國立交通大學

資訊科學與工程研究所

碩士論文

以根索引及預先雜湊加速自動機式字串比對硬體： 設計，實作，與評估

Automaton Based String Matching Hardware with Root-Indexing and Pre-Hashing Techniques: Design, Implementation and Evaluation

研 究 生：洪振洲

指導教授：林盈達 教授

中 華 民 國 九 十 五 年 六 月

以根索引及預先雜湊加速自動機式字串比對硬體：設計，實作，與評估

Automaton Based String Matching Hardware with Root-Indexing and Pre-Hashing Techniques: Design, Implementation and Evaluation

研 究 生：洪振洲　　Student: Chen-Chou Hung

指導教授：林盈達　　Advisor: Dr. Ying-Dar Lin

國立交通大學

資訊科學與工程研究所

碩士論文

A Thesis

Submitted to Institutes of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
In

Computer Science and Engineering

June 2006

HsinChu, Taiwan, Republic of China

中華民國九十五年六月

以根索引及預先雜湊加速自動機式字串比對硬體 ： 設計，實作， 與評估

學生：洪振洲　指導教授：林盈達

國立交通大學資訊科學與工程研究所

摘要

字串比對在內容過濾應用程式中扮演相當重要的角色，例如入侵偵測、防毒、擋廣告信和網站過濾，這不僅相當耗時並且佔用大量記憶體，純軟體演算法的實作效能亦無法滿足這類應用程式的高速需求，因此將封包內容交給字串比對硬體去處理是個必然的趨勢。這篇論文主要著重於一個節省記憶體的 AC 變形演算法，即 位圖 AC（bitmap AC），並且使用了兩個平行硬體加速的技巧，一個是跟索引，另一個是預先雜湊，進一步實作在 Xilinx ML310 FPGA 平台上。根索引能夠一次比對多個字元，而預先雜湊能夠避免耗時的位圖 AC 計算。此外，這篇論文所提的方法對於內部記憶體或是外部記憶體的架構都是相當適合的，採用內部記憶體的架構可以提供高效能，而外部記憶體的架構則提供更高的應用範疇，這兩種實作方式都具有優越的處理能力。

關鍵字: 字串比對, 位圖 AC, 預先雜湊, 跟索引, 基於記憶體架構, 高範疇

Automaton Based String Matching Hardware with Root-Indexing and Pre-Hashing Techniques: Design, Implementation and Evaluation

Student: Chen-Chou Hung Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao Tung University

Abstract

String matching plays a central role in content networking applications, such as intrusion detection, anti-virus, anti-spam and Web filtering. As it is computation and memory intensive, a software implementation of string matching algorithms is insufficient to meet the high-speed requirement of these applications. Thus, offloading the packet content to dedicated hardware seems inevitable. This thesis focuses on the memory efficient version of AC, namely bitmap AC, and employs two parallel hardware acceleration techniques, root-indexing and pre-hashing, onto the FPGA-based Xilinx ML310 platform. 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. Furthermore, the proposed string matching hardware approach is feasible for either internal or external memory architecture. The internal memory architecture provides high performance, and the external memory architecture provides high scalability of patterns. These two implementations both have high throughput for matching.

Index Terms: string matching, bitmap AC, pre-hashing, root-indexing, memory-based architecture, high scalability

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Because detecting malicious traffic on the Internet, such as viruses and intrusions, relies on looking for signatures in the packet payload, traditional firewalls inspecting only the packet header are insufficient for the detection. Thus, deeper packet-content inspection is required to detect such application-level attacks, such as intrusion detection, anti-virus, anti-spam and Web filtering which are available in the market. The essential part of these solutions is string matching, and it has been shown to be a time-consuming component that should be accelerated [1], [13].

For string matching, there are several well-known algorithms such as Aho-Corasick (AC) [2], Wu-Manber [3], Boyer-Moore [4] and Bloom filters [5]. A software implementation of these algorithms may not be able to afford escalating traffic in the Internet, so several hardware architectures such as discrete comparators [6]-[8], CAM-based architecture [9]-[11], and Bloom filters hardware [12]-[13], have been proposed to dramatically improve the performance. The throughput of them can mostly achieve up to 10 Gbps, but the common drawback of them is the poor scalability. Their rules and pattern sets are hardwired into the FPGA, so the scalability is limited by the number of logic cells and the size of embedded memory in the FPGA.

In this thesis, we proposed a scalable memory-based hardware architecture which is based on the AC algorithm. AC is a common algorithm with following the key features. First, it has the linear time performance in worst case. Second, it is robust for large and lengthy patterns. Third, it can perform multi-pattern match. However, the most critical defect of AC algorithm is large memory usage. The other

AC-based algorithm, bitmap AC [14], improved the memory utilization by using a 256-bit bitmap to replace 256 word-size pointers of each state in AC. Therefore, bitmap AC is the alternative which we adopted in this work. We also applied two acceleration techniques to make our architecture sub-linear. The first technique is Root-Indexing which comes from the observation of AC's high frequency root-visiting behavior. The second technique is Pre-Hashing which comes from the observation of time-consuming operation in bitmap AC, which is also high cost on the x86-platform. Thus, for reducing this kind of operation, Pre-Hashing can test quickly to avoid the bitmap AC matching. For scalability, our architecture use either internal or external memories to store the whole pattern database of SNORT [19] or even ClamAV [20]. Furthermore, it can easily update pattern without interrupting operation or even shutting the machine down.

In our evaluation, we implemented our work on FPGA-based platform Xilinx ML310, and used ISE and EDK as our design tools. Also, we provided the driver interface for communication between software and hardware component. In the hardware design, we divided the whole architecture into five main modules, which can be paralleled. The first module is the parallel finite state machine which controls signals among the other modules. Next is the string matching controller module which is in charge of data request and assignment. The last three modules are root-indexing, pre-hashing, and the bitmap AC matching.

The rest of this paper is organized as follows. In Chapter 2, we first introduce the related algorithms and issue. Then we introduce the existing hardware acceleration architectures and list the comparison table. Our acceleration techniques are presented and an example is given in Chapter 3. Chapter 4 describes the hardware architecture design of our approach and software interface we provide. The performance evaluation, analysis, and comparison with existing works are presented in Chapter 5.

Finally we make conclusion and list the future work in Chapter 6.

Chapter 2

Background

2.1 AC related algorithms

For content filtering applications, there are a lot of string matching algorithms such as AC, Boyer-Moore and Wu-Manber. Due to the AC-based algorithm is used in our work, we only introduce AC-based algorithm, its key features, and its issues in this chapter.

2.1.1 Aho-Corasick

One of the notable algorithms in exact pattern matching is Aho-Corasick, which is able to match multiple strings with the linear time is the worst case. AC works by constructing a state machine from the patterns to be matched, and is composed of three component functions. The first is *Goto* function which is used to traverse from one node to the other node as shown in Figure 1 (a). The second is *Output* function which output the match pattern for current state as shown in Figure 1 (b). The third is *Failure* function which is traversed when there is no next state, shown in Figure 1 (c).

Constructing the state machine starts with an empty root node and adds states to the state machine for each pattern. Failure pointers are further added from each node to the longest prefix of that node which also leads to a valid node in the trie. After the construction of machine, the state machine is then traversed from the current node to the next node according to the input character. There are two alternatives to store the next state links.

1. One is the construction of 2D-array table. Each state has 256 next state

pointers for all ASCII input case, as shown in Figure 1. (d). It is the most popular implementation for fast matching, but it wastes the memory space when the table is sparse.

2. The other data structure is using link list, and each state only has the link list of exist next states. This kind of data structure has smaller space requirement but is slow when there are many next states.

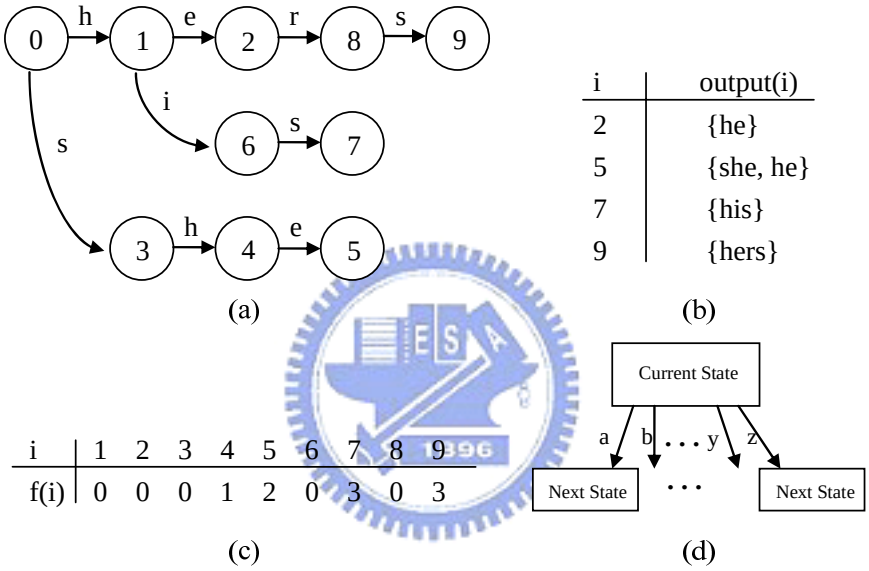


Fig. 1. (a) *Goto* function. (b) *Output* function. (c) *Failure* function. (d) AC table implementation.

2.1.2 Bitmap AC

A variation of AC, bitmap AC, is a compromise between table and link list approaches [13]. It maintains a 256-bit bitmap for each state to indicate whether a traverse with a given character is valid or invalid which requires traversing along the failure pointer path. If the next state is valid, then the next state is obtained by summing all the bits prior to that bit number and adding them to the base address of next node pointer. Figure 2 shows the data structure of bitmap AC and how it locates the next state.

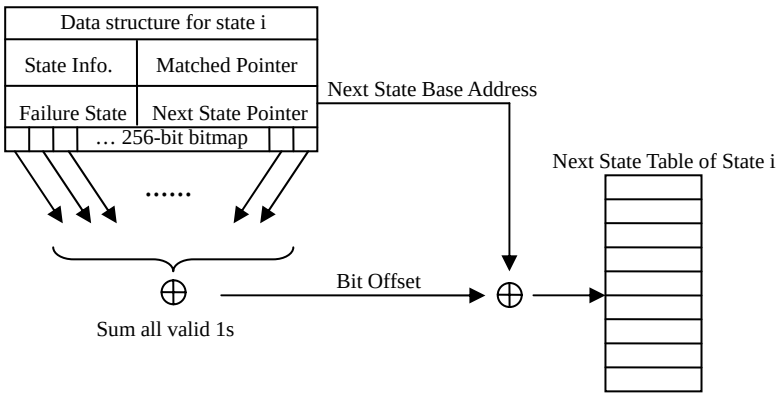


Fig. 2. Data Structure of bitmap AC for state i and using bitmap to locate the next state.

The critical defect of AC table implementation is the waste of memory which using 256 next pointers for each state. Bitmap AC solves this problem, and still keeps the advantages of AC. However, in order to locate the next state in bitmap AC, we must to count all 1s before the valid bit in the 256-bit bitmap. This is known as a time-consuming operation which is the high cost on x86 based systems.

2.2 Hardware-based string matching

The major string matching hardwares are listed in Table 1. The most common matching hardware is the finite automaton approach which uses DFA/NFA to match every input byte. Sidhu and Prassanna [13] introduced the hardwired Nondeterministic Finite Automata (NFAs) for finding matches to a given regular-expression pattern. They implemented on FPGA which matches one text byte per clock cycle. However, this kind of works has lower clock rate and throughput.

Another approach is CAM-based architecture, which is able to match all contents at once to achieve high throughput. Nowadays FPGA often has embedded block

RAMs for high performance design. Thus, CAM is easily constructed from block RAMs. For the CAM-based hardware, Cho [7] proposed a CAM-based solution which uses comparators to perform initial part of pattern matching and uses the matched prefix as an address into a CAM to read the whole patterns. The other CAM-based solution is pre-decoding CAM, which was proposed by Baker for processing a large-rule set [9]. Some CAM-based implementations [10], [16] also combine with hardware comparators for lower usage of circuit and high performance.

In the last two matching hardware, memory-based hardware was also presented by Monther and Thomas [17]. They constructed an AC table which adds the extra failure links of longest prefix, and stores them in external memory. However, the memory requirement is too large in their design. Another different architecture is Bloom filters hardware [12]-[13], which uses multiple hash functions for approximated matching. Once Bloom filter reports a possible hit, the advanced verification is needed for the exact matching. The main drawback of Bloom filters hardware is that dedicated length processing unit is needed for every pattern length.

Most of these aforementioned works can achieve up to 10 Gbps by hardwiring the rule sets into FPGA, which limits the scalability of rules and the size of patterns. Even the data structure of memory-based design wastes too much memory space. Therefore, the scalability of patterns and rules is our most concerned issue, and it is our focus in this work.

Table 1

Comparison of existing string matching hardware

Matching Hardware	Advantage	Disadvantage
NFA / DFA Hardware	Easy to implement Regular expression support	High area cost Modest throughput
CAM-based Hardware	High throughput	High area cost
Bloom Filter Hardware	Low area cost	False positive issue Fixed length
Memory-based Hardware	Reconfigurable High capacity	Low throughput Memory space wasting

Chapter 3

Fast Bitmap AC String Matching Hardware

3.1 Overview

This thesis is base on the Tseng's string matching approach [18]. It can match multiple characters at root state by root-indexing matching, and avoid some slow bitmap AC matching operations by pre-hashing matching. Also these two acceleration techniques can process in parallel. An example is illustrated from Fig. 3 to 5, which shows the difference between bitmap AC and fast bitmap AC.

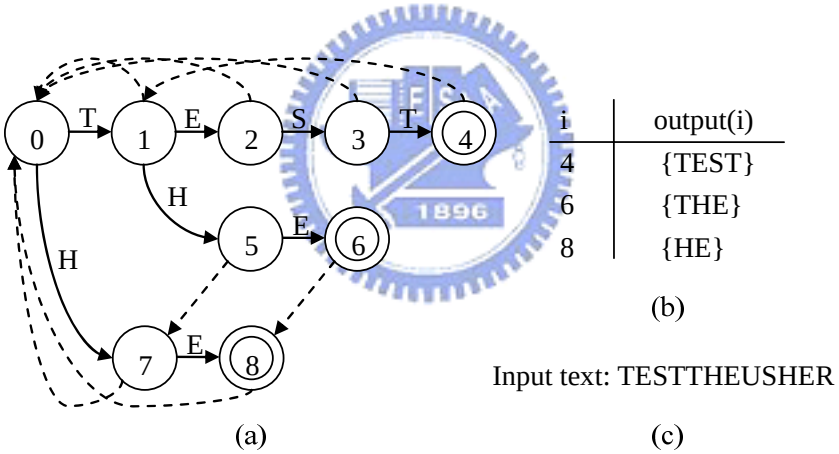


Fig. 3. An example for fast bitmap AC. (a) *Goto* Trie. (b) *Output* function. (c) Input text.

The Fig. 3 is an example of original AC that can be used for bitmap AC and fast bitmap AC. The transition of AC or bitmap AC will both go to next state according to the given byte. Their transition sequences according to the previous described AC and bitmap AC algorithms are the same, as shown in Fig. 4.

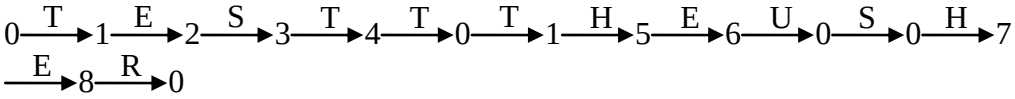


Fig. 4. State transition sequence of conventional AC.

Since our fast bitmap AC approach had implemented root-indexing and pre-hashing techniques, our transition sequence, which is different from original AC and bitmap AC, is shown as Fig. 5. The “RI” symbol means root-indexing.

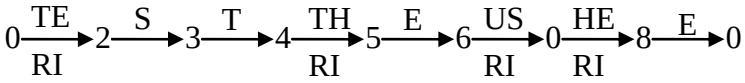


Fig. 5. State transition sequence of fast bitmap AC.

Root-Indexing mechanism can process two or even more bytes at the same time, and is applied to the root state in our design. Therefore, starting from the root state, the next state is decided by root-indexing. Beyond the root state, pre-hashing is used to quickly examine the existence of next state for every state transition. When the pre-hashing unit reports no match for given two bytes, the next state is determined by the root-indexing unit. Otherwise, the next state is decided by the bitmap AC unit. It is worth noting that because of parallel processing, when given two characters “TH” at state 4, the next state 5 will be directly obtained from the root-indexing unit.

3.2 Root-Indexing Matching

In AC trie, most of failure links point to the root state, that is, it will always go back to the root state when there is no any next state for a given character. Thus, it is efficient to apply the root-indexing in the root state. Root-indexing can match

multiple characters simultaneously at the root state. In Fig. 6, root-indexing comprises *k* index tables $IDX_{[1...k]}$ and a root next table *NEXT*, where *k* denotes the maximum length of root-indexing matching in the same time. Each entry of *IDX* stores a partial address for locating the next state in *NEXT*. The partial address is an unique sequential integer to represent the order of appearing characters for the corresponding substrings in the suffixes of root state. Note that, for advancing *k* characters in matching iteration, the substring is started from current byte to *k*, which means the latter *IDX* table is required to include the entries of the former *IDX* tables.

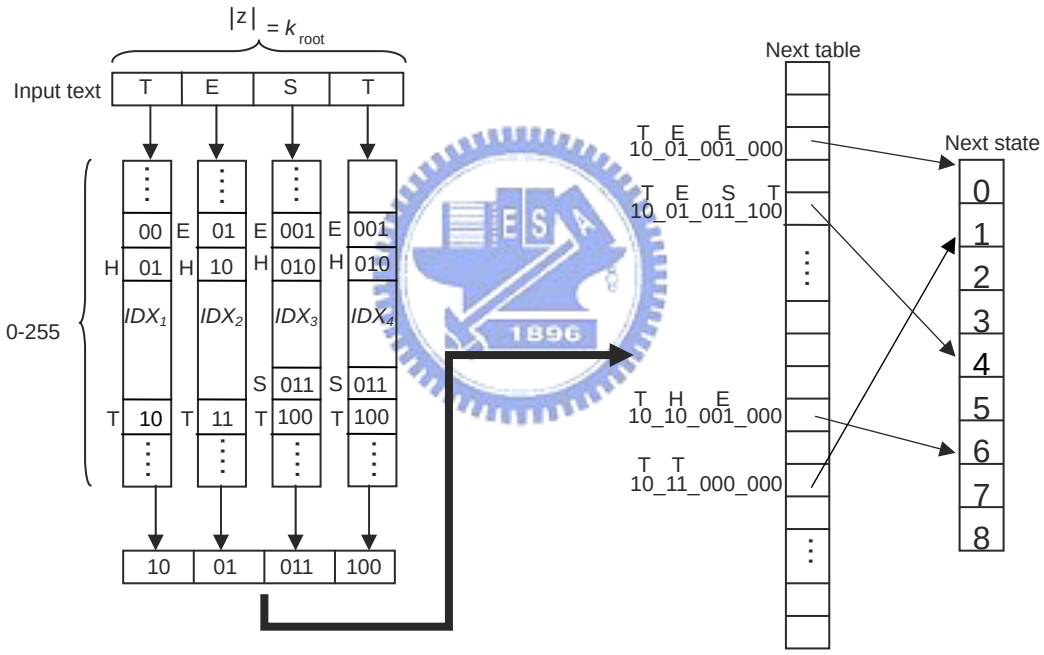


Fig. 6. Root-indexing architecture and example for the input text “TEST” with the patterns “TEST”, “THE” and “HE”.

For example, if patterns are “TEST”, “THE” and “HE”, IDX_1 to IDX_4 will at least contains 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, because the latter tables are required to contain the entries of former tables, IDX_1 to IDX_4 will 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 first *IDX* have 2 appearing characters, thus “H” and “T” are numbered as “01” and “10” in binary format. The second *IDX* table using “01”, “10” and “11” stands for {“E”,”H”,”T”}. The *NEXT* table is used to store all the next states within length *k*, and it is indexed by a concatenation address of lookup value from the all *IDX* tables. In the example of Fig. 6, 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”.

3.3 Pre-Hashing Matching

The pre-hashing method can quickly examine the existence of next state to avoid further slow AC matching. It uses a single hashing function and builds the bit vector for the substrings of each state. When performing the pre-hashing, the next state will be obtained from root-indexing unit instead of from bitmap AC unit if true negative is indicated by the pre-hashing unit. True negative is the condition that the given character is absent in the pre-hashing vector for the suffix of the current state.

Before the pre-hashing matching, it is necessary to build the pre-hashing bit vector in the preprocessing phase. First, we input the AC trie which is built by the conventional AC algorithm. For each state, we extract suffixes within the length 1 which is different from the Tseng’s original design. Recursive failure link of each state except the link to root state is also included in these suffixes. This can avoid filling the bit vector to almost all 1 when number of patterns is large that will lead to high hit rate issue. When suffixes are obtained, the pre-hashing algorithm hashes suffixes into bit vectors. This procedure of building the bit vectors for state 1 in Fig. 7(a) is

illustrated in Fig. 7(b). The mask of rightmost four bits of the characters and transformation from binary to one-hot representation are used as the hash function in our design. However, better mask position is adjustable for lower false positive according to the characteristic of patterns.

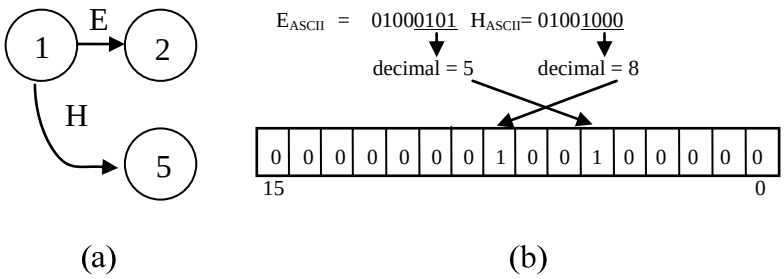


Fig. 7. (a) AC trie of state 1 for building bit vector. (b) Example of building the bit vector for state 1 in the preprocessing phase.

A pre-hashing matching example is shown in Fig. 8. The pre-hashing unit reads a byte substring and then hashes the substring “G”. The hash result will be indicated by the pre-hashing unit, when the pre-hashing unit indicates non-hit, the next state 5 for substring “GDTH” will be obtained from the root-indexing unit. However, if the hit condition is indicated by pre-hashing unit, the slow bitmap AC matching will be performed.

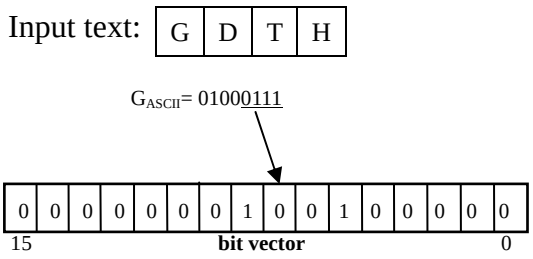


Fig. 8. Example for hashing at state 1.

Chapter 4

Implementation

4.1 Overview

To verify the correctness of fast bitmap AC, we write the C simulation model, and also use it to generate the essential data structures from patterns. These data structures will be loaded to initialize matching hardware. After initializing of our system, the user applications call the string matching API offered by the driver to start the matching operation. Once a text buffer is scanned, the interrupt signal will be triggered by the matching engine. Afterward, the interrupt handler will be invoked to check the matching results and also fill the new text to the buffer if it exists.

For hardware design, it is partitioned into five main modules, and each one is in charge of a specific function. The implementation environment is based on the Xilinx ML310 FPGA platform, and Xilinx's ISE, EDK and Synplicity's SynpifyPro are development tools for embedded software/hardware integration. The MontaVista RTOS is chosen for our system, and ClamAV is the target application for our content filtering system.

4.2 Pre-Processing and Simulation Software

The pre-processing procedure generates the essential data structures for the proposed hardware, as shown in Fig. 9(a). The *Make_Goto() and Make_Failure()* functions are the original functions defined by the AC algorithm, and our data structures are further 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 $IDX_{[1...k]}$ 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 function according to all next states of both current state and recursive failure node for pre-hashing preprocessing.

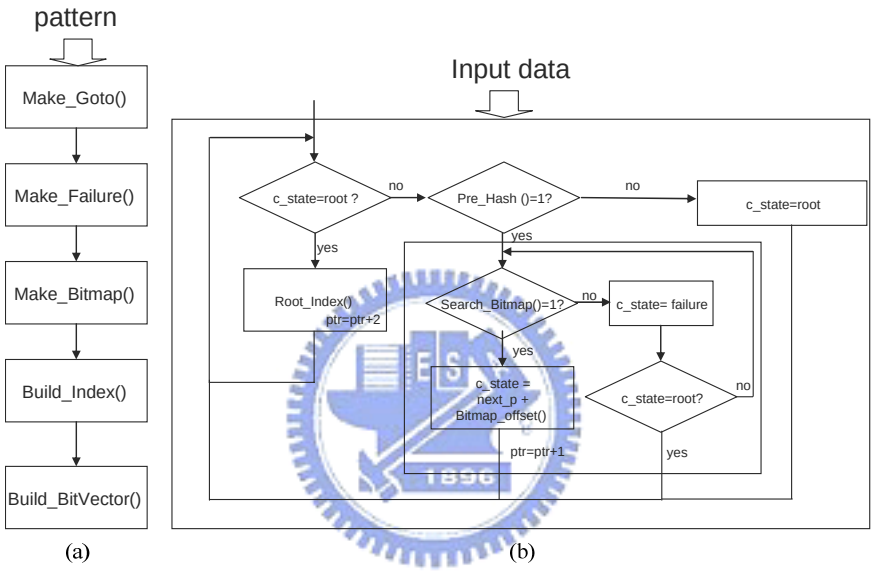


Fig. 9. (a) The pre-processing procedure. (b) The flow of C simulation model.

After the pre-processing procedure is finished, the simulation of proposed fast bitmap AC algorithm can perform matching according to the flow in Fig. 9 (b). For each matching iteration, it checks the current state at first. When the current state is in the root state, the *Root-Index()* matching will be performed, otherwise *Pre_Hash()* will be performed. If *Pre_Hash()* reports the non-hit situation, the current state will be set to root state directly, and do the root-indexing matching. If the hit situation is reported, *Search_Bitmap()* will check the existence of 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 is root state. This C model can be the golden model for the proposed hardware design, and it also can be used to gather statistics for performance analysis.

4.3 Matching Hardware

This subsection introduces the proposed hardware architecture, block diagram and FPGA platform individually.

4.3.1 Architecture

The proposed architecture is a highly parallel design that all modules are working at the same time, and this architecture is also flexible for either internal or external memory-based platform. The block diagram of our proposed architecture is shown as Fig. 10.

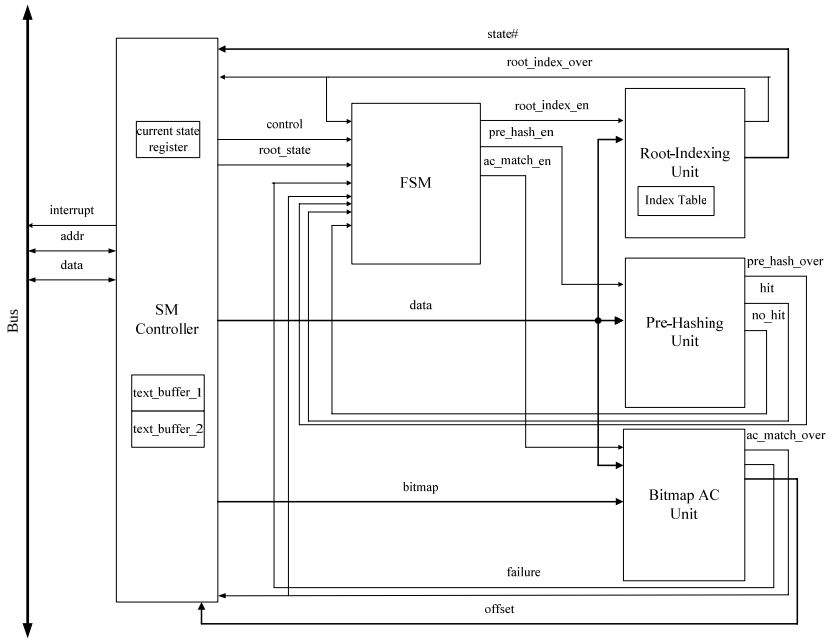


Fig. 10. The block diagram of proposed fast bitmap AC architecture.

1) FSM module: The most important part in our architecture is FSM module to control the working flow of whole hardware system. Once the SM controller is enabled, FSM will control all other modules in parallel. The detailed state translations are shown in Fig 11.

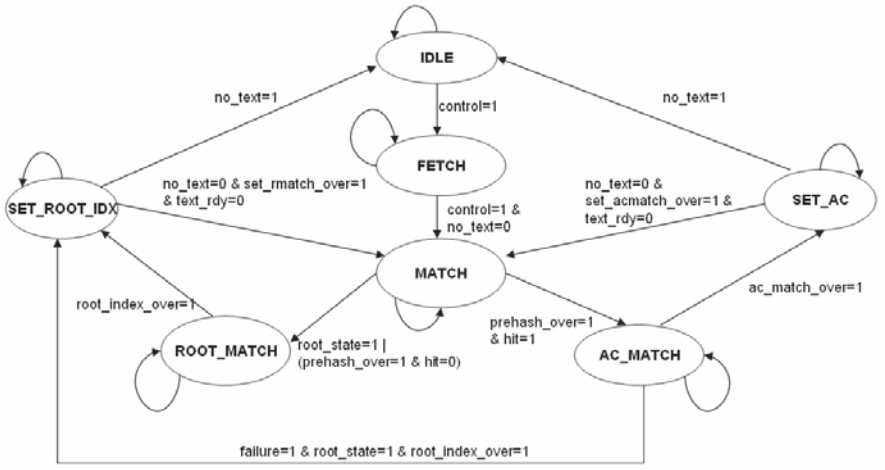


Fig. 11. State transition diagram of FSM module.

In this FSM diagram, the starting point is the *IDLE* state. When the control signal is enabled, the *FETCH* state will fetch waiting-scan text if the text buffer is empty. Otherwise, the *MATCH* state will enable the root-indexing, pre-hashing, and bitmap AC matching units simultaneously. If the current state is at root or the result of pre-hashing is non-hit, the control state of the FSM will translate to *ROOT_MATCH* to keep the root-indexing module working. Once the root-indexing matching is done, the current state will be assigned by root-indexing module at *SET_ROOT_IDX* state. Afterward, FSM will return to *MATCH* state to match subsequent texts. When the hit situation is reported, the bitmap AC matching and root-indexing matching will be triggered in *AC_MATCH* state, and the next state will be assigned by root-indexing module if current state of AC trie is required to set to root by failure link. Otherwise,

the next state will be provided by bitmap AC matching module.

2) Root-Indexing module: This module is used for fast state indexing at the root state. It contains internal RAMs for index tables and root next table. For the practical memory and throughput considerations, it only matches two characters at the same time. For a large number of patterns, two characters can used to directly index the next state from *NEXT* table when the next states of root is over 128. In this case, it takes only one clock cycle. For a small number of patterns, the given two characters will first index an encoded address from *IDX* table, and the obtained address can used to index the next state from *NEXT* table. This can save large memory space, but it takes two clock cycles to index a next state. After finishing the root-indexing matching, this module will output the next state to SM controller.

3) Pre-Hashing module: The pre-hashing module will test the bit vector for two input bytes by hashing function and send the hashing result to FSM. For external memory architecture, the bit vector which is stored in internal memory can save large time to fetch 256-bit bitmap when hash result is missing.

4) Bitmap AC matching module: When this module is enabled by FSM, it will firstly check the corresponding bit for input byte. If the corresponding bit is 1, then it will mask off the unnecessary bits and count all 1s for locating the next state. Otherwise, it will issue the failure signal and notify the controller to set the failure state as the current state.

5) SM controller module: This module plays an important role between system bus and the whole string matching module. It provides the control registers including length of text buffer and enable signal for software to program. Besides, it also contains two text buffers and two matching-result buffers for content applications. After a buffer is scanned, the SM controller will trigger the interrupt signal, and the application will read out the matching result if it exists and fill the new text. For the

whole string matching module, it provides the input bytes from text buffers and feeds necessary data structure to each module.

4.3.2 FPGA Implementation

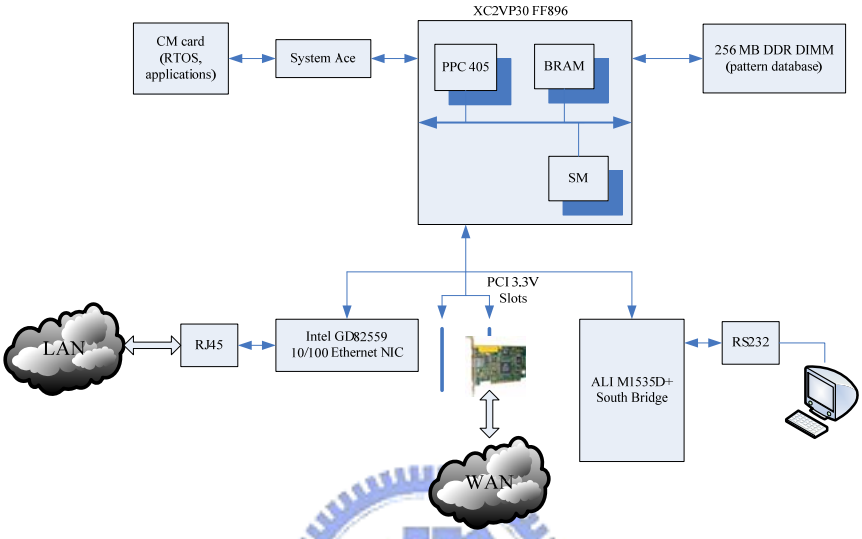


Fig. 12. The architecture of ML310 platform.

We use the Xilinx ML310 FPGA based platform as our development system, as shown in Fig. 12. This platform has 2448 Kbits internal block RAM, 30816 LUTs and also two hardwired IBM PPC405 processors in FPGA. For the peripheral, we use 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 file system. The packets will be inputted from on board Ethernet port, processed by the PPC 405 CPU. Also the packet content is offloaded to string matching engine. Finally, the clean traffic will output from the NIC of PCI extension.

The MontaVista Preview Kit is chosen as our RTOS. Xilinx EDK, ISE and Synplicity's SynplifyPro are the basic development tools. The EDK can generate BSP and bit stream file for our system design. The BSP including mapping address define files and drivers of all peripherals for building the complete RTOS image. As RTL

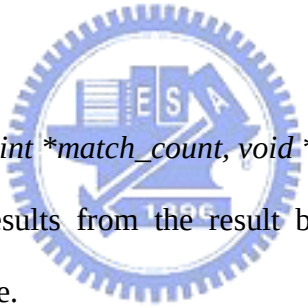
code design for string matching hardware, ModelSim and Debussy are simulator and debugger tools we used, respectively.

4.4 Driver Interface

We also provide the driver interface for communication between hardware and software. The detailed driver functions are listed below.

*Write_Buff(unsigned int *length, char *buff, void *base_addr);*

This function writes the text to buffer for scanning, and also specifies the length and address of target buffer.



*Read_Match_Result(unsigned int *match_count, void *base_addr);*

It reads the matching results from the result buffer and the application will specify the matched virus name.

*Intr_Handler(void * baseaddr_p);*

When the interrupt signal is triggered, this function will be invoked to do the matching result checking and text buffer writing. Therefore, *Write_Buff()* and *Read_Match_Result()* will be called.

*Start_Matching(char *buffer, unsigned int length);*

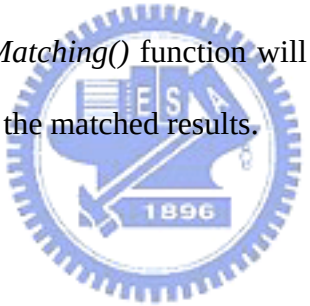
This function is called by the ClamAV string matching function. It will write the waiting-scanned text to the buffer, and setup the text ready register to enable the matching operation.

*Stop_Matching(void * baseaddr_p);*

This function will be called when the applications is closed. It clears the string matching control register to stop the operation.

*cli_ac_scanbuff(const char *buffer, unsigned int length, int *vir_id, struct ac_status *status);*

This is the API of ClamAV to perform the string matching. It specifies the address and length of text buffer which is located by the ClamAV, and returns matched virus IDs and the matched status. It will divide the buffer pointed by **buffer* to several portions depending on the size of text buffer we used, and match them sequentially. Thus, the *Start_Matching()* function will be called in a *for* loop to scan each text partition, and returns the matched results.



Chapter 5

Evaluation

5.1 Simulation Analysis

This simulation analysis can determine the performance of our design by using the software simulation flow described in Chapter 4. In our analysis, the test contents are execution files in Linux, and Windows, and normal text files. The 32-bit bit vector and 1000 virus patterns are used to evaluate the proportion of root-index matching and bitmap AC matching, shown as Fig. 13 (a). The high proportion of fast root-index matching can improve the performance.

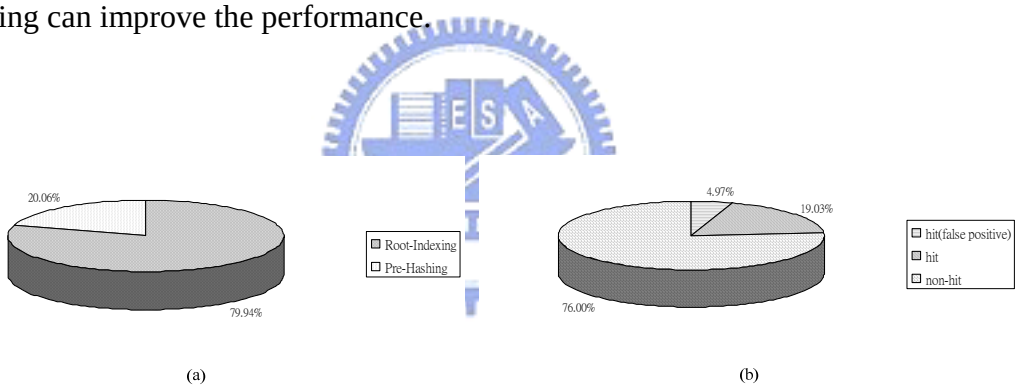


Fig. 13. (a) The proportion of root-indexing and pre-hashing. (b) The proportion of hit, non-hit and false positive.

The pre-hashing portion in Fig. 13(a) can be divided into three sub-portions as shown in Fig. 13(b). The first and second are hit and false positive portions, which have 24% and 12 % and must perform the slow bitmap AC matching operation. The third is non-hit portion, which has 64% and performs the fast root-indexing matching. Thus, as the proportion of non-hit increases, the performance upgrades.

There are two important factors which will affect the rate of the non-hit case. The first factor is the number of patterns. As the increasing of number of patterns, the

branches of a node increases. This means that the performance will be degraded by raising the rate of the hit portion. The second factor is the size of bit vector for pre-hashing matching. For the balance of performance and memory usage, the bit vector size can be adjusted in the preprocessing. A reasonable size is 8 bits or 32 bits for both practical considerations. The 8-bit bit vector is a choice for the development environment when memory resource is limited, and the 32-bit bit vector has better performance when memory resource is available. For analyzing these two key factors, the non-hit rate for different sizes of bit vector and the number of patterns in three different data types are shown in Fig. 14. As the increase of pattern set, 32-bit bit vector has more apparent improvement than 16-bit bit vector.

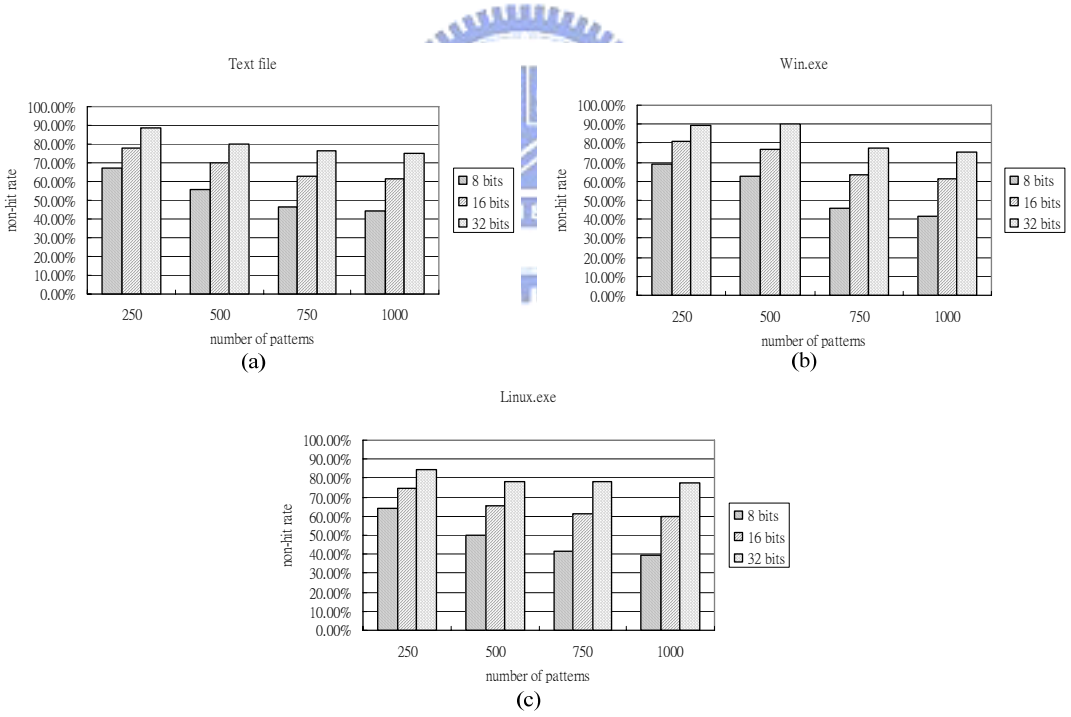


Fig. 14. The non-hit rate of 8-bit, 16-bit and 32-bit bit vectors for (a) text files, (b) Windows execution files, (c) Linux execution files.

In addition to hit rate, the false positive rate of pre-hashing matching is also affected by the size of bit vector, as shown in Fig. 15. The false positive will lead to a

little penalty of clock cycles in the internal SRAM architecture, and great penalty of bus contention for external DRAM architecture.

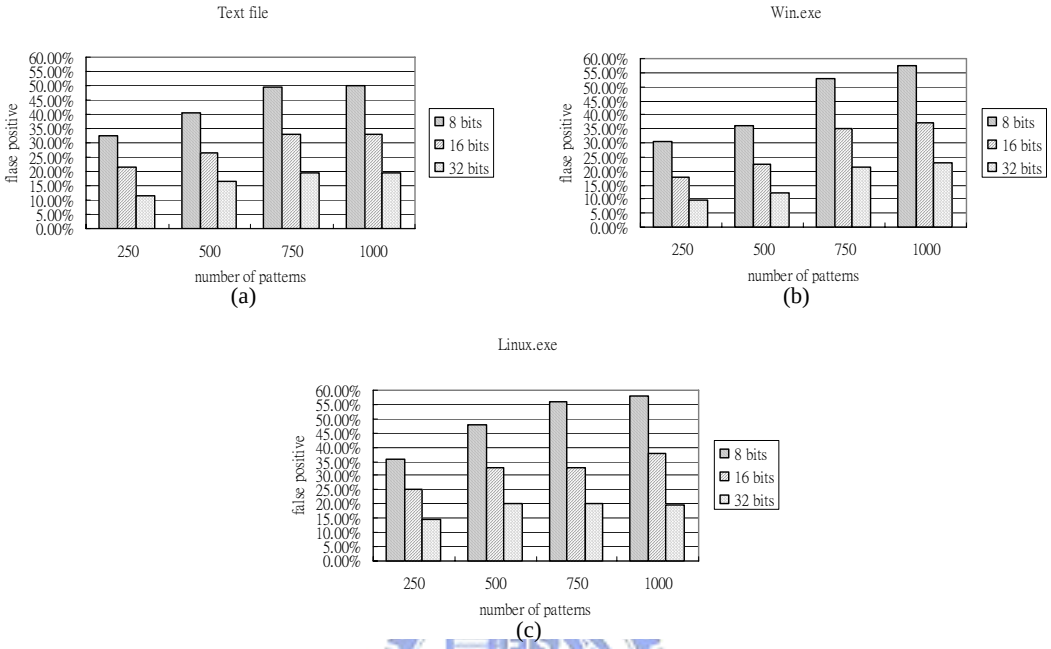


Fig. 15. The false positive rate of 16-bit and 32-bit bit vector for (a) text files, (b) Windows execution files, (c) Linux execution files.

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 next state table and failure state pointer are data structures we used. For each state, it takes 384 bits and 336 bits to store these data structures when the representation bit of state number is 32 and 16 bits, respectively.

5.2 Hardware Analysis

As mentioned before, our approach is flexible for both internal and external memory architecture. The 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, the internal memory architecture can be used for the high

performance with fewer patterns, such as IDS and firewall applications.

In our design, the root-indexing can match four bytes at the same time with address decoding technique which can minimize the memory usage and make it more space efficiency. Furthermore, two string matching engines can be used to take advantage of the hardware feature of dual-port SRAM.

The operating frequency of synthesis result for our internal SRAM architecture is 220 MHz which is reported by SynpilifyPro. 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 which means no byte has been matched is

$$16bits \times 220MHz \div 2(clock) \times 2(SMengine) = 3.52Gbps. \quad (1)$$

The throughput in the average case, depending on the average proportion of root-indexing matching and bitmap AC matching, as shown in Fig. 10(a), can be estimated as

$$(79.94\% \times 32 \div 2 + 20.06\% \times 19.03\% \times 8 \div 8 + 20.06\% \times 76\% \times 32 \div 2) \times 220MHz \times 2$$

$$\approx 3367Mbps = 3.367Gbps. \quad (2)$$

For worst case, all bytes are matched in the text buffer. The throughput is

$$(26.67\% \times 16 \div 2 + 73.33\% \times 1 \div 8) \times 220\text{MHz} \times 2 = 979.1155 \text{ Mbps} \approx 0.98 \text{ Gbps}. \quad (3)$$

It is obvious that the performance in the average case has very high performance which is very close to that in the best case and also has moderate performance in the worst case. This result demonstrates that our pre-hashing and root-indexing techniques are useful for high-performance content filtering applications.

5.3 Compared with Existing Works

Comparing with the pure bitmap AC in hardware design, 96% of bitmap AC matching can be avoided by our proposed two techniques. This can be estimated by the portion of root-indexing, false-positive and non-hit case in Fig. 13(a) and (b), shown as below

$$79.94\% + 20.06\% \times (4.97\% + 76\%) = 96.18\ \%. \quad (4)$$

Furthermore, the throughput of pure bitmap AC hardware in the identical hardware environment can be estimated as

$$8bits \div 8(clock) \times 220MHz \times 2(SMengine) = 440Mbps. \quad (5)$$

Thus, our throughput described in section 5.2 is almost 7.65 times faster than the original bitmap AC in the average case.

Because that our design is memory based architecture, it takes only 1688 LUTs which is far less than other works. Comparing with the memory-based architecture work [17], 384 bits memory usage for each state is much less than their 8192 bits which use 256 32-bit pointers. Also, the operating frequency 220 MHz will not decrease as the number and size of patterns grow. Although some existing works claim that their throughput can achieve up to 10 Gbps, but their designs are not feasible for real systems. Comparing with these works, we provide a flexible and scalable architecture for real applications with acceptable throughput.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we proposed an architecture which takes scalability, flexibility and performance into consideration. Furthermore, root-indexing and pre-hashing are two used acceleration techniques for dramatically improving the performance of our design. Also our data structures are compressed and can be stored either in the internal SRAM or external DRAM. The internal SRAM architecture provides average 3.367Gbps throughput with the size limitation of patterns. The external DRAM architecture provides the high scalability for the integration of multiple applications with acceptable throughput.

The proposed internal SRAM architecture is implemented on the Xilinx ML310 FPGA-based platform, and the driver interface API is provided for software/hardware integration. The string matching function of the target application ClamAV is also modified to setup the string matching engine. We tuned the hardware design according to the analysis results of our software simulation, and also built a complete system solution for content filtering applications such as IDS, URL blocking and ClamAV.

6.2 Future Work

Although the average throughput of our internal SRAM design can achieve 3.367 Gbps, our architecture is too complicated to design into a pipeline architecture which can get the better throughput. Therefore, now we adopt the multi-cycle implementation method which will degrade the throughput. For higher performance

of the internal SRAM architecture, the pipeline is a necessary trick to be applied. Thus, the proposed design should be refined to make it simple. Also, for the external memory-based architecture, the most defeat is the bus bandwidth and contention issue. It is the native limitation. Furthermore, the path compression of bitmap AC is not used in our design. However, this technique can use the memory effectively, and also can reduce the access frequency of memory in the external DRAM architecture. Thus, the path compression technique is worth to take into consideration in the future.

References

[1] S. Antonatos, K. Anagnostakis, and E. Markatos. Generating realistic workloads for network intrusion detection systems. In ACM Workshop on Software and Performance, Redwood Shores, CA, Jan. 2004.

[2] Aho and M. Corasick. Fast pattern matching: an aid to bibliographic search. In Commun. ACM, volume 18(6), pages 333-340, June 1975.

[3] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

[4] R. Boyer and J. Moore. A fast string searching algorithm. Communications of the ACM, vol.20, no10, pp762-772, October 1977.

[5] Broder and M. Mitzenmacher. Network applications of Bloom Filters: A survey. In Proc. Of Allerton Conference, 2002.

[6] Young H. Cho, Shiva Navab, and William Mangione-Smith. Specialized hardware for deep network packet filtering. In Proceedings of 12th International Conference on Field Programmable Logic and Applications, France, 2002.

[7] Young H. Cho and William H. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In IEEE Symposium on Filed-Programmable Custom Computing Machines, Napa, CA, USA, April 2004.

[8] Z. K. Baker and V. K. Prasanna. Time and area efficient reconfigurable pattern matching on FPGAs. In Proceedings of FPGA ’04, 2004.

[9] Z. K. Baker and V. K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In IEEE Symposium on Field-Programmable Custom Computation Machines, Napa, CA, USA, April 2004.

[10] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In Proceedings of 13th International Conference on Filed Programmable Logic and Applications, Lisbon, Portugal, September 2003.

[11] I. Sourdis and D.Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed nids pattern matching. In IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, USA, April 2004.

[12] Sarang Dharmapurikar, Praven Krishnamurthy, Todd Spoull, and John Lockwood. Deep packet inspection using bloom filters. In Hot Interconnects, Stanford, CA, August 2003.

[13] S. Dharmapurikar, M. Attig and J. Lockwood. Design and Implementation of a

String Matching System for Network Intrusion Detection using FPGA-based Bloom filters. In the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04), April 2004.

[14] N. Tuck, T. Sherwood, B. Calder and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In Proceedings of the IEEE Infocom Conference, Hong Kong, China, 2004.

[15] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In IEEE Symposium on Field-Programmable Custom Computing Machines, Rohnert Park, CA, USA, April 2001.

[16] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, Sept. 2002.

[17] M. Aldwairi, T. Conte, P. Franzon. Configurable string matching hardware for speeding up Intrusion Detection. In ACM Sigarch Computer Architecture News. Vol. 33, No. 1, March 2005.

[18] Kuo-Kun Tseng, Ying-Dar Lin, Tsern-Huei Lee, Yuan-Cheng Lai, A Parallel Automaton String Matching with Pre-Hashing and Root-Indexing Techniques for Content Filtering Coprocessor, 16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors, Samos, Greece, July 2005.

[19] SNORT official web site. http://www.snort.org

[20] ClamAV official web site. http://www.clamav.net