A Fast Scalable Automaton-Matching Accelerator for Embedded Content Processors

KUO-KUN TSENG
Hungkuang University
YUAN-CHENG LAI
National Taiwan University of Science and Technology
and
YING-DAR LIN and TSERN-HUEI LEE
National Chiao Tung University

Home and office network gateways often employ a cost-effective embedded network processor to handle their network services. Such network gateways have received strong demand for applications dealing with intrusion detection, keyword blocking, antivirus and antispam. Accordingly, we were motivated to propose an appropriate fast scalable automaton-matching (FSAM) hardware to accelerate the embedded network processors. Although automaton matching algorithms are robust with deterministic matching time, there is still plenty of room for improving their average-case performance. FSAM employs novel prehash and root-index techniques to accelerate the matching for the nonroot states and the root state, respectively, in automation based hardware. The prehash approach uses some hashing functions to pretest the input substring for the nonroot states while the root-index approach handles multiple bytes in one single matching for the root state. Also, FSAM is applied in a prevalent automaton algorithm, Aho-Corasick (AC), which is often used in many content-filtering applications. When implemented in FPGA, FSAM can perform at the rate of 11.1Gbps with the pattern set of 32,634 bytes, demonstrating that our proposed approach can use a small logic circuit to achieve a competitive performance, although a larger memory is used. Furthermore, the amount of patterns in FSAM is not limited by the amount of internal circuits and memories. If the high-speed external memories are employed, FSAM can support up to 21,302 patterns while maintaining similar high performance.

Authors' addresses: K. K. Tseng, Department of Computer and Information Engineering, Hungkuang University, Taichung, Taiwan, 433; email: kktseng@sunrise.hk.edu.tw; Y. C. Lai, Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan, 106; email: laiyc@cs.ntust.edu.tw; Y. D. Lin, Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, 300; email: ydlin@cis.nctu.edu.tw; T. H. Lee, Department of Communication Engineering, National Chiao Tung University, Hsinchu, Taiwan, 300; email: tlee@banyan.cm.nctu.edu.tw.

DOI 10.1145/1509288.1509291 http://doi.acm.org/10.1145/1509288.1509291

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet-switching networks*; I.5.4 [**Pattern Recognition**]: Applications—*Text processing*

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: String matching, content filtering, automaton, Aho-Corasick, Bloom filter

ACM Reference Format:

Tseng, K. K., Lai, Y. C., Lin, Y. D, Lee, T. H. 2009. A fast scalable automaton-matching accelerator for embedded content processors. ACM Trans. Embedd. Comput. Syst. 8, 3, Article 19 (April 2009), 30 pages. DOI = 10.1145/1509288.1509291 http://doi.acm.org/10.1145/1509288.1509291

1. INTRODUCTION

In recent years, deeper and more complicated content filtering has been required for applications dealing with intrusion detection, keyword blocking, antivirus, and antispam. In such applications, string matching usually occupies 30% to 70% of the system workload [Mike et al. 2001; Antonatos et al. 2004]. In particular, new content-filtering applications are increasingly being built on the home and office network gateways, which are often implemented with an embedded network processor with moderate performance. Thus, as transmission speed increases, it becomes more necessary to design an appropriate string-matching accelerator to offload the work of string matching from the network processor.

To understand the necessary requirements of string-matching algorithms, we surveyed real patterns from open source software, which includes Snort [Roesch et al. 2006] for intrusion detection, ClamAV [2006] for antivirus, SpamAssassin [2006] for antispam, and SquidGuard [2006] and DansGuardian [2006] for Web blocking. In Table I, the necessary requirements can be concluded to be those matching the variable-length patterns, multiple patterns, and online processing of all content-filtering applications. Because content-filtering applications often perform the exact matching, then the string-matching allowing errors is not always necessary. Moreover, the complex patterns, such as those created by adopting class, wildcard, regular expression, and case sensitivity, might increase the expressive power of the patterns and hence might increase the matching time and space requirement. Since the complex patterns can be converted into patterns composed of multiple simple patterns [Navarro et al. 2002], they are optional in most applications.

Current existing online string-matching algorithms for content-filtering can be classified into five categories: simple matching, dynamic programming, bit parallel, backward filtering, and automaton, as shown in Table II. The simple matching compares the text against patterns with the naïve algorithm, and its average and worst-case time complexities are both poor as $O(nm)$, where n and m are the lengths of the text and patterns, respectively. The dynamic-programming [Navarro 2001] and bit-parallel [Wu et al. 1992] algorithms have the better deterministic average and worst-case time complexities $O(n)$, but they are inappropriate for variable-length and multiple patterns. The

Table I. String-matching Requirements for Content-Filtering Applications

Functions	Description	Intrusion detection	Web blocking	Anti-virus	Anti-spam
Error allowance	Allow error with some number of characters	Unnecessary	Unnecessary	Unnecessary	Optional
Multiple patterns	Arbitrary pattern amount in a single matching	Necessary	Necessary	Necessary	Necessary
Class	One character represents multiple alphabets	Unnecessary	Optional	Unnecessary	Optional
Wildcard	Don't care multiple characters	Optional	Optional	Optional	Optional
Regular expression	Kleene star, concatenation, OR	Optional	Optional	Optional	Optional
Variable length	Arbitrary length of patterns	Necessary (Short length)	Necessary (Medium length)	Necessary (Long length)	Necessary (Medium length)
Online processing	Text is unknown before match	Necessary	Necessary	Necessary	Necessary
Case sensitivity	Alphabet is case sensitive	Unnecessary	Optional	Unnecessary	Optional

backward-filtering algorithm [Boyer et al. 1977] employs a heuristic technique for variable-length patterns with the sublinear average-case time complexity, but its worst-case time complexity O(nm) is poor, and the performance is not deterministic for a large pattern set. Only the automaton-based algorithms, such as Aho-Corasick (AC) [Aho et al. 1975], support the variable-length and multiple patterns and also have the deterministic worst-case time complexity O(n). Thus, the automaton-based algorithm was selected as a base to develop our new approaches.

AC is a typical deterministic finite automaton (DFA)-based algorithm used for string-matching, and there are several variations. Bitmap AC [Tuck et al. 2004] used bitmap compression to reduce the storage of AC states. AC_BM [Mike et al. 2001; Coit et al. 2002; Desai et al. 2002] was a combination of the AC and Boyer Moore (BM) algorithms and aimed to improve the conventional AC from O(n) to the sublinear time complexity with the BM approach. AC_BDM [Raffinot 1997] combined AC with backward dawg matching (BDM) and also improved the average-case time complexity of the conventional AC. Bit-split AC [Tan et al. 2005] split the width of the input text into a smaller bit-width to reduce the memory usage and the number of comparisons for selecting the next states. Since AC_BM has the worst-case time complexity O(nm), AC_BDM requires double space and has overhead for switching between AC and BDM, and bit-split AC requires a large match vector for each bit-split state, they are impractical for a large number of patterns. Hence, a scalable bitmap AC with superior space efficiency is preferable for our purpose.

Although bitmap AC has the good worst-case matching time complexity of O(n), it is insufficient for high-speed processing. In this article, we present

Table II. Comparison of the Online String-Matching Algorithms

Algorithm	Simple Matching	Dynamic Programming	Backward Filtering	Automaton	Bit Parallel
Description	Compare the text and the patterns byte by byte	Compute matrix similarity of the texts and the patterns	Do backward scanning in the text window for skipping the multibytes text	Search through a Deterministic Finite Automation (DFA)	Simulate Non Deterministic Finite Automation (NFA) by bitwise operations
Average Time	O(nm)	O(n)	Sublinear	O(n)	O(n)
Worst Time	O(nm)	O(n)	O(nm)	O(n)	O(n)
Text Length	Variable long length	Fixed short length	Variable long length	Variable long length	Variable long length
Pattern Length	Fixed short length	Fixed short length	Variable short length	Variable short length	Fixed short length
Multiple Pattern	No	No	Yes	Yes	Yes
Regular Expression	No	No	No	Yes	Yes
Pros for hardware	Easy for parallelism and pipeline	Systolic, array is regular	Storage is smaller than Automaton	Comparison is a lookup operations	Bitwise operation is fast
Cons for hardware	Duplicated circuits or slow performance	Large array circuit is impractical	Complex to compute skipping length	Table size is Larger than Bit-Parallel	Not feasible to have a long vector
Typical Algorithm	Naïve	Edit Distance	Boyer-Moore	Aho-Corasick	Shift-OR

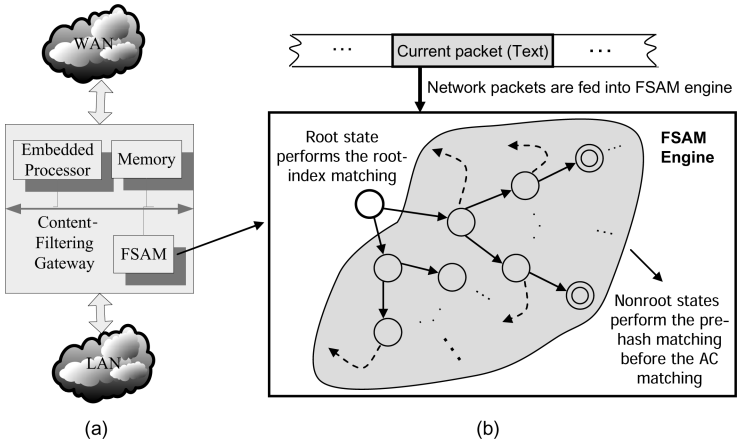


Fig. 1. (a) Content filtering gateway, (b) FSAM performing two techniques: root-index matching for the root state and the prehash matching for the nonroot states.

a fast scalable automaton matching (FSAM) that is built on an embedded system and applied to a network gateway to perform deep-content filtering, as shown in Figure 1(a). FSAM employs two novel techniques: prehash for the nonroot states and root-index for the root state in order to accelerate the automaton-based algorithms as shown in Figure 1(b). The prehash approach is

a quick scanning for the nonroot states to avoid the time-consuming automaton matching. The idea is to have an initial hashing for the substring of the input text and comparing the result with the vector for the suffixes of the state in the bitmap AC finite automaton. If no-hit occurs, meaning a true negative, then the slow automation matching is no longer required. For the root state, the root-index approach uses a compressed technique to remember all the next states whose lengths, counting from the root state, are less than $l (l > 1)$. Thus, multiple bytes of length l, rather than 1 byte, can be handled in one single matching for the root state to accelerate the matching speed, although the processing of such variable bytes per cycle is dependent on the characteristics of the matched texts and patterns. In fact, since the root state is often visited in the matching operation, the root-index approach is an effective acceleration approach.

We developed the appropriate hardware design according to our proposed algorithms. To evaluate our approaches, the space and time complexities are formally analyzed using real patterns. Also, the FSAM design, which was implemented in Xilinx FPGA, can achieve 11.1Gbps throughput with a pattern set of 32,634 bytes. The results demonstrate that our proposed approach uses a small logic circuit to achieve a competitive performance and support a large pattern set, comparing with the previous matching hardware.

The rest of this article is organized as follows: Section 2 includes the surveys of AC related algorithms, related hashing-matching works, and existing string-matching hardware. Section 3 describes the algorithm and architecture of FSAM as well as the detailed pseudocode of prehash and root index. The formal analysis and evaluation of real patterns and network traffic are shown in Section 4. The hardware implementation and its performance comparison with other methods are demonstrated in Section 5. Finally, we draw our conclusion in Section 6.

2. BACKGROUND

The most related works to our approaches are AC, bitmap AC, and hashing matching algorithms, so a brief tutorial for the first two is presented in Section 2.1 and for the third one is given in Section 2.2. Finally, the related string-matching hardware is introduced in Section 2.3.

2.1 AC and Bitmap AC Algorithms

As AC is our accelerating target, we need to know more about AC and AC related algorithms. AC state machine is constructed from the patterns to be matched in the preprocessing phase and requires three preprocessing functions. The first is *goto* function, which is used to traverse from node to node. The second is *failure* function, which is traversed when there is no next state, and the third is *output* function, which output the matched state pattern for matched patterns. Actually, AC is a special automaton for string-matching that uses the failure links to reduce the number of the next state links.

In the searching phase, AC being algorithm for processing multiple patterns searches the patterns in the text by traversing the patterns in a data structure

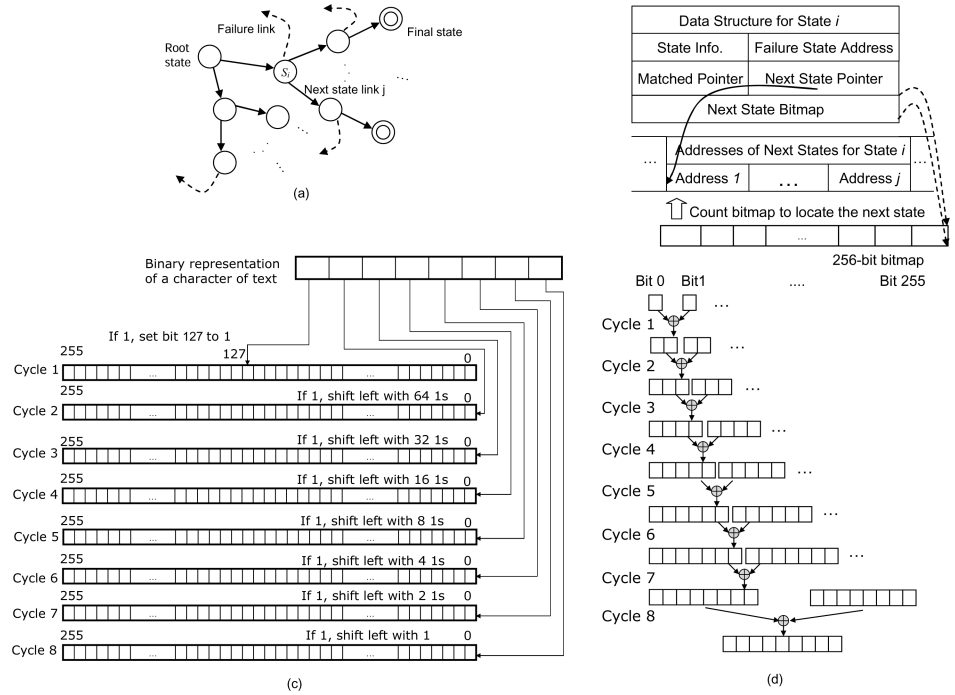


Fig. 2. AC and bitmap AC, (a) AC tree for string-matching, (b) data structure of bitmap AC for state i using a bitmap to locate the next state among 0-j states, (c) circuit for converting a character to the corresponding bit position in the 256-bit bitmap, (d) circuit for counting the number of 1s in the 256-bit bitmap.

of automaton as in Figure 2(a), the root state is the initial matching for the AC tree. A state S_i has a next-state link if it matches a corresponding character in the patterns. For handling the unmatched case, each state keeps a failure link to another state, except the failure to the root state. There are two alternatives to storing the next state links.

(1) *Table:* Each state has 256 next states for all letters. Table data structure has the merit of fast matching, but it wastes the space if the table is sparse.
(2) *Link list:* Each state only has the link lists of the existing next states. This data structure has smaller space, but it is slow if there are many next states.

Bitmap AC is a compromise between the table and link list approaches. Bitmap AC uses a 256-bit bitmap to store the next state links for each state. The 256-bit bitmap is added in the data structure of the AC, as in Figure 2(b). The idea for locating the next state is using the next state pointer of S_i as a base address, and counting the number of 1s in the 256-bit bitmap to locate the offset of the next state.

Although bitmap AC can reduce the memory requirement, its matching procedure uses a lot of clock cycles, which is dominated by loading the state and

performing the population count. For the cycle of loading bitmap, each state contains 32-bit next state, 32-bit failure state, 32-bit pointer and 256-bit next-state bitmap, causing 11 words to be required in total. If memory access time is 2 cycles per word, 22 cycles are needed for loading a state. For performing the conversion, converting from decimal to bit position and counting the number of 1s for locating the next state, as shown in Figures 2(c) and 2(d), respectively. Due to the processing of the 256-bit register, 8 cycles are required for each operation, and there are at least 16 cycles for matching a byte in the text. Consequently, bitmap AC requires at least 38 cycles to process a single byte.

2.2 Related Hashing-Matching Works

There are several hashing-matching algorithms, mainly including BFSM [Dharmapurikar et al. 2004], PHmem [Sourdis et al. 2005], Hash-Mem [Papadopoulos et al. 2005], and Piranha [Antonatos et al. 2005] applied to the string-matching. The basic idea of the hashing-matching works is to use hashing functions to reduce the probability of false positive for the original matching algorithms. Their common problems are that they require nondeterministic verification time and that they cannot afford long patterns and a large set of patterns.

Since BFSM was the first approach to use hashing function in the string-matching and our prehash technique is motivated from it, we introduce BFSM as a representative for the hashing string-matching works. In BFSM, the Bloom filter hashing is employed to perform the approximate matching and cooperates with the other exact-matching algorithms for content-filtering as shown in Figure 3(a). The main idea of the Bloom filter is to use multiple hashing functions to improve the hashing performance. In the preprocessing phase of BFSM, each length j of all patterns are hashed into the corresponding bit vectors V_j, and each V_j is associated with k hashing functions $H_{j,k}$. For example, in Figure 3(b), the hashing functions $H_{1,1}$, $H_{1,2}$ ... $H_{1,k}$ are used for length one of all patterns. Figure 3(c) shows its searching phase where the substring of each length in the compared text is hashed with k hashing functions and compared with the corresponding bit vector to determine whether the text is possibly matched or not with the AND function.

The basic philosophy of BFSM is that it uses multiple hashing functions to reduce the probability of false-positive. When BFSM chooses k independent hashing functions to hash N independent patterns into a vector with size M, the probability of false-positive P_{fp} is obtained as

$$P_{fp} = \left(1 - \left(1 - \frac{1}{M}\right)^{Nk}\right)^{k}, \tag{1}$$

according to Dharmapurikar et al. [2004]. Equation (1) holds under the assumption of the uniform hashing function, meaning that the probability of hashing to any position is equal to $1/M$.

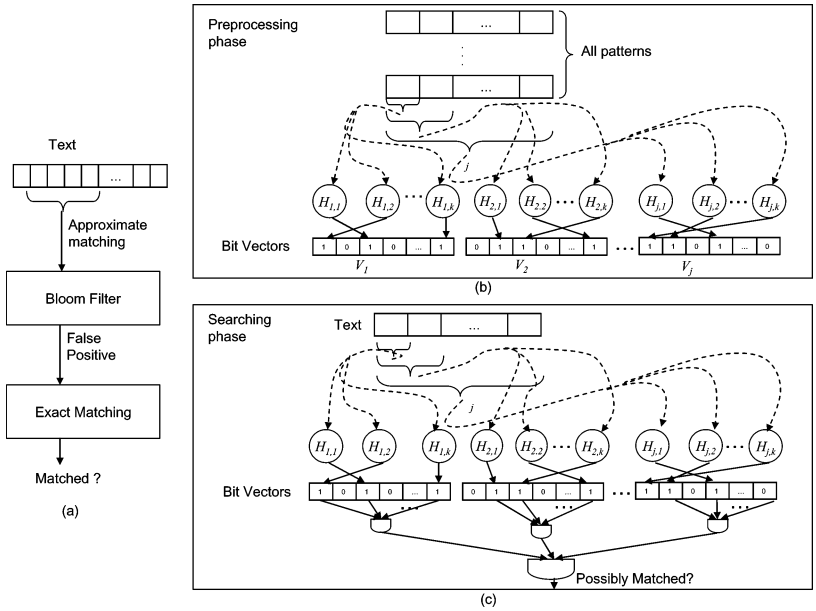


Fig. 3. (a) Bloom filter for string-matching, (b) Each pattern is hashed into bit vectors in the preprocessing phrase, (c) Text is hashed and compared with the bit vectors in the searching phrase.

2.3 String-Matching Hardware

As mentioned earlier, the sting matching is a bottleneck for content-filtering systems, and hence the hardware solutions are required for high-speed content processing. For the algorithms in Table II, the existing string-matching hardware are mainly based on dynamic programming, simple matching, and automaton algorithms, while backward filtering and bit-parallel algorithms are seldom implemented as the matching hardware. The work from Blüthgen et al. [2000] and Sastry et al. [1995] implemented the dynamic programming algorithm with the systolic array, which is only appropriate for short patterns and text, since the circuit size is proportional to the lengths of patterns and the text.

Since the naïve algorithm (simple matching) is easily implemented in hardware design, many previous works applied the parallel circuits, content addressable memory (CAM), and hashing function techniques to accelerate the naive algorithm. For instance, the works Park et al. [1999], Sourdis et al. [2003], and Cho et al. [2005] used the parallel circuit; the works Gokhale et al. [2002] and Sourdis et al. [2004] applied the content addressable memory (CAM); and the works Dharmapurikar et al. [2004], Sourdis et al. [2005], Papadopoulos et al. [2005], and Rubin et al. [2006] employed the hashing functions. However, the previously described accelerating techniques are not scalable to a large set of patterns since parallel, CAM, and internal hashing circuits are increased as the number of patterns is increased.

The other prevalent hardware is automaton-based hardware, due to the support of the deterministic matching time and a large amount of patterns. The

automaton-based hardware can be classified into two categories, namely, the DFA and the nondeterministic finite automaton (NFA)-based hardware. DFA-based hardware has a unique transition, which activates one state at a time and normally has a large number of states. NFA can handle multiple transitions at one time, but it requires parallel circuits for entering its multiple next states. Therefore, most DFA-based hardware use the table or link list to store their patterns, while most NFA-based hardware use parallel reconfigurable circuits to handle their patterns.

For DFA-based hardware, there are three common designs in recently developed string-matching hardware, namely, the AC based hardware [Tan et al. 2005, Aldwairi et al. 2005], the Regular Expression (RE)-based hardware [Lockwood et al. 2001; Moscola et al. 2003] based hardware, and the Knuth-Morris-Pratt (KMP), [Baker et al. 2004; Tripp 2005; Bu et al. 2004]. In order to save a great number of states, KMP and AC were simplified from RE DFA by disabling the regular expression patterns. Each AC DFA supports multiple simple patterns, while each KMP DFA only supports a single simple pattern. Thus, many KMP DFAs use duplicate hardware for supporting multiple patterns.

For NFA-based hardware, there are two variations, namely, the comparator NFA [Sidhu et al. 2001; Franklin et al. 2002], which used the distributed comparators, and the decoder NFA [Clark et al. 2003; Clark et al. 2004; Clark et al. 2005], which used the character decoder (shared decoder) to build its NFA circuits.

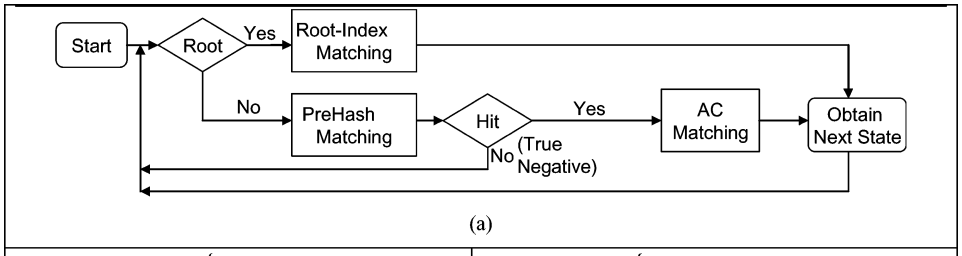
3. ARCHITECTURE AND ALGORITHM DESIGN

Our FSAM incorporates two techniques, the prehash matching and root-index matching. Except the root state, each state is applied the prehash technique to avoid the bitmap AC matching. Dissimilar to BFSM, prehash uses a single hashing function and only builds the corresponding bit vector for the substrings of each state, making the hashing technique feasible in string matching by reducing the hardware complexity. On the other hand, because the root state is frequently visited in the AC matching and usually has a large number of next states, a root-index technique is applied to advance multiple bytes in one single matching.

In Section 3.1, we introduce the algorithm of FSAM to obtain its overall image. In Section 3.2 and 3.3, the detailed algorithms of the prehash and root-index matching are formally described. In the last section, the parallel architecture of FSAM is proposed for its feasibility.

3.1 Algorithm of FSAM

The algorithm of FSAM can be described as consisting of a preprocessing phase and a searching phase. The preprocessing phase produces the required data structure and data for further processing in the searching phase. The sequential matching flow in the searching phase is logically presented in Figure 4(a). If the current state is the root state, root-index is applied for multiple-byte matching to obtain the next state; otherwise, prehash is used. If the result of the prehash matching reports a “no-hit,” a true negative is indicated and the



(a)

```
Preprocessing(P) {
  S ← Build − AC(P)
  V ← Build − Pre − Hash(S)
  IDX[1..k_root] ← Build − Root − Index(S[0..k_root])
  NEXT ← Build − Root − Next(IDX[1..k_root])
}.
```

(b)

```
Searching(T, S){
S_c ← S_0
 For 1 ≤ i ≤ |T| {
    w ← T[i..(i + k_pre−hash)]
    If S_c = S_0 {
       z ← T[i..(i + k_root)]
       S_c ← Match − Root − Index(z)
       i ← i + k_root
    }
    Else {
       V_c ← Load − Bit − Vector(S_c)
       If Match − Pre − Hash(w, V_c) = no − hit
          S_c ← S_0
       Else {
          S_c ← Match − AC(S_c, T[i])
          i ← i + 1
       }
     }
    }
 }.
```

(c)

Fig. 4. Sequential algorithm of FSAM, (a) Logical matching flow in the searching phrase, (b) Function of the preprocessing phase, (c) Function of the searching phase.

state immediately returns to the root state, implying that the slow AC matching can be avoided. If a "hit" occurs, the AC matching is definitely required to obtain the next state. The pseudocodes in the preprocessing phase and the searching phase are described in the following text. The functions and their parameters will be further explained in the next two sections.

1. *Preprocessing phase:* The function $Preprocessing(P)$ does the preprocessing phase of FSAM and is written in Figure 4(b). First of all, *Preprocessing* () translates all the patterns P into the states S of the AC tree using the conventional AC function *Build - AC* (). After S is obtained, *Preprocessing* () then builds all bit vectors V by the function *Build - Pre - Hash* (), and generates the multiple root-index tables $IDX_{[1..k_{root}]}$ and the root next table *NEXT* by *Build - Root - Index* (S) and $Build - Root - Next\,(IDX_{[1..k_{root}]})$, respectively.

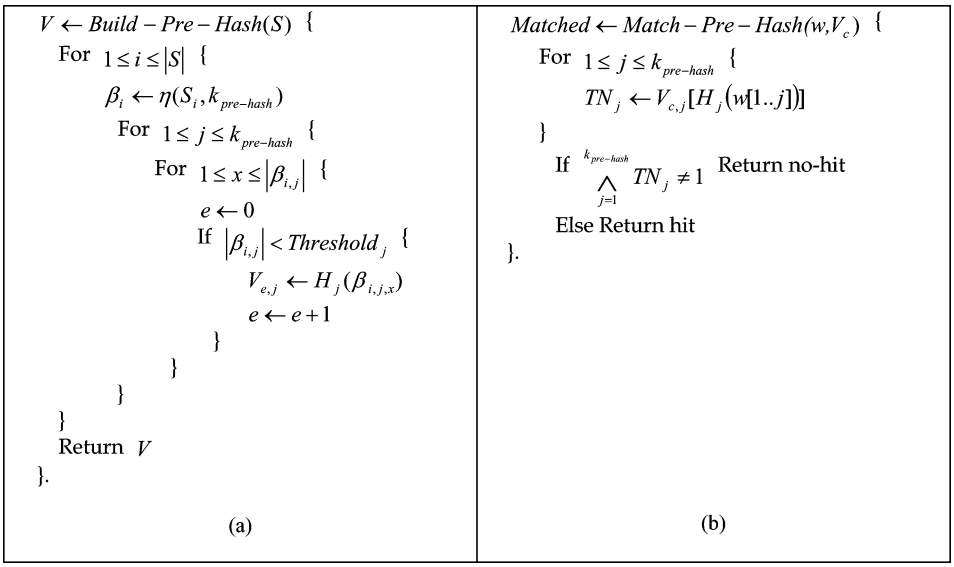


Fig. 5. Prehash-matching algorithm (a) Build the bit vector function in the preprocessing phase (b) Matching function in the searching phase.

2. *Searching phase:* The function searching $Searching(T, S)$ for FSAM is invoked in this phase and is described as in Figure 4(c) where T denotes the input text. Initially, current state S_c is set to root state S_0. In each matching loop for $1 \le i \le |T|$, if the current state S_c is the root state S_0, root index can be used to accelerate the performance by using $Match - Root - Index(z)$, where the substring of the text $T[i..(i+k_{root})]$ is set as z for root-index matching and it can advance k_{root} characters. Otherwise, $Searching()$ loads the current bit vector V_c for S_c and sets the substring of text $T[i..(i+k_{pre-hash})]$ to w for prehash matching, where i is the current matching position of the text. Then, $Match - Pre - Hash(w, S_c, V_c)$ tests whether w has a hashing hit in V_c or not. If it returns $True$, $Searching()$ must continue the original AC matching using $Match - AC(S_c, T[i])$ to match a single character.

3.2 Prehash Matching

The prehash method can quickly test the multiple partial patterns of the current state against the compared substring of text to avoid consequent slow AC matching. The AC matching can be skipped if a true negative is indicated in the prehash matching. True negative is the condition where the compared substring of text is absent in the prehash vector for the suffixes of the current state.

The prehash algorithm can be described as in Figure 5. S is the set of all AC states and $|S|$ is the number of states, built by the conventional AC algorithm from a set of multiple patterns P. Let $\beta_{i,j}$ be the set of suffixes of length j for state S_i, and $\beta_{i,j,x}$ represents the xth suffixes in length j for state S_i. A

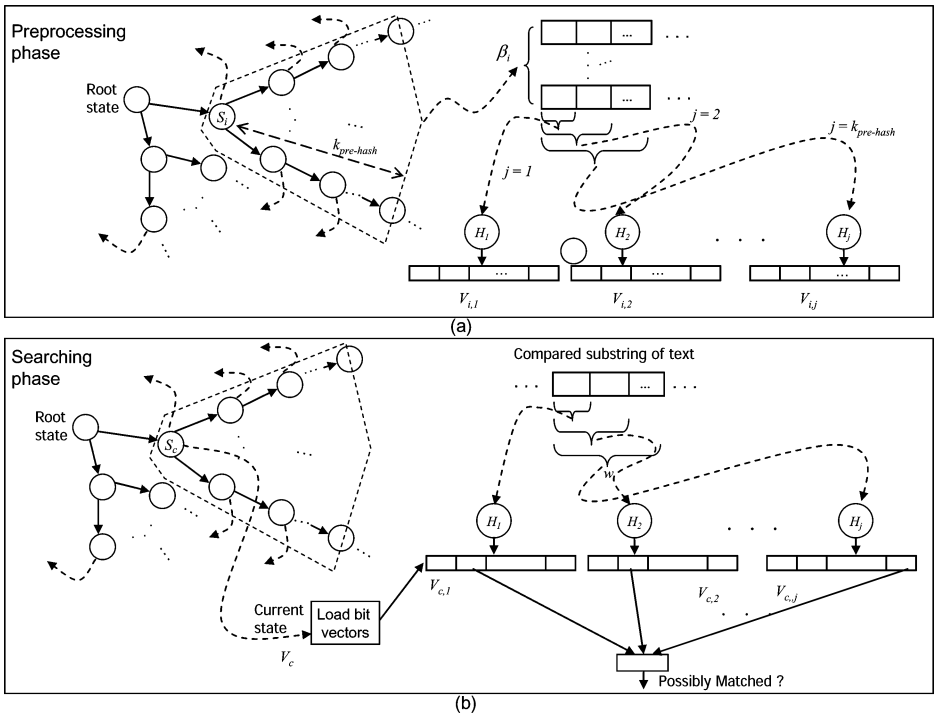


Fig. 6. Prehash matching for state S_i (a) Building the bit vector in the preprocessing phase (b) Load bit vector and compare text in the searching phase.

transition function η can collect the possible $\beta_{i,j}$ from S_i to the states with length j.

$Build - Pre - Hash\,(S)$builds the prehash bit vector in the preprocessing phase as shown in Figure 5(a). This function initially inputs the AC tree, which is built by the conventional AC algorithm. Then it extracts suffixes β_i within the length $k_{pre-hash}$ for the specific state S_i by using $\eta(S_i, k_{pre-hash})$, where $k_{pre-hash}$ is maximum length of prehash suffixes and also the length of the substring in text for each prehash matching. β_i also includes the failure links in the AC tree. When suffixes are obtained, the prehash algorithm hashes suffixes into bit vectors by $V_{i,j} \leftarrow H_j(\beta_{i,j,x})$, where H_j is a hashing function for the corresponding bit vector $V_{e,j}$ and the same H_j is used for all states. This procedure of building the bit vectors is also illustrated in Figure 6(a).

In the searching phase, prehash performs $Match - Pre - Hash\,(w, V_c)$ to rapidly match for the current state in the AC tree as shown in Figure 5(b), where w is the current compared substring of the text and V_c is the current bit vector. The operation $TN_j \leftarrow V_{c,j}[H_j\,(w[1..j])]$ looks up the bit value of the position $H_j\,(w[1..j])$ in $V_{c,j}$, in order to return true-negative TN_j for length j. TN_j is 1 (True) if the hashed $w[1..j]$ bit is set in $V_{c,j}$. The prehash matching return no-hit when $\overset{k_{pre-hash}}{\underset{j=1}{\wedge}} TN_j \neq 1$, where the operation $\overset{k_{pre-hash}}{\underset{j=1}{\wedge}}$ is an AND operation for multiple TN_j, which amount is $k_{pre-hash}$.

A $Threshold_j$ parameter is used to limit the space of bit vectors for the states, which has more suffixes than $Threshold_j$. If $|\beta_{i,j}|$ is smaller than $Threshold_j$, $\beta_{i,j,x}$ will be hashed into $V_{e,j}$ by the hashing function H_j. Otherwise, no bit vector will be built for this state.

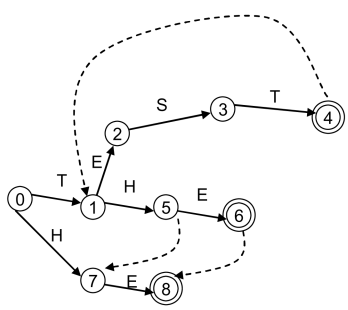
The diagram of Figure 6(b) illustrates this searching process wherein the matching unit loads the current bit vectors V_c, then performs $TN_j \leftarrow V_{c,j}[H_j\,(w[1..j])]$ operation to test whether each $w[1..j]$ is true negative or not.

To clearly explain the previous prehash algorithm, an example for extracting the suffixes from the AC tree is shown in Figure 7(a). The AC tree and the suffixes table are built for the patterns "TEST," "THE," and "HE." The suffixes are the possible transition paths and include the failure links, which are denoted as dash lines. After the suffixes are extracted, the bit vector for each state can be generated. Figure 7(b) plots the suffixes and bit vectors, which are generated from Figure 7(a). With referring a related hashing article [Erdogan et al. 2006], if the proper masking bits are selected, the mask-hashing function has the fastest speed, the smallest circuit, and the similar performance to other hashing functions. In fact, it can provide the satisfactory results of uniform hashing, the required condition of Equation (1). Thus, in generating the bit vectors, we use the adjustable mask-hashing function that allows selecting the masking position for different sizes of bit vectors to achieve a more uniform distribution in the preprocessing phase. For this simplified example, the masking position is selected to be the rightmost 3 bits of the characters. When conversion from binary to one-hot representation (each bit represents a binary number) is used as the hashing function, the 00100001 and 00101000 bit vectors are generated from {01000101, 01001000} and {01000101_01010011, 01001000_01000101}, respectively. For instance, Figure 7(c) depicts that state 4 has suffixes "E" and "H" of length one, and their ASCII codes are {01000**101**, 01001**000**} in the binary format, and thus its bit vector of depth one is {00100001}.

Figure 7(d) shows an example for state 4 in the suffix matching. The prehash unit reads a 2-byte substring and then hashes the length 1 substring "A" and length 2 substring "AB" with H_1 and H_2 in parallel, respectively. When the prehash unit indicates any no-hit for H_1 and H_2, which means substring "A" and "AB" has no any possibility to match patterns, the current state will transit to the root state and then root-index is performed to match multiple characters in a single matching. Note that returning to the root state by prehash is different from the conventional AC failure transition, since the former provides a faster state transition, by using the hashing technique.

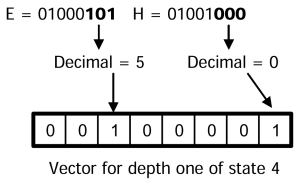
Although our prehash idea is motivated by BFSM, there are two main differences between BFSM and our prehash.

1. Since BFSM requires multiple Bloom filters and builds the bit vector of each Bloom filter from all patterns, it requires a large memory and multiple memory accesses for the bit vectors. Therefore, BFSM makes implementing the bit vectors impractical by using either a register or SRAM. However, our approach builds the distinct bit vectors from the suffixes of each state S_i only. The number of suffixes is quite small, which makes implementing



State	Suffixes		Binary (length 1 value, length 2 value)	Bit Vectors
	length 1	length 2		
0	T	TE	01010**100**, 01010100_ 01000**101**	00010001,00100001
	H	TH	01001**000**, 01010100_ 01001**000**	
		HE	, 01001000_ 01000**101**	
1	E	ES	01000**101**, 01000101_ 01010**011**	00100001,00101000
	H	HE	01001**000**, 01001000_ 01000**101**	
2	S	ST	01010**011**, 01010011_ 01010**100**	00001000,00010000
3	T		01010**100**	00010000,00000000
4	E	ES	01000**101**, 01000101_01010**011**	00100001,00101000
	H	HE	01001**000**, 01001000_01000**101**	
5	E		01000**101**	00100000,00000000
6				00000000,00000000
7	E		01000**101**	00100000,00000000
8				00000000,00000000

(b)



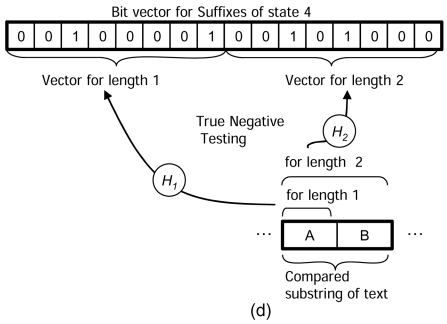


Fig. 7. Prehash example, (a) AC tree, (b) Suffixes and bit vectors for all states in the AC tree, (c) Mask hashing function for prehash, (d) Prehash matching for state 4.

the bit vectors more feasible. Actually, the prehash matching requires a very small bit vector, that is, about 8 to 32 bits.

2. Since more hashing functions will set more bits to 1 in a bit vector, BFSM employing multiple hashing functions can reduce the probability of false positive only and thus cannot reduce the amount of subsequent exact matching. Our approach intends to improve the probability of true negative by ascertaining the unmatching suffixes. Hence, using one hashing function for each bit vector is sufficient and can significantly reduce the hardware cost and latency. The probability of true negative P_{tn} is adapted from (1) as

$$P_{tn} = \left(1 - \frac{1}{M}\right)^{|\beta|}, \tag{2}$$

where $|\beta|$ is the number of suffixes, and M is the size of the bit vector. This equation holds under the same assumption as Equation (1).

3.3 Root-Index Matching

Root-index can match multiple characters of the text at the same time. When the prehash result is a no-hit (true negative), the matching transition will

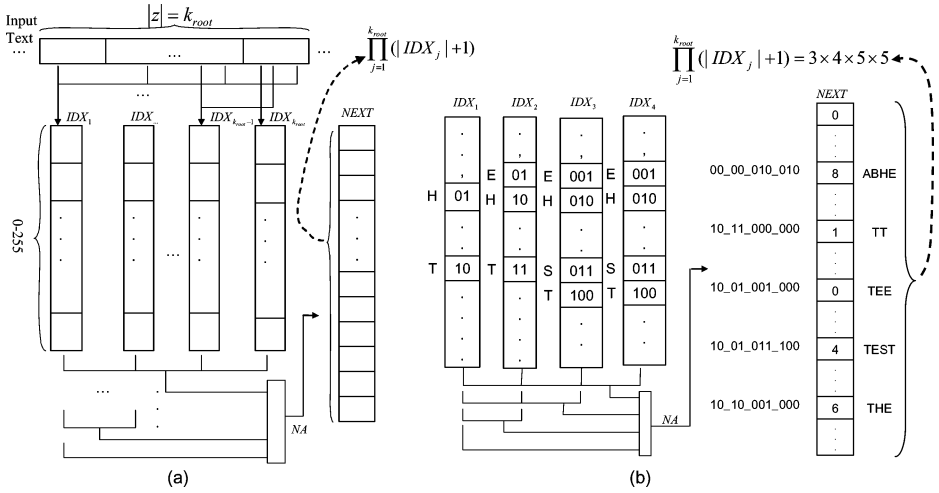


Fig. 8. (a) Root-index architecture, (b) A root-index example for matching the texts "ABHE," "TT," "TEE," "TEST," and "THE" with the patterns "TEST," "THE," and "HE."

return to the root state to perform the root-index matching. Since most bytes of the text will visit the root state, the root-index technique is worth using. In fact, the root-index matching is a compressed technique for matching in the automaton, which generates $2^{k_{root}}$ next states for matching the substrings of length k_{root}.

In Figure 8(a), root index comprises k_{root} root-index tables $IDX_{[1..k_{root}]}$ and a root-next table *NEXT*, where k_{root} denotes the maximum length of the root-index matching. Each entry of *IDX* stores a partial address for locating the next state in *NEXT*, where the partial address is a sequential integer to represent the order of appearing characters for the corresponding substrings in the suffixes of the root state. Note that, for advancing k_{root} characters in one matching iteration, the substring begins from the current byte to k_{root}, meaning the later *IDX* table requires including the entry of the former *IDX* tables. The width of each *IDX* is equal to the number of characters appearing in the binary format.

The *NEXT* table is used to store the next state addresses of the states within length k_{root}, counting from the root state S_0. *NEXT* is indexed by a concatenation address of lookup value from the *IDX* tables. Thus, the number of *NEXT* entries is equal to $\prod_{j=1}^{k_{root}} (|IDX_j| + 1)$, which is the product of the numbers of nonzero entries adding one zero entry in each root-index table, where $|IDX_j|$ denotes the number of alphabets that appeared in the jth index table.

In the preprocessing of root-index, $Build - Root - Index(S)$ is first invoked to build $IDX_{[1..k_{root}]}$ as Figure 9(a). The length of input text and the number of *IDX* tables are equal to k_{root}. This function builds the *IDX* table from IDX_1 to $IDX_{k_{root}}$. It first performs $IDX_j[x] \leftarrow 0$ to initialize the current *IDX* table and performs $IDX_j[x] \leftarrow IDX_{j-1}[x]$ to bring the IDX_{j-1} to the IDX_j, and finally performs $IDX_j[\alpha_j[x]] \leftarrow \rho$ to set the index value from the current character of

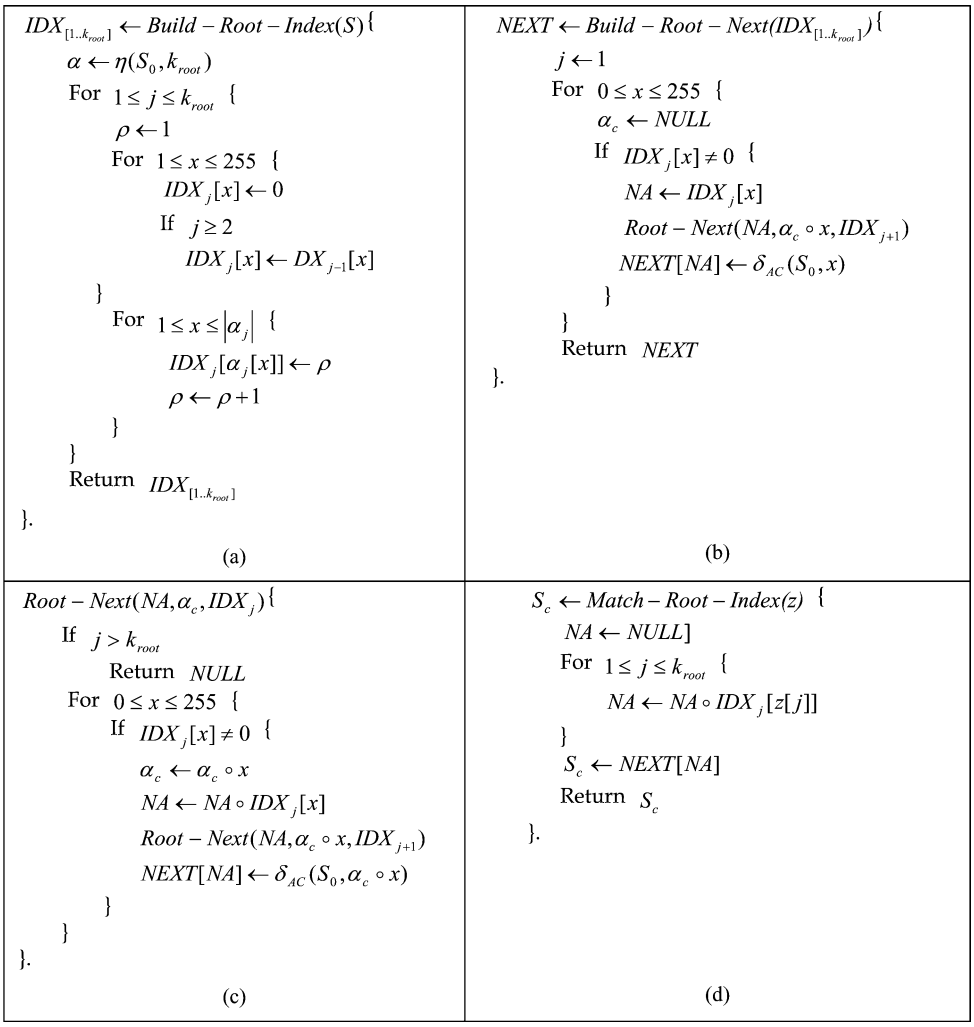


Fig. 9. Root-index algorithm, (a) Function of building the root-index tables, (b) Function of building the root next table, (c) Assisted recursive function of building the root-next table, (d) Function of performing the root-index matching.

the suffixes. α comprises the suffixes of S_0, which is a set of possible transition paths from root state S_0 to the states within length k_{root} and can be defined as $\alpha \leftarrow \eta(S_0, k_{root})$. The xth suffix of length j in α will be indexed into the entry by $IDX_j[\alpha_j[x]]$ and numbered by an increasing value ρ. If the corresponding entry in IDX_j is appearing in suffix $\alpha_j[x]$, ρ will be put into that entry and increased by 1.

After root-index tables are built, a root-next table *NEXT* for the root state is required to be built using the function $Build - Root - Next(IDX_{[1..k_{root}]})$ as shown in Figure 9(b). *NEXT* stores all next states within length k_{root}. The entry of *NEXT* is accessed using the next address *NA*, which is a concatenation of

all lookup values in $IDX_{[1..k_{root}]}$. Therefore, a recursive $Root - Next\,(NA, \alpha_c \circ x, IDX_{j+1})$ is used to concatenate NA from the deeper root-index table IDX_{j+1}, where α_c is temporary storage for current suffix, symbol $\circ$ is a concatenation operation, and x is the character of suffix to index the entries of IDX. IDX_{j+1} is indexed by supplying the deeper $(j+1)$th byte of suffixes.

The recursive $Root - Next\,(NA, \alpha_c, IDX_j)$ can be written as Figure 9(c). After NA is obtained, the entry of the root-next state *NEXT[NA]* will be set using the function of the conventional AC transition $\delta_{AC}(S_0, \alpha_c \circ x)$, which can move to the new next state from the root state by supplying a suffix.

In the matching phase of root-index, $Match - Root - Index(z)$ inputs a substring of the text z to locate the new state S_c, and it is defined as Figure 9(d). The lookup operation inputs $z[j]$ into $IDX_j(z[j])$ to generate a NA, repeatedly, which is defined as $NA \leftarrow NA \circ IDX_j[z[j]]$. When NA is obtained, S_c is then lookup by $NEXT[NA]$.

An example of root index is illustrated in Figure 8(b). When patterns are "TEST," "THE," and "HE," IDX_1 to IDX_4 will at least contain the characters appearing in the corresponding position as {"H,""T"}, {"E,""H"}, {"E,""S"}, and {"T"}, respectively. However, suffixes of text might be in the prefixes of patterns, thus the later tables must contain the entries of the former tables, leading that IDX_1 to IDX_4 actually contain {"H,""T"}, {"E,""H,""T"}, {"E,"H,""S,"T"}, and {"E","H","S","T"}, respectively. For numbering the entries of IDX tables, the third and fourth IDX have four appearing characters. Thus, "H,""E,""S," and "T" are numbered as "001," "010," "011," and "100" in the binary format, respectively. The other nonnumbered entries will be filled with zero.

In the matching phase, 00_00_010_010, 10_11_000_000, 10_01_001_000, 10_01_011_100, and 10_10_001_000 are NAs to locate S_c as next states 8, 1, 0, 4, 6 for the texts "ABHE," "TT," "TEE," "TEST," and "THE," respectively. For instance, root index can lookup $IDX_1[T] \circ IDX_2[E] \circ IDX_3[S] \circ IDX_4[T]$ to obtain 10_01_011_100 to locate the text "TEST". Note that the zero value of IDX_j is mapped into the entry of the symbol $(\sim)$, which is a termination symbol for the length of z is shorter than k_{root}. For example, NA of the text "TEE" is not $IDX_1[T] \circ IDX_2[E] \circ IDX_3[E]$, but $IDX_1[T] \circ IDX_2[E] \circ IDX_3[E] \circ IDX_4[\sim]$.

3.4 System Architecture

Different from the sequential algorithm in Section 3.1, a preferred parallel architecture for the FSAM coprocessor is suggested in Figure 10. Three independent matching units—the prehash matching, the root-index matching, and the bitmap AC matching—simultaneously perform $Match - Root - Index\,()$, $Match - Pre - Hash\,()$, and $Match - AC\,()$, respectively, which are described in the sequential algorithm. Hence, a control logic coordinates these units for parallel processing and each matching function has its individual memory interface to access its preprocessing data. Since the design methodologies for System-on-Chip (SOC) have become popular and well developed in recent times, the use of such a component in modern IC technology is quite feasible.

In the FSAM coprocessor, the three units can read the text in different lengths and perform their matching concurrently. This example in the FSAM

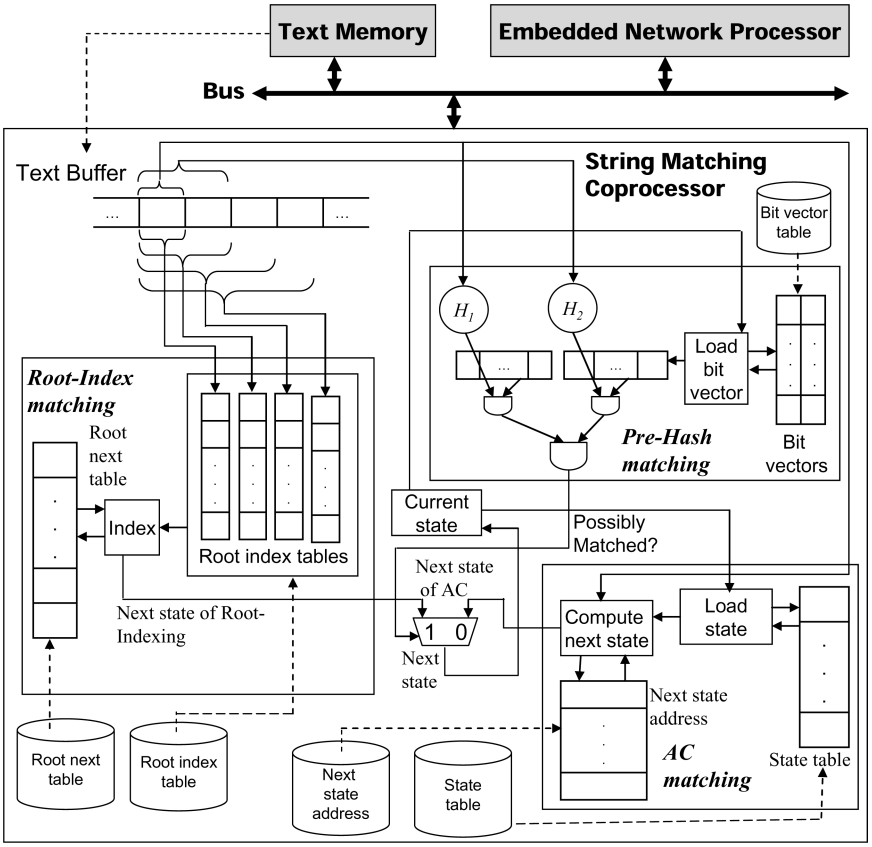


Fig. 10. The parallel architecture of the FSAM coprocessor.

coprocessor processes a 1-byte substring for AC matching, a 2-byte substring for prehash matching and a 4-byte substring for the root-index matching in a single matching iteration. The root index and bitmap AC are used to locate the next states, and the prehash matching is used to decide which next state is to be used in the next matching iteration.

In addition to the original-state and next-state address tables, the root-index matching requires the root-index tables and a root-next table, while prehash needs a bit vector table for their pattern storage. For the flexibility of using storage, these tables can be stored in either internal or external memories. The detailed discussion is described in Section 5.1.

For the performance, the memory access requires two clock cycles as in the case of bitmap AC. For per byte processing, prehash is three cycles, composed of two cycles for loading a 4-byte bit vector and one cycle for the hashing operations. Root-index requires four cycles, composed of two cycles for processing index codes and two cycles for processing the root-next table. Since the sizes of the index tables are fixed and small, it is feasible to implement them as multiple banks memory. Thus, the time required for accessing the index tables can

be less than two cycles and root-index takes two cycles for loading the 32-bit root-next state address because the state table is stored in SRAM.

4. EVALUATIONS

This section intends to evaluate the performance and space requirement of our FSAM. In the first subsection, we formally derive the time and space requirement of FSAM, as well as the probability of using prehash and root index. Then, we use the real URL and virus patterns to show their results in using FSAM in Section 4.2. To demonstrate more realistic results, the evaluation of real network traffic is investigated in the last Section.

4.1 Formal Analysis

If prehash, root-index, and bitmap AC are run using the sequential algorithm in Section 3.1, the average time for our FSAM is

$$T_{avg\_time} = \frac{T_{hash} + P_{root} \times T_{root} + (1 - P_{root}) \times T_{AC}}{(k_{root} \times P_{root}) + (1 - P_{root})}, \tag{3}$$

where T_{avg_time} is the average time to process a byte, T_{hash} is the prehash matching time, T_{root} is the root index matching time, T_{AC} is the AC matching time, and P_{root} is the probability of using the root-index matching,

However, in Figure 10, prehash, root index, and AC can be performed in parallel and the computation of the next states in these three units are independent. Thus, the average time can be reduced to

$$T_{avg\_time} = \frac{P_{root} \times T_{root} + (1 - P_{root}) \times T_{AC}}{(k_{root} \times P_{root}) + (1 - P_{root})}. \tag{4}$$

Since the AC matching is the critical path, the worst-case time of FSAM is equal to T_{AC} as

$$T_{worst\_time} = T_{AC}. \tag{5}$$

The probability P_{root} is an average probability that the root-index matching is performed and calculated by

$$P_{root} = 1 - \prod_{j=1}^{k_{pre-hash}} (1 - P_{tn\_j} \times TH_j), \tag{6}$$

where $P_{tn_j}$is the probability of true negative for the suffixes of length j, and TH_j is the ratio of states in which the number of suffixes of length j is less than $Threshold_j$. As stated before, a large number of suffixes will require a big bit vector. Thus, a $Threshold_j$ parameter is applied to limit the rapid growth of the bit vector size. TH_j can be obtained as

$$TH_j = \frac{N_j}{|S|}, \tag{7}$$

where N_j is the number of states in which the number of suffixes is less than $Threshold_j$, and $|S|$ is the number of states in the AC tree. Note that Equation

P_{root}	$K_{pre\text{-}hash}$			
P_{tn}	1	2	3	4
0.1	0.10	0.19	0.27	0.34
0.2	0.20	0.36	0.49	0.59
0.3	0.30	0.51	0.66	0.76
0.4	0.40	0.64	0.78	0.87
0.5	0.50	0.75	0.88	0.94
0.6	0.60	0.84	0.94	0.97
0.7	0.70	0.91	0.97	0.99
0.8	0.80	0.96	0.99	1.00
0.9	0.90	0.99	1.00	1.00

(a)

M	$\lvert\beta\rvert$							
P_{tn}	2	4	8	16	32	64	128	256
0.1	1	2	4	7	14	28	56	112
0.2	2	3	5	10	20	40	80	160
0.3	2	4	7	14	27	54	107	213
0.4	3	5	9	18	35	70	140	280
0.5	3	6	12	24	47	93	185	370
0.6	4	8	16	32	63	126	251	502
0.7	6	12	23	45	90	180	359	718
0.8	9	18	36	72	144	287	574	1148
0.9	19	38	76	152	304	608	1215	2430

(b)

Fig. 11. (a) P_{root} versus P_{tn} from 0.1 to 0.9 and $k_{pre-hash}$ from 1 to 4, (b) The size of bit vectors M versus P_{tn} from 0.1 to 0.9 and $|\beta|$ from 2 to 256.

(6) holds under the assumption that each nonroot state has the equal probability of being visited, because of the calculation of TH_j.

From observing Equation (6), P_{root} is influenced by two parameters, P_{tn} and $k_{pre-hash}$. In Figure 11(a), we present the effect of these two parameters by using Equation (6). TH_j is assumed to be 1. It obviously shows that under moderate P_{tn}, even small $k_{pre-hash}$ still achieves acceptable P_{root} ($P_{root} > 0.5$). Therefore, setting the maximum suffix length $k_{pre-hash}$ to 2 is sufficient. For example, when P_{tn} is set to 0.6 and $k_{pre-hash}$ is set to 2, P_{root} is equal to 0.84.

For the space evaluation, we first need to determine the size of bit vectors M. Since the probability of true negative is defined in Equation (2), M can be determined by given $|\beta|$ and P_{tn} as

$$M = \frac{1}{1 - p_{tn}^{\frac{1}{|\beta|}}}. \tag{8}$$

Figure 11(b) shows that M increases significantly as $|\beta|$ and P_{tn} grow, and thus, M is feasible under small $|\beta|$ and moderate P_{tn}.

The space requirement can be determined by summing the bitmap AC space $Size_{AC}$, the prehash bit vector space $Size_{pre-hash}$, and the root-index space $Size_{root}$, that is,

$$Size_{total} = Size_{AC} + Size_{root} + Size_{pre-hash}. \tag{9}$$

The original space requirement of bitmap AC, $Size_{AC}$ is mainly dominated by the state table, which is equal to the number of states $|S|$ multiplied by the state size $Size_{state}$,

$$Size_{AC} = |S| \times Size_{state}. \tag{10}$$

Each state size $Size_{state}$ includes the 1-byte state information, the failure and next state address $Size_{state\_address}$, as well as the size of bitmap $Size_{bitmap}$ for locating the next state. Hence, $Size_{state}$ can be determined by

$$Size_{state} = 1 + Size_{state\_address} \times 2 + Size_{bitmap}. \tag{11}$$

The prehash size $Size_{pre-hash}$ is determined from $\sum_{j=1}^{k_{pre-hash}} M_j$, which is the size of all bit vectors for one state, where M_j is a bit vector size for length j and $k_{pre-hash}$ is the maximum length of prehash. $|S| \times TH_j$ is the number of states where the number of suffixes is smaller than $Threshold_j$. Thus, $Size_{pre-hash}$ is obtained from

$$Size_{pre-hash} = \sum_{j=1}^{k_{pre-hash}} M_j \times |S| \times TH_j. \tag{12}$$

$Size_{root}$includes all root-index tables and the root-next table as illustrated in Figure 8. The size of all the root-index table is 256 multiplied by k_{root}, and the root-next table is the number of the next-state addresses multiplied by the state address size $Size_{state\_address}$. The number of root-next state addresses is the cross product of the numbers of appearing alphabets in the index tables IDX_j and one zero entry. Then $Size_{root}$ is formulated as

$$Size_{root} = 256 \times k_{root} + \prod_{j=1}^{k_{root}} (|IDX_j| + 1) \times Size_{state\_address}. \tag{13}$$

4.2 Evaluation of Real Patterns

Section 4.1 provides the formal analysis of time and space requirements. However, some parameters, such as TH_j and k_{root}, depend on the profile of patterns. Thus, in this section, we choose the URL blacklists and virus signatures from http://www.squidguard.org/blacklist/ and http://www.clamav.net, respectively. Since the URL blacklists and virus signatures contain many patterns as well as long patterns, such patterns are sufficient to evaluate the performance of our FSAM.

In this evaluation, we first obtain two statistics, namely, the suffix counting and the index counting for real patterns. The suffix counting counts the number of suffixes with a specific length for each state and is used to compute TH_j. The index counting counts appearing alphabets for each length in the root state and is used to determine k_{root}.

The analyzed URL blacklists contain 21,302 patterns and generate 194,096 states, while the virus signatures contain 10,000 patterns and generate 402,173 states. Figure 12 shows the ratio of states for the range of the suffix counting, aiming the suffixes of length 1 and 2 to give a proper $Threshold_j$ in Equation (7). Figures 12(a) and 12(b) show that, when $Threshold_j$ is set to 8 for length 1, the URL and virus patterns have 68% and 49% states using the prehash matching, respectively. For length 2, Figures 12(c) and 12(d) show 41% and 32% states for the URL and virus patterns, respectively. These results show that most states of the URL and virus patterns only have a few numbers of suffixes, so that the prehash approach is useful in reducing the matching time.

For the index counting of the root state, the URL patterns generate 36, 38, 38, and 38 states while the virus patterns generate 256, 256, 256, and 256 states. Since the virus patterns are nonalphabet binary values, they have a higher

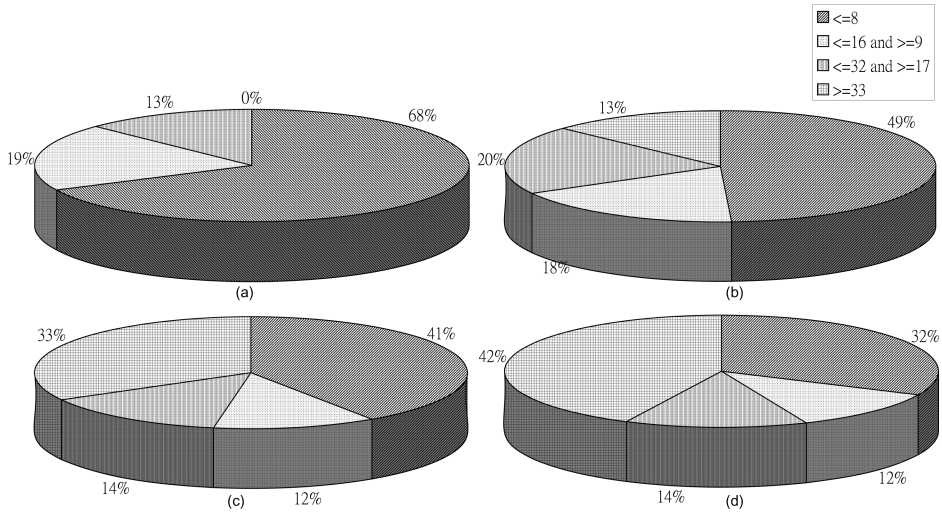


Fig. 12. The ratio of states (TH_j) for the range of the number of suffixes, (a) Length 1 counting for URL patterns, (b) Length 1 counting for virus patterns, (c) Length 2 counting for URL patterns, (d) Length 2 counting for virus patterns.

index counting than the URL patterns in our analysis. These results show that the URL patterns can use the longer suffixes than virus patterns for the root state in the root-index matching. To avoid the large space requirement, k_{root} is set to 4 and 2 for the URL patterns and virus patterns, respectively.

Using the previous equations and the statistic results of real patterns, the time and space can be computed for the URL and virus patterns. We obtain 38 cycles for T_{AC} from Section 2.1, 3 cycles for T_{hash}, and 4 cycles for T_{root} from Section 3.4. We set $Threshold_j$ to 8 and have 16-bit vector size for each length of suffixes. From Section 4.1, P_{tn_1} and P_{tn_2} are at least 0.6 according to Figure 11(b) because all $|\beta|<Threshold_j$. Also from the previously described statistics of real patterns, $TH_1 = 0.68$ and $TH_2 = 0.41$ for the URL patterns. Similarly, $TH_1 = 0.49$ and $TH_2 = 0.32$ for the virus patterns, respectively. With the previously-mentioned parameters, when $k_{pre-hash}$ is 2 and k_{root} is 4, according to Equation (6), the average probability of true negative is $P_{root} = 0.56$, and the parallel average time is obtained by Equation (4) as $T_{avg\_time} = 7.07$ cycles per byte.

For the case of the virus patterns, when $k_{pre-hash}$ and k_{root} are both 2, and the other parameters are same as the previously mentioned setting, the probability of root-index matching is computed as $P_{root} = 0.43$. Also we obtain $T_{avg\_time} = 16.3$ cycles per byte.

From these results, bitmap AC requires 38 cycles for one character matching. Thus, our approach is 537% and 233% faster than bitmap AC for the URL and virus patterns, respectively.

For the space requirements, we can obtain the following parameters from indexing counting of the URL patterns as follows: $|S|$ is 194,096, RN_1 is 36, RN_2 is 38, RN_3 is 38, and RN_4 is 38 .

Then, by using Equations (10), (11), (12), and (13), we can obtain $Size_{State} =$ 37 bytes, $Size_{AC} = 6.85$MB, $Size_{pre-hash} = 3.23$MB, and $Size_{root} = 7.5$MB. By summing the these results, the space requirement of the URL patterns is $Size_{total} = 17.58$MB.

For the virus patterns, most of the computation parameters are the same as those of the URL patterns. In order to avoid the large size in building root-index data, only the length 1 and length 2 matching are used for root-index matching. According to the virus patterns analysis, $|S|$ is 402,173, k_{root} is 2, RN_1 is 256, and RN_2 is 256. Then, we can obtain $Size_{state} = 37$ bytes, $Size_{AC} = 14.19$MB, $Size_{pre-hash} = 4.97$MB, and $Size_{root} = 0.25$MB. Finally, the total space for the virus patterns is computed to be $Size_{total} = 19.41$MB.

From the analysis of real patterns, although $Size_{total}$ of the URL and virus patterns are larger than the original size $Size_{AC}$ by 10.73MB and 5.22MB, respectively, the space requirements of the URL and virus patterns are acceptable for modern content-filtering systems because the high capacity memories are now becoming steadily cheaper.

4.3 Evaluation of Real Network Traffic

As mentioned previously, Equation (6) holds under the assumption that each nonroot state has the equal probability of being visited. However, this assumption may not be true under the real input text. Some strings in the text may frequently occur, causing some states are usually visited. Thus, we use the real network traffic to directly measure the value of P_{root}, rather than calculating from Equation (6). The Google website and the ethereal captured data consisting of over 100MB and 120MB, respectively, are selected as the texts used to evaluate the previously mentiones URL and virus patterns, respectively. Since these large data already contain diverse types of network traffic, using them should have a representative for the performance evaluation. Note in this experiment, we use the adjustable mask-hashing function to provide a significantly uniform distribution of hitting in the bit vectors.

Our experiment shows that when the Google website as the input text for the URL patterns, P_{root} is 0.59, which is slight larger than 0.56, the value in Section 4.2. When the ethereal captured data as the input text for the virus patterns, P_{root} is 0.49, which is also larger than 0.43, the value in Section 4.2. The differences between them are caused by two points: (1) Since *Threshold* is an upper bound for the number of suffixes, many states having less $|\beta|$ will own the higher probability of true negative P_{tn}. Thus, the value of 0.56 for the URL patterns and 0.43 for the virus patterns are conservative. (2) The probability of visiting the states with high P_{tn} is larger. However, the effect of the second point completely depends on the profile of the input text.

5. HARDWARE IMPLEMENTATION AND PERFORMANCE COMPARISON

In this section, Section 5.1 gives the detailed description of hardware implementation, including the block diagram, finite state machine, components, and interface of the FSAM hardware. Section 5.2 gives an exhausted comparison with previous hardware implementations.

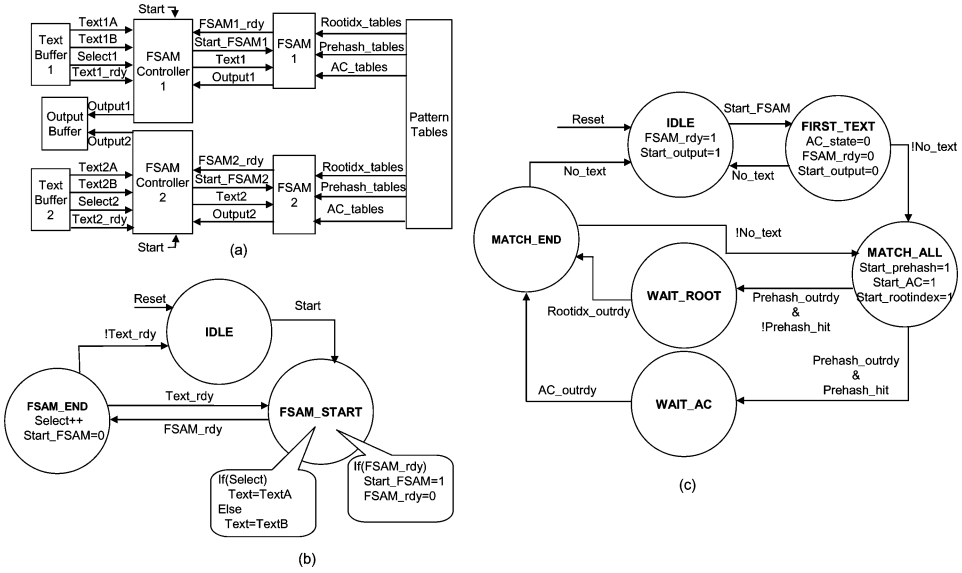


Fig. 13. FPGA Implementation of the double-engine FSAM, (a) block diagram of the double-engine architecture, (b) finite state machine for the FSAM controller, (c) finite state machine for FSAM.

5.1 Hardware Implementation

Figure 13 illustrates the FPGA implementation of the double-engine FSAM, and includes (a) a block diagram of the hardware architecture, (b) a finite state machine for the FSAM controller, and (c) a finite state machine for the FSAM.

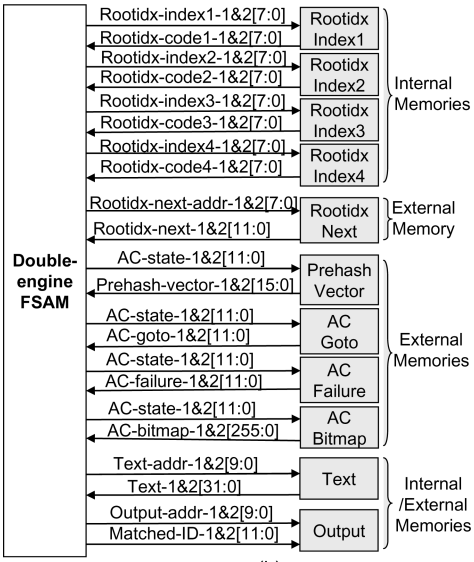
In the double-engine FSAM, two FSAMs perform their matching for different texts at the same time, and thus, they perform independently without affecting the other. That is, FSAM1 and FSAM2 have their own texts, Text1 and Text2, respectively. Also the ping-pong buffers are used for each FSAM. For instance, the "Select1" signal is used to choose either the Text1A or Text1B buffers as Text1. At the ping-pong buffers, one buffer is used during the matching and the other buffer is prepared concurrently by the processor or DMA.

As shown in Figure 13(b), the controller feeds the text to the corresponding FSAM and activate it via the Start_FSAM signal at the FSAM_START state. If the FSAM_rdy signal is one, representing that the FSAM is idle, the controller will set the Start_FSAM to one to proceed a new matching process and disable the FSAM_rdy. Once the matching process finishes, the FSAM sets the FSAM_rdy to one and sends this signal to the FSAM controller. In this case, the FSAM control transits from the state FSAM_START to FSAM_END, the end of the matching operation. The Select signal switches between 0 and 1 in the FSAM_END state to obtain the alternative text in ping-pong buffers.

This default unit of text handled in the double-engine FSAM is a message. However, when the granularity is a packet, rather than a message, our FSAM can still operate well with little modification. The method is keeping the last

FSAM Components	Circuit Size (LE)	Memory Size (KB)	Data Width (Bits)
FSAM Controller	10 x 2	-	-
FSAM Finite State Machine	20 x 2	-	-
AC	138 x 2	369.6	Goto : 12 Failure : 12 Bitmap : 256
Root-index	102 x 2	28.6	Index 1: 8 Index 2: 8 Index 3: 8 Index 4: 8 Next : 12
Pre-hash	36 x 2	9.7	Prehash : 16
Text Manager	12 x 2	5.8	Text : 32
Output Manager	11 x 2	1	Output : 12

(a)



(b)

Fig. 14. (a) Components of FSAM Implementation, (b) suggested memory interfaces for the double-engine FSAM.

AC state of the previous packet for the next packet matching, and thus, the FSAM can easily do matching across the multiple packets.

The statistics of the FSAM components in Figure 14(a) reveals the detailed hardware usages. Its circuit size is measured in terms of logic element (LE) counts. The result demonstrates that the root index and prehash modules consume less circuit size, memory size, and bandwidth, as compared to the AC module, representing that AC dominates the hardware cost in FSAM. In the case of the single-engine implementation, its total circuit size is 329 LE only.

For the scalability of the storage, Figure 14(b) shows the suggested memory interfaces for the double-engine FSAM. The suffix "1&2" of the signal symbols denotes the first and second interfaces of each memory bank. Since four root-index tables are very small, storing them into the internal memory is recommended. The text and output memories are implemented as the internal or external memories according to their scales. Finally, since the root-index next table, prehash vector, and bitmap AC related tables could be large for a large amount of patterns, they should be implemented as the external memory for scalability.

For the suggested external memory interface, if the high-speed and high-capacity memories are used, about two cycles with up to 500MHz clock rate are obtainable for the QDR-III SRAMs (www.qdrsram.com). Moreover, since the ASIC hardware can often run at a much higher speed than the FPGA devices, the ASIC implementation, with the external memories for a large set of patterns, is quite feasible to maintain the competitive throughput as our FPGA implementation.

5.2 Performance Comparison

Since many string-matching hardware [Aldwairi et al 2005; Mosola et al. 2003; Baker et al. 2004; Cho et al. 2005; Dharmapurikar et al. 2004] store their patterns in on-chip hardwired circuits and internal memories, we also implemented our FSAM using FPGA internal memories to reach a fair evaluation. Besides, because several previous matching hardware [Aldwairi et al. 2005; Tan et al. 2005; Moscola et al. 2003; Baker et al. 2004; Dharmapurikar et al. 2004, etc.,] employed duplicated hardware for parallel processing, comparing our double-engine architecture with them is still fair in the performance comparison. In particular, our optimally utilized dual port block RAM of Xilinx FPGA not only virtually doubles the performance, but also increases no extra block RAM.

We synthesized FSAM on various Xilinx FPGA devices and compared it with the major types of hardware described in related works, as shown in Table III. The common goals of the hardware are pursuing higher throughput, larger pattern sizes, and smaller circuit size, which are also our concerned factors in this comparison. The pattern size is equal to $Number\_of\_patterns \times Average\_length\_of\_patterns$ and used for evaluating scalability. The throughput is used for measuring performance.

The results demonstrate that FSAM has throughput of 11.1Gbps for the double engines and 5.6Gpbs for the single engine in a Xilinx Virtex2P device. For the storage, FSAM implementation uses an internal memory, Xilinx block RAM, to store the pattern set. Among all matching hardware, our FPGA implementation can handle the largest pattern size of 32,634 bytes, which is the truncated URL patterns and composed of 2,940 patterns with the average length of 11.1 bytes. Thus, our FSAM is superior to all previous string-matching hardware in term of both space requirement and performance.

Next, the pattern placement column shows the major difference between our FSAM and the other matching hardware. The architecture of previous hardware often employed hardwired circuits and internal memories for storing their patterns, thus their amount of patterns was limited by FPGA resources.

6. CONCLUSION AND FUTURE WORKS

In this article, we present a fast and scalable matching automaton (FSAM) with the novel prehash and root-index techniques. The prehash technique is used to quickly verify the text in order to avoid AC matching. The prehash technique has two distinguished enhancements from the previous BFSM. First, Bloom filter uses all patterns to build a big vector, but our approach builds the bit vector from partial patterns (suffixes of the current state). Second, BFSM uses multiple hashing functions, but our approach uses only one hashing function. Therefore, our prehash significantly reduces the hardware complexity and makes the hashing technique more feasible in string-matching. In addition to prehash, our root-index technique is a space-efficient matching technique for matching multiple bytes in one single matching. Since the root state is frequently visited in the string-matching, it is an effective approach to accelerate the automaton.

Table III. Comparisons of String-Matching Hardware

Type	Matching Hardware	Device	Circuit Size (LE)	Pattern Size (Byte)	Throughput (Gbps)[1]	Pattern Placement
AC DFA	FSAM[2]	Virtex2P	656	32,634	11.1	Internal Memory
		Virtex2 1000	322		6.5	
		Virtex2 6000	322		8.9	
		Virtex2 8000	322		7.3	
		Virtex4 xc4vlx80	314		6.8	
		Spartan3 xc3s400	322		5.7	
		VirtexE 2000	2,928		2.1	
		Virtex 800[3]	5,110		1.7	
	Reconfigurable Multi-AC [Aldwairi et al. 2005]	Altera EP20k400E	45,000	3,000	5.0	Internal Memory
	Bit-split AC [Tan et al. 2005]	Xilinx FPGA	N/A[4]	2,048	10.0	Internal Memory
RE DFA	DFA+counter [Lockwood et al. 2001]	VirtextE 1000	98	11	3.8	Hardwired Circuit
	Parallel Regular DFA [Moscola et al. 2003]	VirtexE 2000E	8,134	420	1.2	Internal Memory
KMP DFA	KMP Comparators [Baker et al. 2004]	Xilinx Virtex2P	130	32	2.4	Internal Memory
Comparator NFA	Comparator NFA [Sidhu et al. 2001]	Xilinx Virtex 100	1,920	29	0.5	Hardwired Circuit
	Meta Comparator NFA [Franklin et al. 2002]	Xilinx VirtexE 2000	20,618	8,003	0.4	Hardwired Circuit
Decoder NFA	Decoder NFA [Clark et al. 2002]	Virtex 1000	19,660	17,550	0.8	Hardwired Circuit
	Multi-character decoder NFA [Clark et al. 2003]	Xilinx Virtex2	29,281	17,537	7.3	Hardwired Circuit
	Approximate Decoder NFA [Clark et al. 2004]	Virtex2 6000	6,478	17,537	2.0	Hardwired Circuit
Parallel Comparator	Offset Index Comparators [Cho et al. 2005]	Spartan3 400	1,163	20,800	1.9	Internal Memory
	Discrete Comparators [Sourdis et al. 2003]	Virtex2 6000	76,032	2,457	8.0	Hardwired Circuit
	Pre-decoded CAM Comparators [Sourdis et al. 2004]	Virtex2 6000	64,268	18,032	9.7	Hardwired Circuit
	CAM Comparators [Gokhale 2002]	VirtexE 1000	9,722	640	2.2	CAM[5]

(*continued on next page*)

Table III. (continued)

Hashing	Parallel Bloom Filter [Dharmapurikar et al. 2004]	VirtexE 2000	6,048	9,800	0.6	Internal Memory
	PHmem [Sourdis et al. 2005]	Virtex2 1000	8,115	20,911	2.9	Hardwired and Internal Memory
	Hash-Mem [Papadopoulos et al. 2005]	Virtex2 1000	2,570	18,636	2.0	Internal Memory

1. Throughput is an average performance. The hardware except FSAM and BFSM has the worst-case throughput equal to the average-case throughput.
2. The single-engine FSAM requires 329, 159, 154, 1,426, and 2,430 LE, while it performs 5.6, 3.2, 3.4, 1.0, and 0.8 Gbps for the Virtex2P, Virtex2 1000, Virtex4 xc4vlx80, VirtexE 2000, and Virtex 800 devices, respectively.
3. Since FSAM cannot be fit into Virtex 100, we performed the Virtex 800 device instead. Since the Virtex 800 and VirtexE series do not support block RAM, the bitmap table is placed in the external memories with the dedicated bus, which should be acceptable in the evaluation.
4. Since we lack the matching hardware to provide sufficient information, N/A represents that information is not available.
5. CAM is the content address memory, which can match content against data in parallel.

Substantial evaluation exhibited that the proposed FSAM can achieve the 573% and 233% increases in speedup compared to bitmap AC for 21,302 URL and 10,000 virus patterns, respectively. Moreover, our FSAM has the same worst-case time as bitmap AC when performing the prehash, root-index, and bitmap AC matching in parallel. For the space requirements, our FSAM increases by only 4 bytes of the bit vector for each state and the root-index tables for the root state. Therefore, the extra space requirements of 10.73MB and 5.22MB for 21,302 URL and 10,000 virus patterns, respectively, are quite acceptable with the currently available technologies.

In the implementation with a Xilinx Virtex2P device, the result demonstrates that our FSAM surpasses all other existing hardware in terms of the pattern size and throughput. Our FSAM can support the largest pattern size of 32,634 bytes and run at the high throughput of 11.1Gpbs. Furthermore, since our architecture works for both external and internal memories, and the external ASIC memories often run at a much higher clock rate than FPGA memories, our architecture is scalable to a large amount of patterns. If the high-speed external memories are employed, FSAM can support up to 21,302 patterns while maintaining similar high performance.

There are two possible future directions for this work. First, for broadening the applications using FSAM, our prehash and root-index techniques can be applied to the other automaton matching algorithms such as the regular expression automaton and the suffix automaton. Second, our FSAM for the content-filtering service can be integrated into a network gateway for field trial evaluation.

ACKNOWLEDGMENTS

Many thanks to anonymous reviewers who gave their time and helpful advices.

REFERENCES

AHO, A. V. AND CORASICK, M. J. 1975. Efficient string matching: an aid to bibliographic search. *Comm. ACM*, 333–340.

ALDWAIRI, M., CONTE, T. AND FRANZON, P. 2005. Configurable string matching hardware for speeding up intrusion detection. *ACM SIGARCH Comput. Archit. News*.

ANTONATOS S., POLYCHRONAKIS M., AKRITIDIS P., ANAGNOSTAKIS K. D., AND MARKATOS E. P. 2005. Piranha: fast and memory-efficient pattern matching for intrusion detection. In *Proceedings of the 20th IFIP International Information Security Conference*. Springer, Berlin, Germany.

ANTONATOS, S., ANAGNOSTAKIS K., AND MARKATOS, E. 2004. Generating realistic workloads for network intrusion detection systems. In *Proceeding of the ACM Workshop on Software and Performance*. ACM, New York.

ATTIG, M., DHARMAPURIKAR, S. AND LOCKWOOD, J. 2004. Implementation results of bloom filters for string matching. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* IEEE, Los Alamitos, CA.

BAKER, Z. K. AND PRASANNA, V. K. 2004. Time and area efficient pattern matching on FPGAs. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. ACM, New York.

BLÜTHGEN, H. M., NOLL, T. AND AACHEN, R. 2000. A Programmable processor for approximate string matching with high throughput rate. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors.* IEEE, Los Alamitos, CA.

BOSE, P., GUO, H., KRANAKIS, E., MAHESHWARI, A., MORIN, P., MORRISON, J., SMID, M., AND TANG, Y. 2005. On the false-positive rate of bloom filters. http://cg.scs.carleton.ca/~morin/publications/ds/bloom-submitted.pdf.

BOYER, R. S., AND MOORE, J. S. 1977. A fast string searching algorithm. *Comm. ACM 20*, 10, 762–772.

BU, L. AND CHANDY, J. A. 2001. A keyword match processor architecture using content addressable memory. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*. ACM, New York.

CHO, Y. H. AND MANGIONE-SMITH, W. H. 2005. A pattern matching coprocessor for network security. In *Proceedings of the 42nd Annual Conference on Design Automation.* ACM, New York.

CLAM ANTIVIRUS. 2006. Clam Anti-virus. http://www.clamav.net/.

CLARK, C. R. AND SCHIMMEL, D. E. 2003. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. Lecture Notes in Computer Science, vol. 2778.

CLARK, C. R. AND SCHIMMEL, D. E. 2004. A pattern-matching co-processor for network intrusion detection systems. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '03).* IEEE, Los Alamitos, CA.

CLARK, C. R. AND SCHIMMEL, D. E. 2004. Scalable pattern matching for high speed networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*. IEEE, Los Alamitos, CA.

COIT, C., STANIFORD, S., AND MCALERNEY, J. 2002. Towards faster string matching for intrusion detection. In *Proceedings of the DARPA Information Survivability Conference and Exhibition.* ACM, New York, 367–373.

DANS GUARDIAN. 2006. DansGuardian content filter. http://dansguardian.org.

DESAI, N. 2002. Increasing performance in high speed NIDS. http://www.snort.org/docs/Increasing_Performance_in_High_Speed_NIDS.pdf.

DHARMAPURIKAR, S. AND KRISHNAMURTHY, P., SPROULL, T. S., AND LOCKWOOD, J. W. 2004. Deep packet inspection using parallel bloom filters. *IEEE Micro 24*, 1.

ERDOGAN, O. AND CAO, P. 2006. Hash-AV: fast virus signature scanning by cache-resident filters. http://crypto.stanford.edu/~cao/hash-av.html.

FRANKLIN, R., CARVER, D. AND HUTCHINGS, B. L. 2002. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA.

GOKHALE, M., DUBOIS, D., DUBOIS, A., BOORMAN, M., POOLE, S., AND HOGSETT, V. 2002. Granidt: towards gigabit rate network intrusion detection technology. Lecture Notes in Computer Science, vol. 2438.

LOCKWOOD, J. 2001. An open platform for development of network processing modules in reconfigurable hardware. In *Proceedings of the International Engineering Consortium Design Conference*.

MIKE, F. AND GEORGE, V. 2001. Fast Content-Based. Packet Handling for Intrusion Detection. Tech. rep. CS2001-0670, University of California, San Diego.

MITZENMACHER, M. 2005. Compressed bloom filters. *IEEE/ACM Trans. Netw.*

MOSCOLA, J., LOCKWOOD, J., LOUI, R. P., AND PACHOS, M. 2003. Implementation of a content-scanning module for an internet firewall. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA.

NAVARRO, G. 2001. A guided tour to approximate string matching. *ACM Comput. Surv. 33*, 31–88.

NAVARRO, G. AND RANOT, M. 2002. *Flexible Pattern Matching in Strings*. Cambridge University Press, Cambridge, MA.

PAPADOPOULOS, G. AND PNEVMATIKATOS, D. 2005. Hashing + memory = low cost, exact pattern matching. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. Springer, Berlin, Germany.

PARK, J. H. AND GEORGE, K. M. Parallel string matching algorithms based on dataflow. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*. IEEE, Los Alamitos, CA.

RAFFINOT, M. 1997. On the multi backward dawg matching algorithm (MultiBDM). In *Proceedings of the 4th South American Workshop on String Processing*.

SASTRY, R., RANGANATHAN, N. AND REMEDIOS, K. 1995. CASM: a VLSI chip for approximate string matching. *IEEE Trans. Pattern Anal. Mach. Intell. 17*.

SIDHU, R. AND PRASANNA, V. 2001. Fast regular expression matching using FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, Los Alamitos, CA.

SNORT. 2006. Snort: The Open Source Network Intrusion Detection System. http://www.snort.org.

SOURDIS, I. AND PNEVMATIKATOS, D. 2003. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. Lecture Notes in Computer Science, vol. 2778.

SOURDIS, I. AND PNEVMATIKATOS, D. 2004. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*. IEEE, Los Alamitos, CA.

SOURDIS, I., PNEVMATIKATOS, D., WONG, S. AND VASSILIADIS, S. 2005. A reconfigurable perfect-hashing scheme for packet inspection. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. Springer, Berlin, Germany.

SPAMASSASSIN. 2006. The Apache SpamAssassin Project. http://spamassassin.apache.org/

SQUIDGUARD. 2006. SquidGuard filter. http://www.squidguard.org/.

TAN, L. AND SHERWOOD, T. 2005. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. ACM, New York.

TRIPP, G. 2005. A finite-state-machine based string matching system for intrusion detection on high-speed network. In *Proceedings of the EICAR Conference.* IEEE, Los Alamitos, CA, 26–40.

TUCK, N., SHERWOOD, T., CALDER, B. AND VARGHESE, G. 2004. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the IEEE INFOCOM Conference.* IEEE, Los Alamitos, CA.

WU, S. AND MANBER, U. 1992. Fast text searching allowing errors. *Comm. ACM 35*, 83–91.

Received May 2006; revised March 2007, June 2007; accepted August 2007