Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 937–950
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

A platform-based SoC design and implementation of scalable automaton matching for deep packet inspection

Ying-Dar Lin [a], Kuo-Kun Tseng [a,*], Tsern-Huei Lee [b], Yi-Neng Lin [a], Chen-Chou Hung [a], Yuan-Cheng Lai [c]

[a] *Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan*
[b] *Department of Communication Engineering, National Chiao Tung University, Hsinchu, Taiwan*
[c] *Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan*

Received 1 November 2006; received in revised form 14 March 2007; accepted 14 March 2007
Available online 11 April 2007

Abstract

String matching plays a central role in packet inspection applications such as intrusion detection, anti-virus, anti-spam and Web filtering. Since they are computation and memory intensive, software matching algorithms are insufficient to meet the high-speed performance. Thus, offloading packet inspection to a dedicated hardware seems inevitable. This paper presents a scalable automaton matching (SAM) coprocessor that uses Aho-Corasick (AC) algorithm with two parallel acceleration techniques, root-indexing and pre-hashing. The root-indexing can match multiple bytes in one single matching, and the pre-hashing can be used to avoid bitmap AC matching which is a cycle-consuming operation. In the platform-based SoC implementation of the Xilinx ML310 FPGA, the proposed hardware architecture can achieve almost 10.7 Gbps and support over 10,000 patterns for virus, which is the largest pattern set from among the existing works. On the average, the performance of SAM is 7.65 times faster than the original bitmap AC. Furthermore, SAM is feasible for either internal or external memory architecture. The internal memory architecture provides high performance, while the external memory architecture provides high scalability in term of the number of patterns.

Keywords: Deep packet inspection; Automaton; String matching; Content filtering

1. Introduction

Since detecting malicious traffic on the Internet, such as viruses and intrusions, relies on looking for signatures in the packet payload, traditional firewalls that inspect only the packet header are insufficient for detection. Thus, deeper packet inspection such as intrusion detection, anti-virus, anti-spam and Web filtering are required to detect such application-level attacks that can be found in the field. The essential part of such solutions is the string matching which has been shown to be a time-consuming component that should be accelerated [1,2].

* Corresponding author.
E-mail address: kktseng@cis.nctu.edu.tw (K.-K. Tseng).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2007.03.005

For string matching, several algorithms and hardware architectures have been proposed to improve performance. Although the throughput of some approaches can achieve up to 10 Gbps, their common drawback is poor scalability. Their rules and pattern sets are hardwired into the FPGA and thus, the scalability is limited by the number of logic cells and the size of the embedded memory in the FPGA.

In this paper, we propose a scalable automaton matching (SAM) which is based on the Aho-Corasick (AC) algorithm with external memory architecture. AC is a common algorithm with the following features. First of all, AC has a deterministic performance in the worst case. Secondly, AC is robust for large and long patterns. Thirdly, AC can perform a multiple patterns match in a single matching iteration. However, the large memory usage is a critical problem for the AC algorithm. Another AC-based algorithm, the bitmap AC, improves memory utilization by using a 256-bit bitmap to replace the 256 word-size pointers of each state in AC. Hence, bitmap AC is the alternative that we adopted in this work.

SAM is developed with two acceleration techniques to archive a sub-linear matching time. The first technique is root-indexing that comes from the observing the AC's high frequency root-visiting behavior. The second technique is pre-hashing that comes from the observing the time-consuming bitmap operation during the bitmap AC matching. To reduce this kind of operation, pre-hashing can test quickly to avoid the bitmap AC matching. For scalability, our architecture uses external memories to store the whole pattern database of SNORT or even ClamAV.

Furthermore, SAM can easily update patterns without interrupting the operation or shutting down the machine since it is a memory based architecture. In our evaluation, we implemented SAM on a System On Chip (SoC) based platform with Xilinx FPGA Virtex2P and EDK design tool.

The rest of this paper is organized as follows. In Section 2, we first introduce the related algorithms and string matching hardware. Then, the proposed architecture and acceleration techniques are presented and an example is given in Section 3. Section 4 presents the software and hardware implementation of the SAM approach. The performance analysis, evaluation, and comparison with existing works are given in Section 5. Finally, we draw up our conclusion in Section 6.

2. Background

2.1. Selecting matching algorithms for packet inspection

To understand the appropriate requirements of string matching algorithms, we surveyed the real patterns from open source software including SNORT (http://www.snort.org) for intrusion detection, ClamAV (http://www.clamav.net) for anti-virus, SpamAssassin (http://spamassassin.apache.org) for anti-spam, and SquidGuard (http://www.squidguard.org) and DansGuardian (http://dansguardian.org) for Web blocking. The necessary requirements are variable-length, multiple patterns and on-line processing for all packet inspection systems.

Although the complex patterns such as class, wildcard, regular expression and case sensitive patterns might increase the expression power of the patterns and has been used in some applications, they can be converted into multiple simple patterns [3], and they are optional for matching algorithms.

Current existing on-line string matching algorithms for packet inspection can be classified into four categories: dynamic programming, bit parallel, filtering, and automaton algorithms. As summarized in Table 1, dynamic programming [3] and bit parallel [4] algorithms are inappropriate for variable-length and multiple patterns, and the filtering algorithms [5] have poor worst-case time complexity $O(nm)$, where n and m are the length of the text and patterns, respectively. Only the automaton based algorithms such as Aho-Corasick (AC) [6] support variable-length and multiple patterns, and also have the deterministic worst-case time complexity $O(n)$. Hence, the automaton based algorithm is a better choice for the packet inspection system, and was selected as the base to develop new approaches.

2.2. 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. 1a–c are AC's three major functions for patterns "TEST", "THE", "HE".

The first *Goto* function shown in Fig. 1a 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*

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

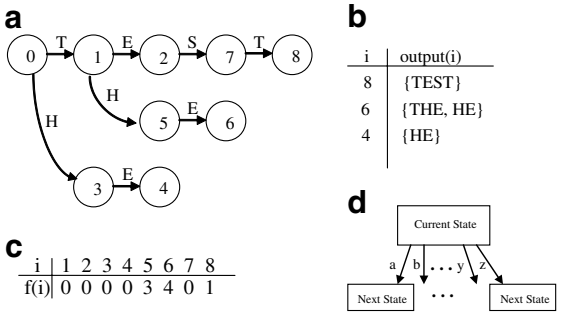


Fig. 1. (a) *Goto* function. (b) *Output* function. (c) *Failure* function. (d) AC table implementation.

function is traversed from one state to the other with the text byte by byte.

The second is the *Output* function shown in Fig. 1b 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. 1c. 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.

For the data structure of the *Goto* Function, there are two alternatives to store the next state links:

- The first alternative is the construction of a 2D-array table. Each state has 256 next state pointers for all of the ASCII input cases, as shown in Fig. 1d. It is the most popular implementation for fast matching, but it wastes memory space when the table is sparse.
- In the second alternative, data structure uses the link list, and each state only has the link list of existing next states. This kind of data structure has a smaller space requirement, but it is slow when there are many next states.

AC is a typical deterministic finite automaton (DFA) based on string matching. However, there are several variations of it. Bitmap AC [7] uses bitmap compression to reduce the storage of AC states. AC_BM [8] is a combination of the AC and Boyer Moore (BM) algorithms, and aims to improve the conventional AC from O(n) to the sub-linear time complexity. AC_BDM [9] combines AC with backward DAWG matching (BDM) to

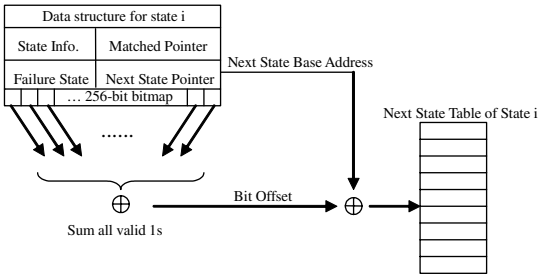


Fig. 2. Data structure of bitmap AC for state *i*, using bitmap to locate the next state.

improve the average-case time complexity of the conventional AC. Bit-split AC [10] splits the width of the input text into a smaller bit width to reduce the memory usage and the number of comparisons when selecting the next states. Since AC_BM has the worst-case time complexity O(*nm*) and 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 large patterns. Thus, a scalable bitmap AC with space efficiency is more preferable for our embodiment.

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. 2 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.

2.3. Hardware-based string matching

Since sting matching is a bottleneck for packet inspection systems [1], hardware solutions are required for high-speed content processing. Among the existing string matching hardware, the most prevalent one is the finite automaton (FA) based hardware because it has support for deterministic matching times and large patterns. FA based hardware can be divided into deterministic FA (DFA) and non-deterministic FA (NFA) based hardware. DFA based hardware has a unique transition that activates one state at a time and normally has a larger number of states compared to NFA. NFA can handle multiple transitions at one time, but it requires parallel circuits for comparing its variable multiple next states. Thus, majority of DFA based hardware uses the table or link list to store their patterns, and most NFA based hardware uses parallel reconfigurable circuits to handle their patterns.

For DFA based hardware, there are three common designs in recent string matching hardware: Aho-Corasick (AC) based hardware [11,12] Regular Expression (RE) based hardware [13,14] and Knuth–Morris–Pratt (KMP) [15–17] based hardware. To save more 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 support single simple patterns only. Thus, many KMP DFAs use duplicated hardware to support multiple patterns.

As for NFA based hardware, there are two variations: comparator NFA [18,19], which uses the distributed comparators, and decoder NFA [20], which the uses the character decoder (shared decoder) to build their NFA circuits.

Other existing non-DFA based hardware are the parallel comparator [21–23], Bloom filter [24], systolic array [25] and parallel-and-pipeline [26] hardware in our classification. Parallel comparator based hardware improves the performance of brute force algorithm by exploiting architecture parallelism and pipelining. Bloom filter based hardware uses multiple hashing keys for quick approximate matching. Using systolic array implementing dynamic programming for string matching is only proper for short patterns and text because the circuit size is proportional to the length of the patterns and text. Parallel and pipeline hardware uses naïve string matching and only accelerates processing time by increasing the hardware circuits. Like the systolic array, this approach also has the drawback of only being suited for short-length patterns.

3. SAM design

Although bitmap AC has the good worst-case matching time complexity in O(*n*), this is insufficient for high speed processing. In this paper, we present a scalable automaton matching (SAM) that is built on an embedded based system and applied to a network gateway to perform deep content filtering as

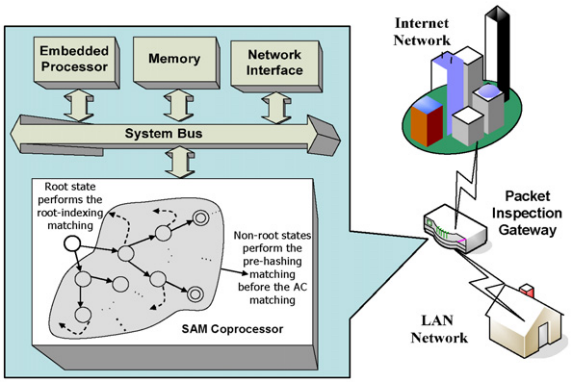


Fig. 3. Packet inspection gateway with SAM coprocessor, which performs two techniques: root-indexing matching for the root state and the pre-hashing matching for the non-root states before the AC matching.

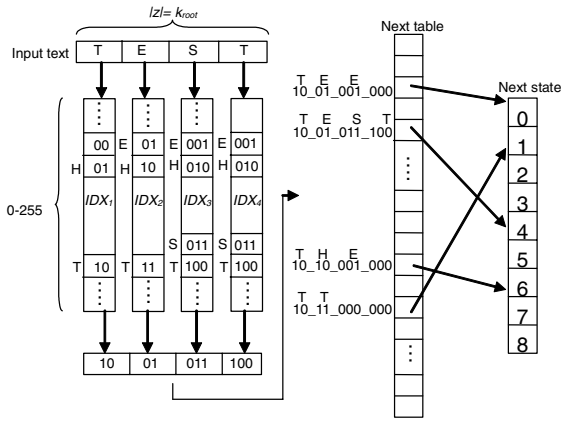


Fig. 4. Root-indexing architecture and example for the input text "TEST" with the patterns "TEST", "THE" and "HE".

shown in Fig. 3 SAM employs two techniques: novel pre-hashing for the non-root state and root-indexing for the root state to accelerate automaton based algorithms. The pre-hashing approach is a quick scanning for the non-root state to avoid the time-consuming automaton matching. First of all, the idea is hashing the substring of text and comparing the result with the vector for the suffixes of the state in the bitmap AC automaton. If a non-hit occurs, the slow automaton matching is not required. For the root state, the root-indexing approach uses a compressed technique to remember all the next states whose lengths are counting from the root state, less than l ($l > 1$). Thus, multiple bytes of length l, rather than one byte, can be handled in one single matching for the root state to accelerate the matching speed. Actually, since the root state is often visited in the matching operation, the root-indexing approach is an effective acceleration approach.

3.1. Root-indexing matching

In the AC tree, most of the failure links point to the root state, i.e., it will always go back to the root state when there is no next state for a given character. Thus, it is efficient to apply root-indexing in the root state where it can match multiple characters simultaneously. In Fig. 4, root-indexing comprises k_{root} index tables $IDX[1,\ldots,k_{\text{root}}]$ and a root next table $NEXT$, where k_{root} denotes the maximum length of root-indexing matching at the same time. Each entry of IDX stores a partial address for locating the next state in $NEXT$.

For example, if patterns are "TEST", "THE" and "HE", $IDX1$ to $IDX4$ will at least contain the appearing characters in the corresponding position as {"H","T"} for level one, {"E","H"} for level 2, {"E","S"} for level 3, {"T"} for level 4, respectively. However, since the latter tables are required to contain the entries of former tables, $IDX1$ to $IDX4$ will actually contain {"H","T"}, {"E","H","T"}, {"E","H","S","T"} and {"E","H","S","T"}, respectively.

In numbering the entries of IDX tables, the first IDX has 2 appearing characters and thus, "H" and "T" are numbered "01" and "10" in the binary format, respectively. The second IDX table using "01", "10" and "11" stands for {"E","H","T"}, respectively. The $NEXT$ table, indexed by a concatenation address of lookup value from the all the IDX tables, is used to store all the next states within length k_{root}. In the example of Fig. 4, 10_01_001_000, 10_01_011_100, 10_10_001_000 and 10_11_000_000 are concatenation addresses to locate the next states for "TEE", "TEST", "THE" and "TT", respectively.

3.2. Pre-hashing matching

The pre-hashing method can quickly examine the existence of next states to further avoid slow AC matching. Before the pre-hashing matching, it is necessary to build the pre-hashing bit vector in the preprocessing phase. First, we input the AC tree, which was built using conventional AC algorithm. For each state, we extract suffixes with the length 1.

When suffixes are obtained, the pre-hashing algorithm hashes suffixes into bit vectors. This procedure of building the bit vectors for state 1 is illustrated in Fig. 5a. In Fig. 5b and c, the mask of the rightmost four bits of the characters and the transformation from binary to one-hot representation are used as the hash function in our design. However, a better mask position is adjustable for a lower false positive according to the characteristics of the patterns.

In pre-hashing matching, the pre-hashing unit reads a byte substring and then hashes the substring. The hash result will be indicated by the pre-hashing unit. When the pre-hashing unit indicates a non-hit, the next state will be obtained from the root-indexing unit. However, if the hit condition is indicated by the pre-hashing unit, a slow bitmap AC matching will be performed.

3.3. System architecture

A preferable searching architecture is suggested in Fig. 6, where a string matching coprocessor performs three independent matching units in parallel. Hence, the control logic coordinates pre-hashing, root-indexing and bitmap AC matching for parallel processing. Also each matching function has its individual memory interface to access its pre-processing data. Since the design methodologies of SoC are popular and have matured in recent times, this specific component is quite feasible for use in modern IC technology.

In the SAM coprocessor, the three units can read the text in different lengths and perform their matching concurrently. This example processes a one-byte substring for AC matching, a two-byte

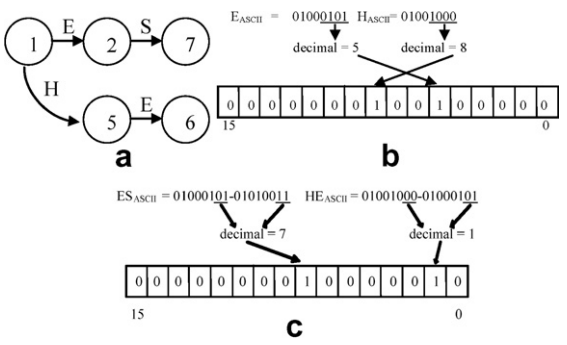


Fig. 5. (a) AC tree of state 1 for building the bit vector. (b) and (c) example of building the bit vector for length 1 and length 2 suffixes of state 1 in the preprocessing phase.

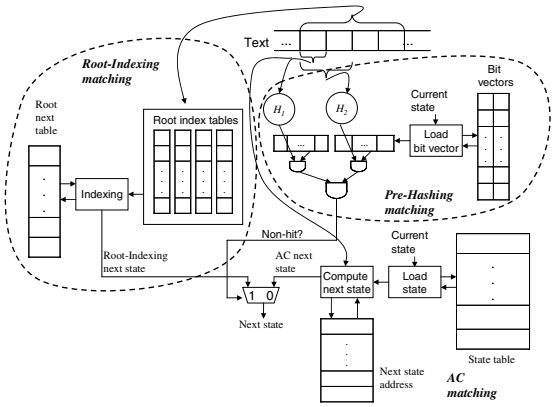


Fig. 6. Architecture of the string matching coprocessor.

substring for pre-hashing matching, and a four-byte substring for root-indexing matching in a single matching iteration. The root-indexing and bitmap AC are used to locate the next states, and the pre-hashing matching is used to decide on which is the next state to be used in the next matching iteration.

4. SAM implementation

4.1. Pre-processing and simulation software

The pre-processing procedure generates essential data structures for the proposed hardware, as shown in Fig. 7a. The *Make_Goto( ) and Make_Failure( )* functions are original functions defined by the AC algorithm, and our data structures were further on built according to the table constructed from these two basic functions. For bitmap AC, the *Make_Bitmap( )* function builds a 256-bit bitmap for each state and sets 1 to the corresponding bit position for each existing next state. It also builds the next state table for each state. The next function is *Build_Index( )* which builds the $IDX_{[1,\ldots,k]}$ tables and root next table *NEXT* for root-indexing pre-processing. In the final stage, *Build_BitVector( )* sets 1 to the bit vector by hashing the function according to all the next states of both the current state and the recursive failure node for pre-hashing preprocessing.

After the pre-processing procedure is finished, the simulation of SAM can perform matching according to the flow in Fig. 7b. For every matching iteration, the first current state is checked. When the current state is in the root state, the *Root-Index( )* matching is performed otherwise *Pre-Hash( )* is performed.

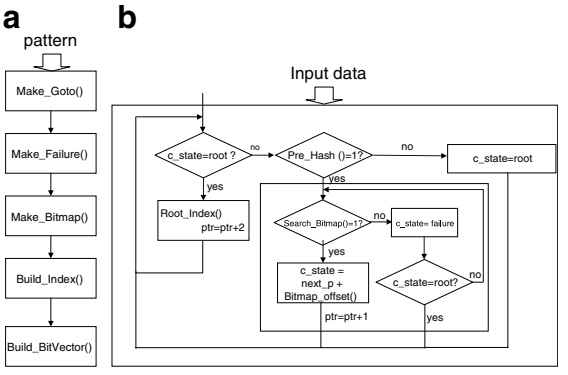


Fig. 7. (a) The pre-processing procedure. (b) The flow of C simulation model.

If *Pre-Hash( )* reports a non-hit situation, the current state will be set to the root state directly, and will do root-indexing matching. If a hit situation is reported, *Search_Bitmap( )* will check the existence of the next state for a given byte. If *Search_Bitmap( ) = 1*, the next state will be obtained from the base address pointer of the next state table plus the return value of *Bitmap_offset( )*. Note that if *Search_Bitmap( )* reports zero, the current state will be set to the failure state in the *while* loop until the current state becomes the root state. This C model can be the golden model for the proposed hardware design, and it can also be used to gather statistics for performance analysis.

4.2. Root-indexing and pre-hashing functions

Since bitmap AC has been done in previous work, its detailed function implementation is not described in this paper, and only the important root-indexing and pre-hashing functions are discussed. In pre-processing, procedures *Build_Index( )* and *Build_BitVector( )* are the two important functions.

The root-indexing technique comprises k_{root} root index tables $IDX_{[1,\ldots,k_{\text{root}}]}$ and a root next table *NEXT*, where k_{root} denotes the maximum length of root-indexing 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 characters appearing in the corresponding substrings in the suffixes of the root state.

In the pre-processing of the root-indexing, *Build_Index*(*S*) is first invoked to build $IDX_{[1,\ldots,k_{\text{root}}]}$ as Fig. 8. The length of input text and the number of

$IDX_{[1..k_{root}]} \leftarrow Build\_Index(S)\{$
  $\alpha \leftarrow \eta(S_0, k_{root})$
  For $1 \le j \le k_{root}$ {
    $\rho \leftarrow 1$
    For $1 \le x \le 255$ {
      $IDX_j[x] \leftarrow 0$
      If $j \ge 2$
        $IDX_j[x] \leftarrow IDX_{j-1}[x]$
    }
    For $1 \le x \le |\alpha_j|$ {
      $IDX_j[\alpha_j[x]] \leftarrow \rho$
      $\rho \leftarrow \rho + 1$
    }
  }
  Return $IDX_{[1..k_{root}]}$
}.

Fig. 8. Function for building root-indexing tables.

IDX tables are equal to k_{root}. This function builds the *IDX* table from IDX_1 to $IDX_{k_{\text{root}}}$. It first performs $IDX_j[x] \leftarrow 0$ to initialize the current *IDX* table and then performs $IDX_{j+1}[x] \leftarrow IDX_j[x]$ to bring the later IDX_j to the former IDX_{j-1}, and, finally, performs $IDX_j[\alpha_j[x]] \leftarrow \rho$ to set the index value from the current character of the suffixes, where α are the suffixes of S_0, such that α from a set of possible transition paths from root state S_0 to the states within length k_{root}, and can be defined as $\alpha \leftarrow \eta(S_0, k_{\text{root}})$. 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 appears in suffixes $\alpha_j[x]$, ρ will be put into that entry and increased by one.

S is the set of all AC states, and $|S|$ is the number of states built by 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 suffix in length j for state S_i. A transition function η can collect the possible $\beta_{i,j}$ from S_i to the states with length j.

Build_BitVector(*S*) builds the pre-hashing bit vector in the preprocessing phase, as seen in Fig. 9. This function first inputs the AC tree that is built by the conventional AC algorithm. Then, it extracts suffixes β_i within the length $k_{\text{pre-hash}}$ for the specific state S_i by using $\eta(S_i, k_{\text{pre-hash}})$, where $k_{\text{pre-hash}}$ is the maximum length of the pre-hashing suffixes, and is also the length of the substring in the text for each pre-hashing matching. β_i also includes the failure links in the AC tree. When suffixes are obtained, the pre-hashing algorithm hashes

```
V ← Build_BitVector(S) {
  For 1 ≤ i ≤ |S| {
      β_i ← η(S_i, k_pre-hash)
      For 1 ≤ j ≤ k_pre-hash {
          e ← 0
          For 1 ≤ x ≤ |β_i,j| {
              V_e,j ← H_j(β_i,j,x)
              e ← e + 1
          }
      }
  }
  Return V
}.
```

Fig. 9. Function for building pre-hashing bit vectors.

the 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.

4.3. Matching functions

In the matching phase of root-indexing, *Root_Index*(z) inputs a substring of the text z to locate the new state S_c in parallel, and is defined in Fig. 10. The lookup operation inputs $z[j]$ into $IDX_j(z[j])$ to generate a NA, repeatedly, which is defined as $NA \leftarrow NA \circ IDX_j[z[j]]$, where symbol $\circ$ is a concatenation operation. When NA is obtained, S_c is then looked up by $NEXT[NA]$.

In the searching phase, the pre-hashing performs $Pre - Hash(w, V_c)$ to rapidly match the current state in the AC tree, as shown in Fig. 11, where w is the current compared substring of the text, and V_c is the current bit vector.

The operation $TN_j \leftarrow H_j(w[1,\ldots,j]) \& V_{c,j}$ performs the bitwise AND (&) for $H_j(w[1,\ldots,j])$ and $V_{c,j}$, in order to return a true non-hit TN_j for length j. TN_j is 1 (True) if the hashed $w[1,\ldots,j]$ bit is set in $V_{c,j}$. The pre-hashing matching returns False (no match) when $\bigwedge_{j=1}^{k_{\text{pre-hash}}} TN_j \neq 1$, where the operation

```
S_c ← Root − Index(z) {
    NA ← NULL]
    For 1 ≤ j ≤ k_root {
        NA ← NA ∘ IDX_j[z[j]]
    }
    S_c ← NEXT[NA]
    Return S_c
}.
```

Fig. 10. Function for root-index matching.

```
Matched ← Pre − Hash(w, V_c) {
  For 1 ≤ j ≤ k_pre-hash {
      TN_j ← H_j(w[1..j]) & V_c,j
  }
  If ⋀_{j=1}^{k_pre-hash} TN_j ≠ 1 {
      Return False
  }
  Else {
      Return True
  }
}.
```

Fig. 11. Function for pre-hashing matching.

$\bigwedge_{j=1}^{k_{\text{pre-hash}}}$ is a conditional AND for multiple TN_j whose amount is $k_{\text{pre-hash}}$. This means that the longer w will not be further matched when the shorter w is not matched.

4.4. Hardware implementation

Xilinx ML310 is a FPGA based platform for SAM system as shown in Fig. 12. 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 SAM is implemented, the deep packet inspection of the packets is offloaded to SAM engine.

For the development tools, the Xilinx EDK and Synplicity SynplifyPro are used in the system implementation. The EDK can generate the bit streams

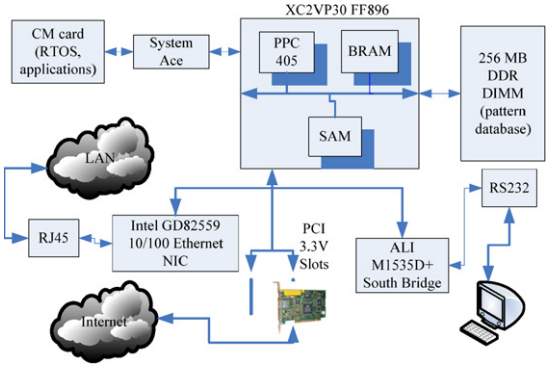


Fig. 12. Development platform for the SAM implementation.

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 SAM 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 as shown in Fig. 13.

- *FSM Unit* controls the working flow of the whole hardware system.
- *Root-Indexing Unit* is used for fast state indexing at the root state.
- *Pre-Hashing Unit* tests the bit vector for two 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.

The most important module in SAM hardware is the *FSM Unit*. Once the *SM controller Unit* is enabled, *FSM Unit* controls all other modules in parallel and its detailed operations are given in Fig. 14. In the FSM diagram, the starting state is the IDLE state. When the control signal is enabled, the FETCH state fetches a waiting text if the text

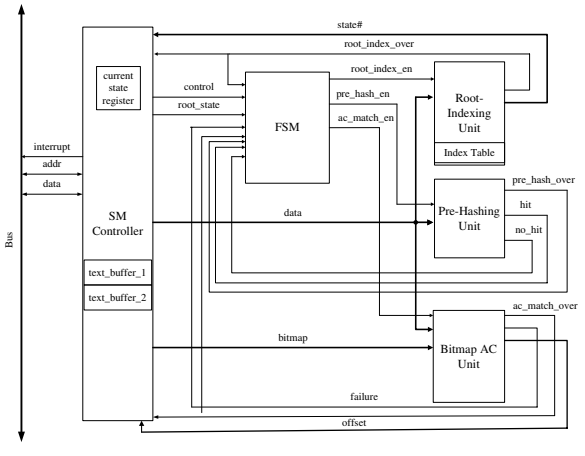


Fig. 13. The block diagram of proposed matching architecture.

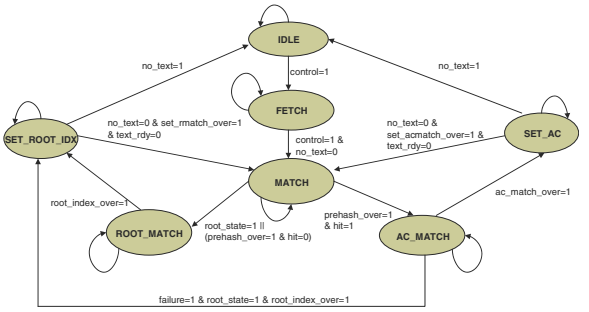


Fig. 14. The finite state machine of SAM hardware.

buffer is empty. Otherwise, the MATCH state will enable *Root-Indexing Unit*, *Pre-Hashing Unit*, and *Bitmap AC Unit* simultaneously.

If the current state is a root or the result of pre-hashing is non-hit, *FSM Unit* moves to ROOT_-MATCH to keep the *Root-Indexing Unit* working. Once the root-indexing matching is done, the current state will be assigned by *Root-Indexing Unit* at SET_ROOT_IDX state. Afterward, the *FSM Unit* will return to MATCH state to match subsequent texts. When a hit situation is reported, the *Bitmap AC Unit* and *Root-Indexing Unit* are triggered in AC_MATCH state, and the next state is assigned by root-indexing module if the current state of AC is required to set to root by failure link. Otherwise, the next state is provided by the *Bitmap AC Unit*.

As for the pattern updating, SAM can update patterns without interrupting the operation or shutting the machine down. Since the pattern is stored in the programmable memory, with the size of the current pattern sets and the download speed, the pattern data can easily be updated in a flash and thus, there is no need to shutdown the machine. In addition to that, SAM can also support the non-interrupting (incremental) update if the data structure of the SAM is ordered by state number.

5. Evaluation

5.1. Formal analysis

If pre-hashing, root-indexing and bitmap AC are to run as the sequential algorithm, the average time is

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

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

However, in the hardware, the pre-hashing, root-indexing 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

$$T_{avg\_time} = \frac{P_{root} \times T_{root} + (1 - P_{root}) \times T_{AC}}{(k_{root} \times P_{root}) + (1 - P_{root})}. \quad (2)$$

The number of state skipping depends on the pattern sets (form of automaton) and input data (network traffic). Hence, the objective performance should consider the average case.

Since AC matching is the critical path, the worst-case time of SAM is equal to T_{AC}, i.e.,

$$T_{worst\_time} = T_{AC}. \quad (3)$$

The probability P_{root} is an average probability to visit the root state and is equal to the average probability of the true non-hit. P_{root} is calculated by

$$P_{root} = \sum_{j=1}^{k_{pre\text{-}hash}} P_{tnc\_j}, \quad (4)$$

where P_{root} is computed by summing the conditional probabilities of a true non-hit P_{tnc_j}, which is the conditional probability of a true non-hit for length j. Now, if the $(j-1)$th pre-hashing is not matched, then the jth pre-hashing function cannot be matched either. P_{tnc_j} is determined from the unconditional probability of a true non-hit P_{tn_j}. P_{tn_1} is the first unconditional probability of a true non-hit P_{tn_j}, and can be obtained by

$$P_{tnc\_1} = P_{tn\_1}, \quad (5)$$

The subsequent P_{tnc_j} for length j can be computed by

$$P_{tnc\_j} = \left(1 - \sum_{y=1}^{j-1} P_{tnc\_y}\right) \times P_{tn\_j}, \quad (6)$$

where $\sum_{y=1}^{j-1} P_{tnc\_y}$ is a summing probability of the previous P_{tnc_j}. When the shorter suffix indicates a true non-hit, the longer suffix definitely outputs a true non-hit too. Hence, P_{tnc_j} is computed by subtracting the previous summing probability $\sum_{y=1}^{j-1} P_{tnc\_y}$, and multiplying by P_{tn_j}. P_{tn_j} is the unconditional probability of a true non-hit, which is referred from [24] as

$$P_{tn\_j} = \left(1 - \frac{1}{M}\right)^{|\beta_j|}, \quad (7)$$

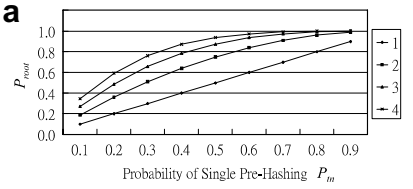
where $|\beta_j|$ is the number of suffixes for the corresponding length j, and M is the size of the bit vector. Pre-hashing intends to improve the probability of a true non-hit by increasing the non-matching suffixes. Thus, using one hashing function for each bit vector is sufficient and can significantly reduce hardware cost and latency.

In Fig. 15, notation P_{tn} is the same as the previous P_{tn_1} and P_{tn_j}. In our observation, a higher P_{tn} results in better matching performance because fewer text bytes need to perform AC matching. Fig. 15 shows the P_{root} value by computing Eq. (4) and it obviously shows that a short length of suffixes can also achieve an acceptable P_{root} whose value is larger than 0.4. Therefore, setting the maximum suffix length $k_{pre\text{-}hash}$ to 2 is sufficient. For example, when P_{tn} is set to 0.6 and $k_{pre\text{-}hash}$ is set to 2, P_{root} is equal to 0.84.

For the space evaluation, we need first of all to determine the bit vector size M. Since the probability of a true non-hit is defined in Eq. (7), M can be determined by given number of suffixes $|\beta|$ and P_{tn} as

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

Fig. 16. shows that M increases exponentially as $|\beta|$ grows and thus, M is feasible when $|\beta|$ is small.

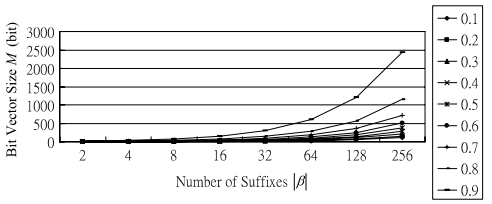


b

P_{root}		Length			
		1	2	3	4
P_{tn}	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

Fig. 15. (a) P_{root} simulation for length 1–4 and P_{tn} from 0.1 to 0.9. (b) The values for the diagram (a).

a



b

P_m \ M \ $\lvert\beta\rvert$	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

Fig. 16. (a) The simulation result of bit vector size M for P_m from 0.1 to 0.9, with the number of suffixes $|\beta|$ from 2 to 2048. (b) The values for the diagram (a).

The space requirement can be determined by summing the bitmap AC space Size_{AC}, the pre-hashing bit vector space $\text{Size}_{\text{pre-hash}}$, and the root-indexing space $\text{Size}_{\text{root}}$, as

$$\text{Size}_{\text{total}} = \text{Size}_{\text{AC}} + \text{Size}_{\text{root}} + \text{Size}_{\text{pre-hash}}. \quad (9)$$

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 $\text{Size}_{\text{state}}$,

$$\text{Size}_{\text{AC}} = |S| \times \text{Size}_{\text{state}}. \quad (10)$$

Each state size $\text{Size}_{\text{state}}$ includes one byte of state information, the failure and next state address $\text{Size}_{\text{state\_address}}$, and the size of the bitmap $\text{Size}_{\text{bitmap}}$ for locating the next state. Hence, $\text{Size}_{\text{state}}$ can be determined by

$$\text{Size}_{\text{state}} = 1 + \text{Size}_{\text{state\_address}} \times 2 + \text{Size}_{\text{bitmap}}. \quad (11)$$

The pre-hashing size $\text{Size}_{\text{pre-hash}}$ is determined from $\sum_{j=1}^{k_{\text{pre-hash}}} M_j$, which is the size of all bit vectors for one state, where M_j is a bit vector size for length j, and $k_{\text{pre-hash}}$ is the maximum length of the pre-hashing. $|S|$ is the number of states. Thus, $\text{Size}_{\text{pre-hash}}$ is obtained from

$$\text{Size}_{\text{pre-hash}} = \sum_{j=1}^{k_{\text{pre-hash}}} M_j \times |S|. \quad (12)$$

$\text{Size}_{\text{root}}$, which includes all root-indexing tables and the root next table. The size of all root-indexing 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 $\text{Size}_{\text{state\_address}}$. The number of root next state addresses is the cross product of the number of appearing alphabets in the index tables IDX_j and one zero entry. $\text{Size}_{\text{root}}$ is formulated as

$$\text{Size}_{\text{root}} = 256 \times k_{\text{root}} + \text{prod}_{j=1}^{k_{\text{root}}}(|IDX_j| + 1) \times \text{Size}_{\text{state\_address}}. \quad (13)$$

5.2. Simulation analysis

This simulation analysis can determine the performance of our simulation software. In our analysis, the test contents are execution files in Linux and Windows, as well as normal text files. The 32-bit bit vector and 1000 virus patterns are used to evaluate the proportion of root-indexing matching and bitmap AC matching.

There are two important factors which can affect the rate of the non-hit case. The first factor is the number of patterns. As the number of patterns increases, the branches of a node also increase. This means that the performance will be degraded by raising the rate of the hit portion. The second factor is the size of the bit vector for pre-hashing matching. The 8-bit bit vector is a choice for the development environment when the memory resource is limited, while the 32-bit bit vector has a better performance when enough memory is available.

After analyzing these two key factors, the non-hit rate for different sizes of the bit vector and the number of patterns in the three different data types are shown in Fig. 17. As the pattern set increases, the 32-bit bit vector has a better relative improvement than the 16-bit bit vector. In addition to the hit rate, the false positive rate of pre-hashing matching is also affected by the size of the bit vector. The false positive will lead to a little penalty in the clock cycles in the internal SRAM architecture.

For the proposed architecture, the 256-bit bitmap, 32-bit bit vector, two 8-bit width IDX table, one root next table, base address pointer of the next state table and failure state pointer are the data structures we used. Each state takes 384 bits and 336 bits to store data structures when the representation bit of the state number is 32 and 16 bits, respectively. For the overall memory usages, 303 kB and 265 kB are the combined mem-

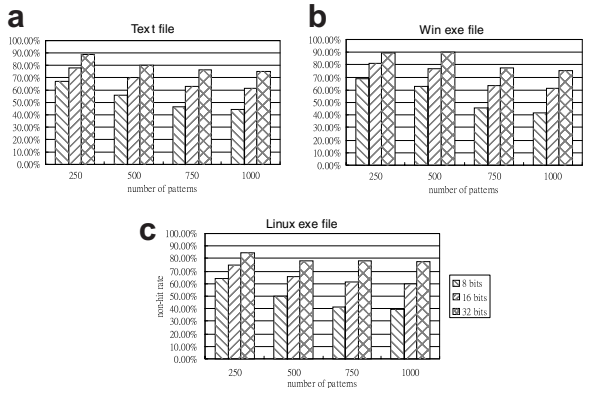


Fig. 17. The non-hit rate of 8-bit, 16-bit and 32-bit bit vectors for (a) text files. (b) Windows execution files. (c) Linux execution files.

ory usages for vector size 32 and 16 bits, respectively.

In addition to the above simulation, we also conducted two simulations for the large patterns, which are the static real patterns and the dynamic real network traffic simulation. The static simulation of real patterns assumes that the probability of non-hit for the text (network traffic) is a uniform distribution, i.e., the performance is no influence by network traffic, and that the result can be obtained with the equations of Section 5.1.

In this analysis, we chose the virus signatures from http://www.clamav.net. Since the virus signatures have a lot of patterns with long patterns as well, such patterns are sufficient to evaluate the performance of our SAM algorithm. The virus signatures have 10,000 patterns and generate 402,173 states. When $k_{\text{pre-hash}}$ and k_{root} are both 2, and the other parameters are assumed to be the same as the above-mentioned setting, then the conditional probabilities of true non-hit for length 1 and length 2 can be computed as $P_{tn\_1} = 0.29$ and $P_{tn\_2} = 0.14$. According to P_{root} equation, the probability of root-indexing matching is computed as $P_{\text{root}} = 0.43$.

As for the dynamic simulation of the real network traffic, the SAM performance is actually not only affected by the pattern sets but also the network traffics. Thus, the over 120 MB ethereal captured data were selected as the text to evaluate the above-mentioned URL patterns. Using the same conditions (10,000 virus patterns) with static real pattern simulation, P_{root} in the dynamic simulation of the real network traffic is 0.49, which is close to 0.43 in the static simulation.

5.3. Hardware analysis and comparison

As previously mentioned, our approach is flexible for both internal and external memory architecture. External memory architecture is suitable for large-pattern applications with modest throughput, such as the anti-virus and anti-spam applications. On the other hand, internal memory architecture can be used for high performance with fewer patterns, such as IDS and firewall applications.

The operating frequency of the synthesis result for our internal SRAM architecture is 350 MHz as reported by SynplifyPro. The root-indexing module takes 2 clock cycles to index a mapping state. The bitmap AC matching module takes 8 clock cycles per operation. Thus, the throughput can be estimated by the probability, frequency and processing bits per cycle. The best case throughput, wherein no byte has been matched, is 5.6 Gbps. The throughput in the average case, depending on the average proportion of the root-indexing matching and the bitmap AC matching, can be estimated at 5.37 Gbps. For the worst case, all bytes are matched in the text buffer. The throughput is 1.56 Gbps.

It is obvious that the average case has a very high performance, and is very close to that in the best case. It also has moderate performance in the worst case. This result demonstrates that our pre-hashing and root-indexing techniques are robust for high-performance packet inspection applications.

Compared to pure bitmap AC in hardware design, 96% of bitmap AC matching can be avoided by our two proposed techniques. This can be estimated by the portion of root-indexing, false-positive and non-hit cases at 96.18%. Furthermore, the throughput of pure bitmap AC hardware in the identical hardware environment can be estimated at 440 Mbps. Thus, our throughput is almost 7.65 times faster than the original bitmap AC in the average case.

Since our design is memory based architecture, in the implemented FPGA, Xilinx Virtex2P with speed grade 6 consumed only 1,688 LUTs and 106 Block RAMs, and are far less than that of other works. Compared to memory-based architecture work [17], the 384 bits of memory usage for each state is much less than their 8192 bits which use 256 32-bit pointers. Also, the operating frequency of 350 MHz does not decrease as the number and size of patterns grow.

Since many related works [11,12,14,24–26] employed the duplicate hardware for parallel pro-

cessing, the two engine architecture of SAM would be fair in this comparison. The double-engine SAM requires a simple control finite state machine (FSM) to coordinate the two single SAM controllers, and needs two extra cycles for each single SAM operation. The double-engine SAM has a slightly slow clock rate than the single engine SAM. Moreover, our optimally utilized dual port block RAM of the Xilinx FPGA not only doubles the performance, but it does not increase an extra block RAM.

The results demonstrate that our design has throughput at 10.7 Gbps and a support pattern of 21,563 bytes.

We compared and analyze about 12 major hardware from recent related works as shown in Fig. 18. The common goals for such kind of hardware are to pursue a higher throughput and a larger pattern size, which are the major evaluated factors in this comparison. Pattern sizes are used for measuring scalability with the unit in byte, and the throughput factor is used for measuring performance with the unit in giga bit per second (Gbps).

Nevertheless, 21,563 bytes is not the largest amount for the external memory version. The proposed SAM architecture is scalable to support more patterns with high performance, and SAM can be implemented with external multiple memory banks. Although external memory produces overhead for memory access, ASIC hardware can often run at a much higher speed than FPGA devices. For instance, the previous example with 21,302 patterns only ran at a clock rate of 800 MHz only to maintain about 10 Gpbs throughput with 35 MB memory requirement, which is quite feasible in today's technology.

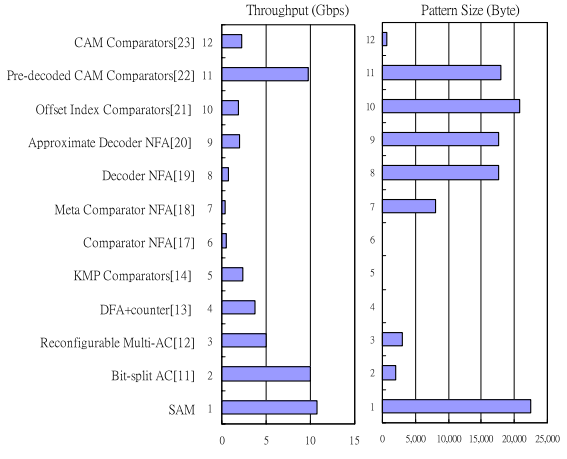


Fig. 18. SAM comparison with the other string matching hardware.

6. Conclusion

In this paper, we presented an architecture which takes scalability, flexibility and performance into consideration. Root-indexing and pre-hashing are the acceleration techniques used to improve the performance of our design. Also, our data structures are compressed and stored in either the internal SRAM or the external DRAM. The internal SRAM architecture provides an average of 10.7 Gbps throughput with the size limitation of patterns. The external DRAM architecture provides high scalability for the large number of patterns with acceptable throughput.

The internal SRAM architecture is implemented on the Xilinx Virtex2P FPGA-based platform. The string matching function of the target application ClamAV is also modified to set up the string matching engine. We tuned the hardware design according to the analysis results of our software simulation, and also built a prototype system for packet inspection applications such as IDS, URL blocking and ClamAV.

For a robust system evaluation, SAM should be operated in a real network environment for our future work. At the moment, the SAM implementation is a prototype system only and is not yet ready for field trial evaluation.

References

[1] S. Antonatos, K. Anagnostakis, E. Markatos, Generating realistic workloads for network intrusion detection systems, in: ACM Workshop on Software and Performance, Redwood Shores, CA, Jan. 2004.
[2] G. Navarro, M. Ranot, Flexible Pattern Matching in Strings, Cambridge University Press, 2002.
[3] G. Navarro, A guided tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.
[4] S. Wu, U. Manber, Fast text searching allowing errors, Communication of the ACM 35 (1992) 83–91.
[5] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (10) (1977) 762–772.
[6] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM, 1975, pp. 333–340.
[7] N. Tuck, T. Sherwood, B. Calder, G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection, IEEE Infocom, Hong Kong, China, 2004.
[8] C. Coit, S. Staniford, J. Mcalerney, Towards faster string matching for intrusion detection, DARPA Information Survivability Conference and Exhibition 2002, pp. 367–373.

[9] N. Desai, Increasing performance in high speed NIDS, <http://www.snort.org/>.
[10] M. Raffinot, On the multi backward Dawg matching algorithm (MultiBDM), Workshop on String Processing, Carleton U. Press, 1997.
[11] L. Tan, T. Sherwood, A high throughput string matching architecture for intrusion detection and prevention, ISCA, 2005.
[12] M. Aldwairi, T. Conte, P. Franzon, Configurable string matching hardware for speeding up intrusion detection, ACM CAN, 2005.
[13] J. Lockwood, An open platform for development of network processing modules in reconfigurable hardware, IEC DesignCon, Santa Clara, CA, Jan. 2001.
[14] J. Moscola, J. Lockwood, R.P. Loui, M. Pachos, Implementation of a content-scanning module for an internet firewall, IEEE FCCM, 2003.
[15] Z.K. Baker, V.K. Prasanna, Time and area efficient pattern matching on FPGAs, ACM/SIGDA FPGA, CA, USA, Feb 2004.
[16] G. Tripp, A finite-state-machine based string matching system for intrusion detection on high-speed network, EICAR, May 2005.
[17] L. Bu, J.A. Chandy, A keyword match processor architecture using content addressable memory, ACM VLSI (April) (2004) 26–28.
[18] R. Sidhu, V. Prasanna, Fast regular expression matching using FPGAs, IEEE FCCM April 2001.
[19] R. Franklin, D. Carver, B.L. Hutchings, Assisting network intrusion detection with reconfigurable hardware, IEEE FCCM, Napa, CA, Apr 2002.
[20] C.R. Clark, D.E. Schimmel, Scalable pattern matching for high speed networks, IEEE FCCM, 2004.
[21] Y.H. Cho, W.H. Mangione, A pattern matching coprocessor for network security, ACM/IEEE DAC, California, USA, June 2005.
[22] I. Sourdis, D. Pnevmatikatos, Pre-decoded CAMs for efficient and high-speed NIDS pattern matching, IEEE FCCM, 2004.
[23] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, V. Hogsett, Granidt: towards gigabit rate network intrusion detection technology, LNCS 2438 (Jan) (2002).
[24] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, J.W. Lockwood, Deep packet inspection using parallel bloom filters, IEEE Micro. 24 (1) (2004).
[25] H.M. Blüthgen, T. Noll, R. Aachen, A programmable processor for approximate string matching with high throughput rate, IEEE ASAP, 2000.
[26] J.H. Park, K.M. George, Parallel string matching algorithms based on dataflow, HICSS, Hawaii, 1999.