Deterministic High-Speed Root-Hashing Automaton Matching Coprocessor for Embedded Network Processor

Kuo-Kun Tseng, Ying-Dar Lin and Tsern-Huei Lee
National Chiao Tung University, Taiwan
{kktseng@cis, ydlin@cis and thlee@atm.cm}.nctu.edu.tw

Yuan-Cheng Lai
National Taiwan University of Science and Technology, Taiwan
laiyc@cs.ntust.edu.tw

ABSTRACT
While string matching plays an important role in deep packet inspection applications, its software algorithms are insufficient to meet the demands of high-speed performance. Accordingly, we were motivated to propose fast and deterministic performance root-hashing automaton matching (RHAM) coprocessor for embedded network processor. Although automaton algorithms are robust with deterministic matching time, there is still plenty of room for improvement of their average-case performance. The proposed RHAM employs novel root-hashing technique to accelerate automaton matching. In our experiment, RHAM is implemented in a prevalent automaton algorithm, Aho-Corasick (AC) which is often used in many packet inspection applications. Compared to the original AC, RHAM only requires extra vector size in 48 Kbytes for root-hashing, and has about 900% and 420% outperformance for 20,000 URLs and 10,000 virus patterns respectively. Implementaion of RHAM FPGA can perform at the rate of 12.6 Gbps with the pattern amount in 34,215 bytes. This is superior to all previous matching hardware in terms of throughput and pattern set.

Categories and Subject Descriptors
C.2.0 [**Computer-Communication Networks**]: General—Security and protection; C.2.0 [**Computer-Communication Networks**]: General—Data communications; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—Packet-switching networks; C.2.3 [**Computer-Communication Networks**]: Network Operations—Network management; I.5.4 [**Computing Methodologies]**: Pattern Recognition—Applications

General Terms
Algorithms, Performance, Design

Keywords
Coprocessor, String matching, Hashing, Finite automaton, Packet inspection.

1. INTRODUCTION
In recent years, deeper and more complicated content matching has been required for applications dealing with intrusion detection, keyword blocking, anti-virus and anti-spam. In such applications, string matching usually occupies 30% to 70% of the systems' workload [1, 2]. New content applications are increasingly built on home and office network gateways which often are implemented with a moderately performing embedded network processor. Therefore, as transmission speed increases, it is very important to design an appropriate matching coprocessor to offload the matching work from the network processors.

TABLE 1. Comparison of the On-Line String Matching Algorithms

Algorithm	Dynamic Programming	Backward Filtering	Automaton	Bit Parallel
Description	Matrix operations to compute the similarity between text and pattern	Discarding window of text that is not a substring of pattern in backward scanning	Search through a Deterministic Finite Automaton (DFA)	Simulate Non-Deterministic Finite Automaton (NFA) by bitwise operations
Average Time Complexity	O(n)	Sub-linear	O(n)	O(n)
Worst Time Complexity	O(n)	O(nm)	O(n)	O(n)
Text Length	Fixed short length	Variable long length	Variable long length	Variable long length
Pattern Length	Fixed short length	Variable short length	Variable long length	Fixed short length
Multiple Pattern	No	Yes	Yes	Yes
Regular Expression	No	No	Yes	Yes
Advantage for hardware	Simple systolic array circuit	Storage is normally smaller than Automaton	Comparison is a lookup operations	Bitwise operation is fast
Disadvantage for hardware	Not feasible to have a large systolic array	Long latency to compute discarding window	Table size is larger than Bit-Parallel	Not feasible to have a long bit mask
Typical Algorithm	Edit Distance	Boyer-Moore	Aho-Corasick	Shift-OR

To understand the necessary requirements of string matching algorithms, we surveyed real patterns from open source software including Snort [3] for intrusion detection, ClamAV [4] for anti-virus, SpamAssassin [5] for anti-spam, and SquidGuard [6] and DansGuardian [7] for Web blocking. The requirements can be concluded to be those matching the variable-length, multiple patterns and on-line processing of all packet inspection systems. Complex patterns, such as those created by varying class, wildcard, regular expression and case-sensitivity may increase the expressive power of the patterns and has been used in some applications. These complex patterns can be converted to patterns composed of multiple simple patterns [8]; they are

optional for matching algorithms.

In this survey, as in Table 1, on-line matching algorithms can be classified into four categories: dynamic programming, bit parallel, filtering, and automaton algorithms. The dynamic programming [9] and bit parallel [10] algorithms are inappropriate for variable-length and multiple simple patterns, and the filtering algorithms [11] have a poor worst-case time complexity O (nm), where n and m are the length of text and patterns respectively. Only automaton based algorithms such as Aho-Corasick (AC) [12] that support variable-length, multiple simple patterns, and deterministic worst-case time complexity O(n) are selected as a base to develop our new approaches.

Although bitmap AC has good worst-case matching time complexity in O(n), this is insufficient for high speed matching. In this paper, we present a root-hashing automaton matching (RHAM) that is built on an embedded system and applied to a network gateway to perform deep content filtering as shown in Fig. 1. This hashing acceleration is the fast matching approach to improve the average-case time of an automaton. The idea is to hash multiple bytes substring of text and to compare the result with the vector of the suffixes of the root state in the bitmap AC automaton. If a root state is visited, slow automaton matching is not required.

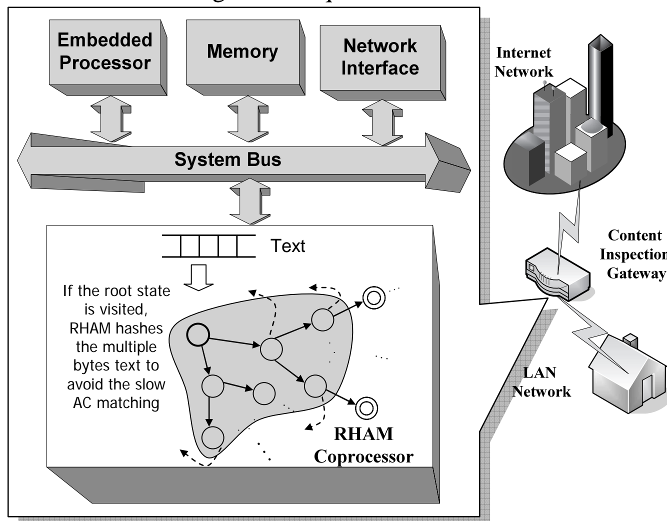

Fig. 1. Packet inspection gateway with RHAM coprocessor.

Since the root state usually has many next states and is often visited during the matching phase, root-hashing is an effective accelerator with low memory usage. For evaluating our approaches, the space and time complexities are formally analyzed with real patterns. Furthermore, our design implemented in Xilinx FPGA can achieve 12.6 Gbps throughput with a pattern set of 34,215 bytes, which significantly outperforms previous matching hardware.

The rest of this paper is organized as follows: Section 3 presents the related AC, hashing matching and string matching hardware. Section 3 describes the architecture and algorithm of RHAM. The formal analysis, evaluation of real patterns and network traffic are demonstrated in Section 4, and result of FPGA implementation are shown in Section 5. Finally, conclusions are drawn in Section 5.

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 subsection 2.1, and that for the last one is given in subsection 2.2. Finally, the related string matching hardware is introduced in subsection 2.3.

2.1 AC Related Algorithms

Before performing AC matching, there is a need to construct a state machine from the patterns. Adapting from the example in [6], Fig. 2 (a), (b) and (c) are AC's three major functions for patterns "TEST", "THE", "HE".

The first *Goto* Function shown in Fig. 2 (a) starts with an empty root node and adds states to the state machine for each pattern. That *Goto* function is a tree structure that shares common prefixes with all of the patterns. During the matching the *Goto* function is traversed from one state to the other with the text byte by byte.

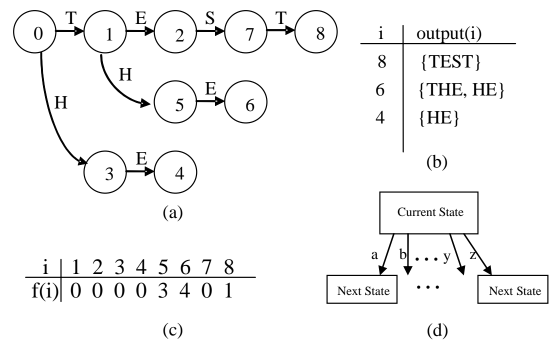

Fig. 2. (a) *Goto* function. (b) *Output* function. (c) *Failure* function. (d) AC table implementation.

The second is the *Output* function shown in Fig. 2 (b) needs a table to store the matched pattern with their corresponding state in the *Goto* tree. *Output* function records a matched state for a matched pattern if that current state is matched during the visiting. The third is the *Failure* function as shown in Fig. 2 (c). During the construction, failure states are added from the state, where their longest prefix also leads to a valid state in the *Goto* tree. During the matching Failure function is used when a match fails after a partial match. After the construction of a machine, the AC state machine is traversed from the current node to the next node according to the input byte.

Aho-Corasick is a typical deterministic finite automaton

(DFA) based algorithm used for string matching. However, there are several variations. Bitmap AC [13] uses bitmap compression to reduce the storage of AC states. AC_BM [1, 14, 15] is a combination of the AC and Boyer-Moore (BM) algorithms, and aims to improve the conventional AC from O(n) to sub-linear time complexity. AC_BDM [16] combines AC with backward DAWG matching (BDM) to improve the average-case time complexity of the conventional AC. Bit-split AC [17] splits the width of the input text into a smaller bit width to reduce the memory usage in selecting the next states. Since AC_BM has the worst-case time complexity O(nm), AC_BDM requires overhead of switching between AC and BDM, and bit-split AC needs large match vector for each bit-split state, they are impractical for packet inspection hardware. Thus, a scalable bitmap AC with space efficiency is more suitable for our purpose.

Bitmap AC is a compromise between table and link list approaches. It resolves the wasted memory of the AC table that uses 256 next pointers for each state. Bitmap AC maintains a 256-bit bitmap for each state to indicate whether a valid next state with a given character is valid or invalid, and it requires traversing along the failure pointer path. Fig. 3 shows the data structure of bitmap AC and how it locates the next state.

Bitmap AC solves this problem for AC. However, in order to locate the next state in bitmap AC, it must count all 1s in the 256-bit bitmap. This is known as a time-consuming operation that is dominated by loading the state and performing the population count.

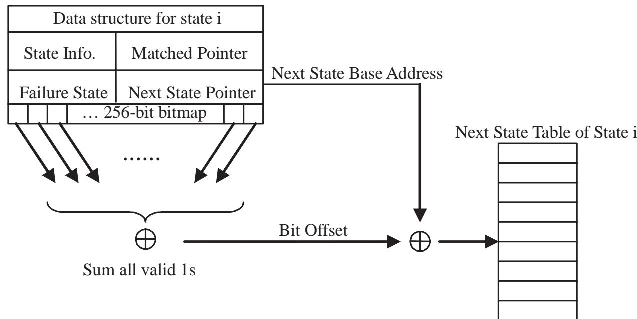

Fig. 3. Data structure of bitmap AC for state i, using bitmap to locate the next state.

2.2 Hashing Matching Related Algorithms

Related works have mentioned the hashing technique in string matching, which is utilized in BFSM [18] and PHmem [19]. The basic idea of the hashing matching technique is to use hashing functions to reduce the possible number of matched patterns for the naive matching algorithm. The problems common to them are that they require non-deterministic verification time and that they do not support long and large patterns.

Since BFSM was the first and famous approach to use hashing function in the string matching, 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 string matching. 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 Fig. 4 (a), the hashing functions $H_{1,1}$, $H_{1,2} \cdots H_{1,k}$ are used for length one of all patterns. Fig. 4 (b) 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.

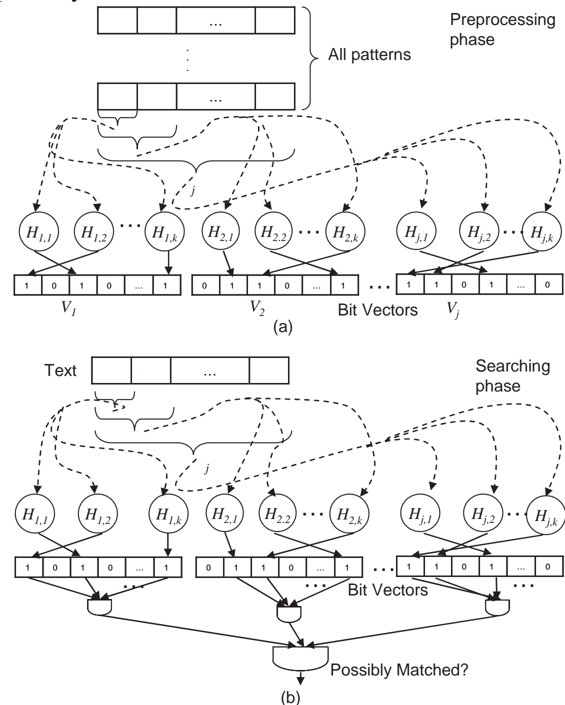

Fig. 4. (a) Bloom filter for string matching, (b) Each pattern is hashed into bit vectors in the preprocessing phrase, (c) Text is hashed and compared with the bit vectors in the searching phrase.

The basic logic behind philosophy of BFSM is that it uses multiple hashing functions to reduce the probability of false positive (hit), i.e., false in the positive match, but the AND function reports a positive value. When BFSM chooses k independent hashing functions to hash N patterns into a vector with size M , the probability of false positive P_{fp} is obtained as

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

BFSM modeling equation (1) is consistent only when a given ratio of $\frac{M}{N}$ is very high. That indicates that P_{fp} is only correct for low collision in the bit vector. For these reasons, we do not consider using Bloom filter in our pre-hashing approach.

In addition, BFSM requires multiple hashing functions that require multiple bank memory accesses for each bit vector table. Therefore, BFSM makes implementing the bit vectors impractical by using internal either the register or SRAM. For these reasons, we do not consider using Bloom filter (multiple hashing functions) in our RHAM design.

2.3 String Matching Hardware

In addition, among the existing string matching hardware, the most prevalent hardware is finite automaton (FA) based hardware. This is the case because of they support deterministic matching time and large patterns. FA based hardware can be divided into deterministic FA (DFA) and non-deterministic FA (NFA) based hardware. For DFA based hardware, there are three common designs in recently developed string matching hardware including Aho-Corasick (AC) based hardware [17, 20], Regular Expression (RE) based hardware [21, 22] and Knuth-Morris-Pratt (KMP) [23, 24, 25] based hardware. To save a greater number of states, KMP and AC are simplified from RE DFA by disabling their regular expression patterns. Each AC DFA supports multiple simple patterns, and each KMP DFA only supports a single simple pattern.

As for the NFA based hardware, there are two variations: comparator NFA [26, 27] which uses the distributed comparators, and decoder NFA [28] which uses the character decoder (shared decoder) to build its NFA circuits. The other existing non-DFA based hardware are parallel comparator [29, 30, 31], hashing matching [18, 19], systolic array [32] and parallel-and-pipeline [33].

3. RHAM DESIGN

In subsection 3.1, we introduce the algorithm of RHAM to obtain its overall operational concept. In subsection 3.2, the parallel architecture of RHAM is proposed for the study of its feasibility.

3.1 Algorithm

The proposed RHAM incorporates root-hashing matching to avoid slow AC matching. Because root state is frequently visited in the AC matching and usually has the large number of next states, a root-hashing technique is applied to advance multiple bytes in one single matching.

```
Preprocessing(P) {
S ← Build_AC(P)
RV ← Build_Roothash(S_0, k)
}.
Searching(T, S) {
S_c ← S_0
For 1 ≤ i ≤ |T| {
    Matched_Check(S_c)
    If S_c = S_0 {
        w ← T[i..(i + k + l)]
        Skip_Length ← Roothash(w, RV)
        If Skip_Length ≠ 0 {
            S_c ← S_0
            i ← i + Skip_Length
        }
        Else {
            S_c ← Match_AC(S_0, T[i])
            i ← i + 1
        }
    Else {
        S_c ← Match_AC(S_c, T[i])
        i ← i + 1
    }
}
}
                (a)
```

```
RV ← Build_Roothash(S_0, k) {
For 1 ≤ i ≤ k {
    RV_i ← {0}
    For 1 ≤ j ≤ |α_i| {
        α_{i,j} ← Prefixes(i, j)
        RV_i[H_i(α_{i,j})] ← 1
    }
}
Return RV
}.
Skip_Length ← Roothash(w, RV) {
Skip_Length ← 0
For 1 ≤ i ≤ k − l {
    Hit_i ← RV_i[H_i(w[1..i])]
    If Hit_i ≠ 1 {
        Skp_Length ← i
    }
    Else
    {
        Return Skip_Length
    }
}
Return Skip_Length
}.
                (b)
```

Fig. 5. Sequential algorithms, (a) Preprocessing and Matching functions of RHAM algorithm, (b) $Build_Roothash()$ and $Roothash()$ Functions of root-hashing.

As the sequential description with C-Like pseudo-code in Fig. 5, the matching algorithm requires both a preprocessing and a searching phase. Fig. 5 (a), $Preprocessing()$ first translates all patterns P into the states S of the AC tree using the conventional AC function $Build_AC()$. After S is obtained, $Preprocessing()$ then builds the root vectors RV by the function $Build_Roothash(S_0, k)$,where S_0 is the root state of AC tree, In the searching phase, $Searching(T, S)$ is described at the bottom of Fig. 5 (a). Initially, the current state S_c is set to the root state S_0, then the text T is processed in each matching loop. In the loop, $Matched_Check(S_c)$ is first performed to check the matching result. Also, $Skip_Length$ is initially set to zero and the substring of text $T[i..(i+k+l)]$ is set to the matching window w, where i is the current matching position of the text. If S_c is equal to S_0, then $Roothash(w, RV)$ is performed to test whether w has the non-hits in the root-hashing vectors RV or not. After the root-hashing matching, $Roothash(w, RV)$ reports the skip length $Skip_Length$. Which text can be skipped more than one character is shown by $i \leftarrow i + Skip_Length$. If S_c is not S_0, $Searching()$ continues the AC matching using $Match_AC(S_c, T[i])$ to match a single character.

In the preprocessing, $Build_Roothash(S_0, k)$ is used to build multiple root vectors, which is described at the top of Fig. 5 (b). This function inputs the prefixes α_i of the patterns within the length k by using the $prefixes(S_0, k)$. In

the processing of root-hashing, all prefixes α_i are hashed into the root vectors RV. The ith root-hashing function H_i hashes the corresponding prefix $\alpha_{i,j}$ into the ith vectors. During searching, when the current state is the root state, the matching algorithm performs $Roothash(w,V_c)$ to avoid AC matching as in the bottom of Fig. 5 (b). The main idea of $Roothash(w,V_c)$ is to test non-hit status in multiple root vectors, then to set the maximum non-hit vector number to $Skip_Length$. During the non-hit testing operation, $Hit_i \leftarrow RV_i[H_i(w[1..i])]$ performs a bit level index in RV_i with $H_i(w[1..i])$ in order to return the hit status Hit_i for the length i. After testing each RV_i, if any Hit_i has a possible hit status ($Hit_i = 1$), the root-hashing matching will stop the operation and return the $Skip_Length$, in which the longest consecutive non-hit length is selected to be skipped.

3.2 Architecture

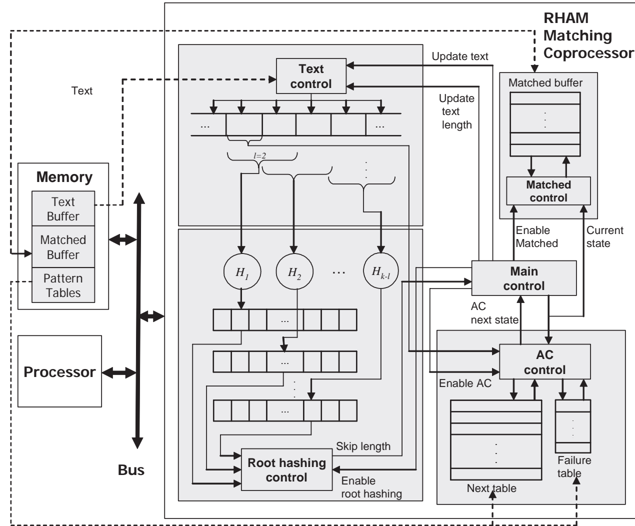

Fig. 6. The parallel architecture of RHAM coprocessor, including the logic circuits and blocks of root-hashing and AC matching.

A preferred parallel architecture is suggested in Fig. 6, where a RHAM coprocessor performs root-hashing and AC matching units in parallel. This architecture can concurrently process a one-byte text for AC matching, and a multiple of a $k-l$ bytes text for root-hashing in a single matching iteration, where l is the length of the substring of each hashing vector. In this example, $l = 2$, and k is the maximum number of hashing vectors.

For the storage, in addition to the original state and next state address tables, RHAM requires the bit vector tables. For the flexibility of the storage, these tables can be stored in either internal or external memories.

4. EVALUATION

This section intends to evaluate the space requirement and performance of our RHAM. In the first subsection, we formally derive the time and space requirement of RHAM. To demonstrate more realistic results, the evaluation of real patterns and network traffic are investigated in the last subsection.

4.1 A. Formal Analysis of Space Requirement and Performance

The space requirement can be determined by summing the original AC space $Size_{AC}$ and the root-hashing space $Size_{root}$. The original space $Size_{AC}$ is equal to the number of states $|S|$ multiplied by the state size. The proposed root-hashing only requires extra root vector space, which is a summation of all root vectors and defined as: $Size_{roothash} = \sum_{i=1}^{k} |RV_i|$, where $|RV_i|$ stands for the bit vector size of the length i vector. The probability of a non-hit is defined in [18]. $|RV_i|$ can be determined from the corresponding p_{nonhit} and the number of prefixes $|\alpha_i|$ as: $|RV_i| = \frac{1}{1-(p_{nonhit})^{\frac{1}{|\alpha_i|}}}$. The root-hashing and AC matching can be performed in parallel; the computation of the next states in the multiple units is independent. Thus the average time is $T_{avg_time} = \frac{P_{root} \times T_{root} + (1-P_{root}) \times T_{AC}}{k_{avg}}$; where T_{avg_time} is the average time to process a byte, T_{root} is the root-hashing time, P_{root} is the probability of non-hit in while using the root-hashing, T_{AC} is the AC matching time and k_{avg} is the average skip length of each text processing operation. Since AC matching is the critical path, the worst-case time of RHAM is equal to T_{AC}.

In the theory, the probability P_{root} can be determined from $P_{root} = P_{roothash} \times \sum_{i=1}^{k} P_i$, where P_{root} is computed as $P_{roothash}$ multiplied by a summation series of non-hit probabilities P_{nonhit_i} from the first to k^{th} vectors. $P_{roothash}$ is the probability of performing root-hashing, and P_i is the consecutive non-hit probabilities from the first to i^{th} vector. P_i can be obtained from

$$P_i = \begin{cases} \prod_{j=1}^{i} P_{nonhit} \times (1 - P_{i+1}),\ for\ i > k \\ \prod_{j=1}^{i} P_{nonhit},\ for\ i = k \end{cases}, \qquad (2)$$

where P_{nonhit_i} is the probabilities of i th vector. In this equation $\prod_{j=1}^{i} P_{nonhit}$ is used to compute the consecutive non-hit probability. For $i > k$, P_i must exclude the non-hit

probability of a longer skip length; thus $(1-P_{i+1})$ is multiplied. The computation of k_{avg} is similar to P_{root}, except each P_i is required to be multiplied by a depth i value and has no $P_{roothash}$. k_{avg} is defined as $k_{avg}=\sum_{i=1}^{k}(P_i \times i)$.

4.2 Real Patterns and Network Traffic Simulations

In these simulations, we choose the URL blacklists and virus signatures from [6] and [4] respectively. Because the URL blacklists and virus signatures contain many patterns and also long patterns, these patterns are sufficient to evaluate the performance of the proposed RHAM algorithm. The analyzed URL blacklists contain 20,000 patterns and generate 194,096 states. The virus signatures used contain 10,000 patterns and generate 402,173 states, respectively. In addition to the patterns, Google website data consisting of over 100 MB and Ethereal capture examples (http://www.ethereal.com) are selected as the text used to evaluate the URL patterns and ClamAV patterns respectively.

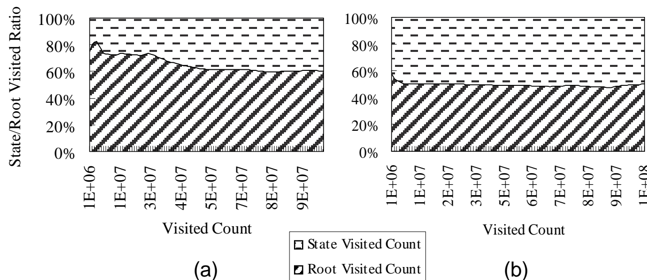

Fig. 7. Visited count for root state simulations (a) URL pattern and Google website text, (b) ClamAV patterns and Etheral capture text.

Demonstrating the high frequency of visiting in the root state can confirm the usefulness of the root-hashing technique, thus we first simulated the number of real traffic visited for the root state. Fig. 7 demonstrates that the Google and Ethereal data have about 60% and 50% root visited rates respectively. This result agreed that root-hashing has high usage in the automaton matching.

For evaluating the effect of different bit vector sizes, we used a 16 bytes text window and compared it with different bit vector sizes ranging from 4,096 to 65,536 bits. Fig. 8 (a) shows that a vector size of 32,768 bits may be a suitable choice to achieve high performance with only moderate memory consumption. The proposed RHAM can use either single or multiple vectors, multiple vectors can increase performance while using more memory resources. Fig. 8 (b) demonstrates the average skip length for the different window sizes with a 32,768 bit vector size; the result shows that text windows sized 12 to 8 may be better configured for low cost and high performance. As per the above results, RHAM is about 900% and 420% faster than bitmap AC for URL and virus patterns respectively. With the above proper configuration, the vector size is 32,768 bits and the text window size ranging from 12 to 8. The extra memory space for the root-hashing vector only requires 48 Kbyte, which is acceptable in light of the low cost implementation.

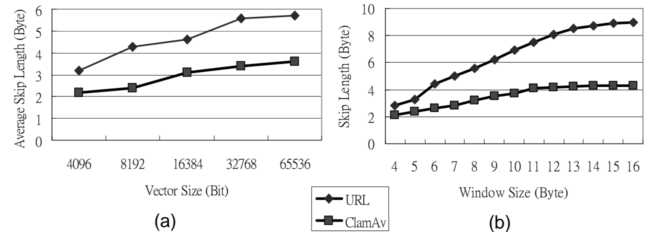

Fig. 8. (a) Skip lengths for 16 bytes windows with different vector sizes. (b) Skip lengths for multiple vectors of different window size.

5. IMPLEMENTATION

In this section, subsection 5.1 gives a description of hardware implementation for the RHAM hardware. Subsection 5.2 gives an exhausted comparison with previous hardware implementations.

5.1 Hardware Implementation

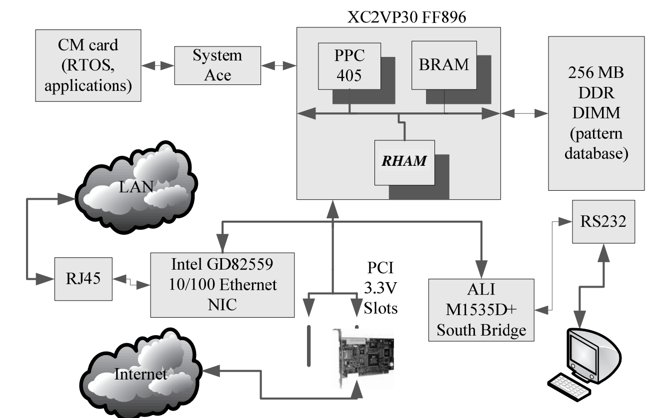

Fig. 9. Development platform for the RHAM implementation.

Xilinx ML310 is a FPGA based platform for RHAM implementation as shown in Fig. 9. This platform has 2448 Kbits internal block RAM, 30816 LUTs and two hardwired IBM PPC405 processors. For the peripheral, ML310 has one Ethernet port, one PCI slot for additional NIC extension, one 256 MB DDR RAM module and one CF card to store the image of the file system. During the operation, the packets are inputted from the on board Ethernet port, and processed by the PPC 405 CPU. Of course, if the RHAM is implemented, the deep packet inspection of the packets is offloaded to RHAM engine.

For the development tools, the Xilinx EDK and Synplicity SynplifyPro are used in the system implementation. The EDK can generate the bit streams from the hardware/software co-design files of SRAM implementation. For the software design, the files include the mapping address define files and the drivers of all peripherals needed for building the complete RTOS image. For the hardware design, the Verilog is used to design string matching hardware, then ModelSim and Debussy are the simulator and debugger tools, respectively, to verify the RHAM design.

The proposed architecture is a parallel design where all modules are working at the same time. The block diagram of the hardware implementation has the following major modules.

- *FSM Unit* controls the working flow of the whole hardware system.
- *Root-Hashing Unit* is used for fast matching at the root state. It tests the bit vector for multiple input bytes by hashing function and sending the hashing result to FSM.
- *Bitmap AC Unit* counts all 1s for locating the next state.
- *SM Controller Unit* provides the control registers including the length of text buffer and enable the signal for the software to program.

5.2 Comparison

TABLE 2. The Comparisons of String Matching Hardware

Matching Hardware	Device	Pattern Size (Byte)	Speed (Gbps) [1]
RHAM[2]	Virtex2P	34,215	12.6
	Virtex2 1000		6.8
	Virtex2 6000		9.3
	Spartan3 400		7.1
	VirtexE 2000		2.4
	Virtex 800[2]		2.0
Bit-split AC [17]	Xilinx FPGA	2,048	10.0
Parallel Bloom Filter [18]	VirtexE 2000	9,800	0.6
Perfect Hashing [19]	Virtex2 1000	20,911	2.9
DFA+counter [21]	VirtextE 1000	11	3.8
Parallel Regular DFA [22]	VirtexE 2000E	420	1.2
KMP Comparators [23]	Virtex2P	32	2.4
Comparator NFA [26]	Virtex 100	29	0.5
Meta Comparator NFA [27]	VirtexE 2000	8,003	0.4
Approximate Decoder NFA [28]	Virtex2 6000	17,537	2.0
Offset Index Comparators [29]	Spartan3 400	20,800	1.9
Pre-decoded Comparators [30]	Virtex2 6000	18,032	9.7
CAM Comparators [31]	VirtexE 1000	640	2.2

1. Speed (Throughput) is of an average performance. Except RHAM and BFSM, other hardware have the same worst and average cases.
2. Since RHAM cannot fit into Virtex 100, the similarly performing the Virtex 800 device is used. The Virtex 800 and VirtexE series did not support block RAM. The bitmap table is placed in the external memories with the dedicated bus, which should be acceptable in the evaluation.

In Table 2, we compare 12 major types hardware analyzed in related works. Since matching hardware are pursuing higher throughput and larger pattern sizes, they are the major factors in this comparison. In addition, many experiments [19, 21, 23, 29, 32] had used on-chip circuits or internal memories. Thus, we implemented the proposed RHAM using internal memories to reach a fair evaluation. In the RHAM implementation with a Xilinx Virtex2P device, FPGA run at 315 MHz with a performance of 12.6 Gbps. For storage, RHAM implementation can handle patterns of 34,215 bytes, which is composed of 1,980 patterns; each with an average length of 14.4 bytes. Conclusively, our RHAM is superior to all the previous string matching hardware in terms of both space requirements and performance.

Nevertheless, even more than 34,215 bytes can be achieved with the external memory version; RHAM can be implemented with external multiple bank memory. Although external memory produces overhead for memory access, the ASIC hardware often runs at a much higher speed than FPGA devices.

6. Conclusion

The proposed RHAM is a novel design with high performance, a scalable pattern set and a deterministic worst-case time, which can quickly verify multiple bytes text to avoid slow AC matching. Since the root state is a highly visited state during the matching, root-hashing is an effective approach to accelerate the automaton. Substantial evaluation determined that the proposed RHAM only requires extra vector size when used in 48 Kbytes, and can achieve around 900% and 420% speedup from the original AC for 20,000 URLs and 10,000 virus patterns respectively. When implemented with a Xilinx Virtex2P device, the result demonstrates that our RHAM surpasses all other existing hardware in terms of pattern size and throughput. Our RHAM supports the largest pattern size of 34,215 bytes and runs at the highest throughput of 12.6 Gbps.

There are two possible future directions for this work. First, in the broadening RHAM's applications wherein our pre-hashing and root-indexing techniques can be applied to the other automaton matching algorithms such as the regular expression automaton and the suffix automaton. Second, our RHAM for the packet inspection service can be integrated into a network gateway for field trial

evaluation.

7. REFERENCES

[1] F. Mike and V. George, "Fast Content-Based. Packet Handling for Intrusion Detection," *UCSD. Technical Report CS2001-0670*, May 2001.

[2] S. Antonatos, K. Anagnostakis and E. Markatos, Generating Realistic Workloads for Network Intrusion Detection Systems. *ACM WOSP*, 2004.

[3] M. Roesch et al, "Snort: The Open Source Network Intrusion Detection System," *http://www.snort.org/*.

[4] T. Kojm et al, "Clam Anti-virus," *http://www.clamav.net/*.

[5] J. Mason et al, The Apache SpamAssassin Project. *http://spamassassin.apache.org/*.

[6] T. D. Internordia et al, "SquidGuard filter," *http://www.squidguard.org/*.

[7] D. Barron et al, "DansGuardian content filter," *http://dansguardian.org/*.

[8] G. Navarro and M. Ranot, "Flexible Pattern Matching in Strings," *Cambridge University Press*, 2002.

[9] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, 33(1):31-88. 2001.

[10] S. Wu and U. Manber, "Fast Text Searching Allowing Errors," *Communication of the ACM*, 35:83-91.

[11] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, 20, 10, 762–772.

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

[13] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *IEEE Infocom*, Hong Kong, China, 2004.

[14] C. Coit, S. Staniford and J. Mcalerney, "Towards Faster String Matching for Intrusion Detection," *DARPA Information Survivability Conference and Exhibition*, pp. 367-373, 2002.

[15] N. Desai, "Increasing performance in high speed NIDS," *http://www.snort.org/*.

[16] M. Raffinot, "On the Multi Backward Dawg Matching Algorithm (MultiBDM)," *Workshop on String Processing*, Carleton U. Press, 1997.

[17] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection And Prevention," *ISCA*, 2005.

[18] S. Dharmapurikar and P. Krishnamurthy, T. S. Sproull and J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro*, Vol. 24, No. 1, Jan. 2004.

[19] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," *International Conference on Field Programmable Logic and Applications*, Aug. 2005.

[20] M. Aldwairi, T. Conte and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," *ACM CAN*, 2005.

[21] J. Lockwood, "An Open Platform for Development of Network Processing Modules in Reconfigurable Hardware," *IEC DesignCon*, Santa Clara, CA, Jan. 2001.

[22] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," *IEEE FCCM*, 2003.

[23] Z. K. Baker and V. K. Prasanna, "Time And Area Efficient Pattern Matching on FPGAs," *ACM/SIGDA FPGA*, California, USA, Feb 2004.

[24] G. Tripp, "A Finite-State-Machine Based String Matching System for Intrusion Detection on High-Speed Network.," *EICAR*, May 2005.

[25] L. Bu and J. A. Chandy, "A Keyword Match Processor Architecture Using Content Addressable Memory," *ACM VLSI*, April 26-28, 2004.

[26] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching using FPGAs," *IEEE FCCM*, April 2001.

[27] R. Franklin, D. Carver and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *IEEE FCCM*, Napa, CA, Apr 2002.

[28] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," *IEEE FCCM*, 2004.

[29] Y. H. Cho and W. H. Mangione, "A Pattern Matching Coprocessor for Network Security," *ACM/IEEE DAC*, California, USA, Jun 2005.

[30] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," *IEEE FCCM*, 2004.

[31] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," *LNCS*, Volume 2438, Jan 2002.

[32] H. M. Blüthgen, T. Noll and R. Aachen, "A Programmable Processor For Approximate String Matching With High Throughput Rate," *IEEE ASAP*, 2000.

[33] J. H. Park and K. M. George, "Parallel String Matching Algorithms based on Dataflow," *HICSS*, Hawaii, 1999.