IN}(I_{k'/2}, \mathtt{substr}(p_{in}, i,$ $k'/2, k'/2)))$. Therefore, by Lemma 1, $m(k, i) = \mathtt{IN}(I_{k'}, \mathtt{substr}(p_{in}, i, k'/2) :: \mathtt{substr}(p_{in}, i+$

$k'/2, k'/2)) = \mathtt{IN}(I_{k'}, \mathtt{substr}(p_{in}, i, k')$ stands. Therefore, the Lemma also stands for $k = k'$. ∎

Lemma 4
In t-th clock cycle, $S = \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t-1))$.

Proof
We prove the Lemma by a derivation on t

- When $t = 0$
  The Lemma stands since the S is initialized to $\mathtt{IN}(T, '')$.
- When the Lemma stands for $t = t' - 1$
  By the assumption of the derivation, in $(t'-1)$-th clock cycle, $S = \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t'-2))$. Since $m(w, 0) = \mathtt{IN}(I_w, p_{in})$ stands by Lemma 3, in t'-th clock cycle, $S = NS_w(\mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t'-2)), \mathtt{IN}(I_w, p_{in}(t'-1)))$. Therefore, by Lemma 2, $S = \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t'-1))$. Thus, the Lemma also stands when $t = t'$ ∎

Finally, we show that the SBT method always outputs correct pattern IDs by proving the following theorem.

Theorem 1
For $i = 1, 2, 3, \cdots, w$, (A) $r(i)$ is uniquely defined, and (B) In t-th clock cycle, $r(i) = \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t-1) :: \mathtt{substr}(p_{in}(t), 0, i))$.

Proof
For $i = 0$, $r(0)$ is uniquely defined as $r(0) = S = \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t-1) :: \mathtt{substr}(p_{in}(t), 0, 0))$. Therefore, both (A) and (B) stand.

For other i, we prove the theorem by proving that the following lemma stands for all $j (0 \leq j \leq \log w - 1)$ by a derivation on j.

> (A) and (B) stands for all the integers i that satisfies num2s$(i) = j$, where the num2s function returns the number of 2's the argument integer has as the primary factors.

Since $2l + 1$ is an odd number in the definition of $r(i)$, $i = k(2l+1)$ stands only when $k = 2^{\text{num2s}(i)}$. Since the k is uniquely determined, the value of the l is also uniquely determined($l = (i/k - 1)/2$). Therefore, the combination of k and l that satisfies $i = k(2l+1)$ is unique. Therefore, $r(i)$ is defined at most one time for each i. Thus, we can prove (A) by simply showing that each $r(i)$ is really defined.

- When $j = \log w - 1$
  Only $i = w/2$ satisfies num2s$(i) = j$. For this i, $i = k(2l+1)$ is satisfied when $k = w/2$ and $l = 0$. Therefore, $r(w/2) = NS_{w/2}(r(2 * w/2 * 0), m(w/2, 2 * w/2 * 0)) = NS_{w/2}(r(0), m(w/2)) = NS_{w/2}(S, \mathtt{IN}(I_{w/2}, \mathtt{substr}(p_{in}(t), 0, w/2)) = NS_{w/2}(\mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t-1)), \mathtt{IN}(I_{w/2}, \mathtt{substr}(p_{in}(t), 0, w/2)) = \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots :: p_{in}(t-1)) :: \mathtt{substr}(p_{in}(t), 0, w/2))$. Thus (B) is satisfied. Furthermore, since $r(0)$ is uniquely defined, (A) is satisfied.
- When (A) and (B) is satisfied for $j > j'$
  For each i such that num2s$(i) = j'$, in the definition of $r(i)$, $i = k(2l+1)$ stands only when $k = 2^{j'}$. Since num2s$(2kl)$ = num2s$(2 \cdot 2^{j'} l)$ = num2s$(2^{j'+1} l) \geq j' + 1$, by the

assumption of the derivation, (A) and (B) stands for $i = 2kl$. Therefore, for each i such that num2s$(i) = j'$,

$$\begin{aligned}
r(i) &= r(k(2l+1)) \\
&= NS_k(r(2kl), m(k, 2kl)) \\
&= NS_k(\mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots \\
&\qquad :: p_{in}(t-1) :: \mathtt{substr}(p_{in}(t), 0, 2kl)), \mathtt{IN}(I_k, \mathtt{substr}(p_{in}(t), 2kl, k))) \\
&= \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots \\
&\qquad :: p_{in}(t-1) :: \mathtt{substr}(p_{in}(t), 0, 2kl) :: \mathtt{substr}(p_{in}(t), 2kl, k)) \\
&= \mathtt{IN}(T, p_{in}(0) :: p_{in}(1) :: \cdots \\
&\qquad :: p_{in}(t-1) :: \mathtt{substr}(p_{in}(t), 0, k(2l+1)))
\end{aligned}$$

stands. Therefore, both (A) and (B) is satisfied for $j = j'$. ∎

5 Concluding Remarks

In this paper, we have presented the correctness proof of SBT method [1]. Therefore, the algorithm description of SBT method is now complete. However, we must further analyze the SBT method in respects including the worst case memory usage and the optimal packing method of row elements into a row element table. These aspects are important for a practical use of SBT method. We will present the analysis as our future work.

Acknowledgements

This research is partially supported by the Special Coordination Fund for Promoting Science and Technology, CREST project of Japan Science and Technology Corporation, and 21st century COE project of Japan Society for the Promotion of Science.

References

[1] Y. Sugawara, M. Inaba, K. Hiraki, "Over 10gbps string matching mechanism for multi-stream packet scanning systems," in *Proc. of 14th Intl. Conf. on Field Programmable Logic and Applica tions(FPL '04)*, August 2004.

[2] C. J. Coit, S. Staniford, J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," in *DISCEXII, DARPA Information Survivability conference and Exposition*, 2001.

[3] Y. H. Cho, S. Navab, W. H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," in *Proc. of 12th Intl. Conf. on Field Programmable Logic and Applications(FPL '02)*, September 2002.

[4] I. Sourdis, D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *Proc. of 13th Intl. Conf. on Field Programmable Logic and Applications(FPL '03)*, September 2003.

[5] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," in *Proc. of 12th Intl. Conf. on Field Programmable Logic and Applications(FPL '02)*, September 2002.

[6] R. Sidhu, V. K. Prasanna, "Fast regular expression matching using fpgas," in *Proc. of 9th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'01)*, April – May 2001.

[7] B. L. Hutchings, R. Franklin, D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. of 10 th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'02)*, pp. 111–120, September 2002.

[8] C. Clark, D. Schimmel, "Scalable pattern matching for high speed networks," in *Proc. of 12th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'04)*, April 2004.

[9] J. Moscola, J. Lockwood, R. P. Loui, M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *Proc. of 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'03)*, pp. 31 – 38, April 2003.

[10] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, J. Lockwood, "Deep packet inspection using parallel bloom filters," in *Proc. of 11th IEEE Symp. on High Performance Interconnects (HotI '03)*, pp. 44 – 51, August 2003.

[11] Z. K. Baker, V. K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," in *Proc. of the 2004 ACM/SIGDA 12th Intl. Symp. on Field programmable gate arrays(FPGA'04)*, pp. 223–232, February 2004.

[12] A. V. Aho, M. J. Corasick, "Efficient String Matching : An Aid to Bibliographic Search," *Communications of the ACM*, vol. Vol. 18, pp. 333 – 340, June 1975.