國 立 交 通 大 學

網路工程研究所

碩 士 論 文

對深層封包檢測進行字串比對之軟硬體協同設計

Hardware Software Co-Design for Deep Packet Inspection with String Matching

研 究 生：劉岱穎

指導教授：林盈達　教授

中 華 民 國 九 十 七 年 六 月

對深層封包檢測進行字串比對之軟硬體協同設計

Hardware Software Co-Design for Deep Packet Inspection with String Matching

研　究　生：劉岱穎　　　　Student: Tai-Ying Liu

指導教授：林盈達　　　　Advisor: Dr. Ying-Dar Lin

國立交通大學

網路工程研究所

碩士論文

A Thesis

Submitted to Institutes of Network and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science

June 2008

HsinChu, Taiwan, Republic of China

中華民國九十七年六月

對深層封包檢測進行字串比對之軟硬體協同設計

學生：劉岱穎 **指導教授：林盈達**

國立交通大學網路工程研究所

摘要

在病毒掃描與深層封包檢測中，字串比對為系統瓶頸，而目前為了加速字串比對的速度大多以硬體來加速整體輸出量，但是目前大多數的研究都著重在字串比對硬體的加速而忽略了硬體與軟體整合時是否仍然保持高輸出的特性。本論文整合了軟硬體並探討出其中為何系統整合後無法發揮高輸出的特性，並發現基於記憶體的字串比對硬體會因為記憶體搬移時間過長導致無法發揮出硬體高輸出的特性。實驗實作了字串比對 BFAST*硬體，此硬體修改自 BFAST 硬體並使它更容易讓軟體控制，而且與開放原始碼的掃毒套件 ClamAV 整合後實際在 FPGA 上計算出輸出量，整合後的掃毒軟體 ClamAV 輸出量為 146.612Mbps 比整合前純軟體的 ClamAV 增加了約 27.3 倍的速度。而資料從 ClamAV 搬移至 BFAST*的搬移時間佔了字串比對約 90%的比重，使得資料搬移時間成為字串比對的瓶頸。如果資料已經在記憶體中，直接使用 DMA 去記憶體搬移資料進 TextRam 可以增加輸出量至 912.7Mbps，此時的 DMA 輸出量為 1.3Gbps，如果 DMA 速度能提昇至原來的 6 倍，ClamAV 的輸出量甚至可達到 2.176Gbps。

關鍵字： 字串比對，軟硬體協同設計，深層封包檢測

Hardware Software Co-Design for Deep Packet Inspection with String Matching

Student: Tai-Ying Liu **Advisor: Dr. Ying-Dar Lin**

Department of Computer Science

National Chiao Tung University

Abstract

String matching is the bottleneck of anti-virus and deep packet inspection. It is a promising trend to offload the inspection to a specific hardware engine for high-speed applications that demand the throughput up to multi-giga bit rate. Most papers emphasized that raising the throughput of hardware engine but not whole system. This work integrates string matching hardware with ClamAV, and discusses why it can’t stay high throughput when integrating hardware with software. This work also discovers that the data moving time is the bottleneck.

The experiment is implementation of string matching hardware BFAST* which make it suitable for software control. We integrate the open source anti-virus package ClamAV with BFAST* and implement on Xilinx FPGA. The experiment includes benchmark the throughput of ClamAV integrated with BFAST*. The throughput of ClamAV with BFAST* is 146.612Mbps and is about 27.3 factors of pure software ClamAV. Time of transferring data from ClamAV to BFAST* occupies about 90% of string matching. If data is well prepared in RAM, we transfer data from RAM to TextRam using DMA, and the throughput of ClamAV is 912.7Mbps. The throughput of DMA is 1.3Gbps. If six times of throughput of DMA is possible, then the throughput of ClamAV is 2.176Gbps.

Keywords: string matching, hardware/software co-design, deep packet inspection

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Deep packet inspection is essential to network content security applications such as intrusion detection systems and anti-virus systems. The inspection searches the packet payload for a database of signatures with a string matching algorithm. The efficiency of string matching is therefore critical to the system performance. The time complexity of a string matching algorithm can be linear or sub-linear time. Linear time algorithms such as those that track the text with a *Deterministic Finite Automata* (*DFA*) must go through every character in the text, so the complexity is $O(n)$, where n is the text length. Sub-linear time algorithms such as the Boyer-Moore algorithm [1] can skip characters not in a match and search the text in sub-linear time on average.

It is a promising trend to offload the inspection to a specific hardware engine for high-speed applications that demand the throughput up to multi-giga bit rate [2-8]. The system requires elegant integration between software and hardware implementation for efficient offloading. Most existing research and implementation of deep content inspection emphasizes only high throughput [4-8] of the scanning engine, but the overall throughput of the system may still be much slower than the raw throughput of the scanning engine due to Amdahl's law. For example, the hardware accelerator is getting high performance of scanning but the text loading time is too slow to scan. The whole performance is bottlenecked by text loading.

This work features a software/hardware co-design from an anti-virus package ClamAV (www.clamav.net) on Linux to offload virus scanning to a hardware scanning engine, which features sub-linear execution time on average by skipping characters not in a match and thus inspecting multiple characters at a time in effect. The first thing of designing an offloading scheme is the partition of hardware and software to evaluate the cost and performance and choose which part should be implemented in hardware or software. After the partition, we design the interface between hardware and software. Usually, the interface in Linux is module driver; it contains the memory address of the hardware engine and the behavior of performing the hardware. Driver must to be designed flexible and simple, so the application can simply operate the hardware.

The hardware engine can quickly filter the “no-match” case, since the majority of network traffic and files in system does not contain viruses. We analyze flow and finding the bottleneck while integrating with ClamAV.

The rest of this work is organized as follows. Chapter 2 reviews string matching algorithms and the Bloom Filter Accelerated Sub-linear Time (BFAST) algorithm. Chapter 3 introduces the offloading scheme, including the partition and communication between hardware and software. Chapter 4 discusses the detailed implementation issues, and presents the system architecture as well as each component. Chapter 5 evaluates the proposed architecture and benchmarks each offloading step. Chapter 6 concludes this work.

Chapter 2 Related works

2.1 String Matching Algorithms

Deep packet inspection involves scanning the content for a set of patterns $P = \{P^1, P^2, \cdots P^r\}$ where *r* is the number of patterns. We also assume *lmin* to be the minimum pattern length and *lmax* to be the maximum. The time complexity of string matching algorithms can be linear and sub-linear. A linear-time algorithm characterizes stable throughput and easy implementation, while a sub-linear time algorithm reaches high throughput but may be slowed down in the worst case.

The Aho-Corasick (AC) algorithm [10] is a typical example of a linear-time algorithm for string matching. It builds a finite automaton that accepts all the patterns in the pattern set in preprocessing, and then tracks the text for a match during scanning. Numerous studies have improved the algorithm recently due to its popularity. Some of them can track multiple characters at a time with multiple hardware engines, while others focus on compressing the automaton. However, the data structure for a large pattern set is still large. Either several hardware engines or a high operating frequency are needed for high performance.

The Wu-Manber (WM) algorithm [13] is a typical algorithm for multiple-string matching in sub-linear time. It searches for the patterns by moving a search window of *lmin* characters along the text. If the rightmost block in the window does not appear in any of the patterns, the window can be shifted by *lmin-B+1* characters without missing any pattern, where *B* is the block size; otherwise, the sift distance is *lmin -j*, where the rightmost occurrence of the block in the patterns ends at position *j*.

2.2 Bloom Filter Accelerated Sub-linear Time (BFAST) algorithm

We presented the Bloom filter Accelerated Sub-linear Time (BFAST) architecture [9] that can search the text for the patterns in sub-linear time using Bloom Filters. The architecture can avoid the large shift table in the WM algorithm [14] and combine the AC algorithm for verification to reduce the impact from the worst-case performance. Fig. 1 illustrates how to derive the shift distance from the heuristic using Bloom filters. Assume $\{P_1, P_2, P_3\}$ is the pattern set, and the blocks in the patterns are divided by position. The Group G_0 is {efgh,mnop,vuts}, G_1 is {defg,lmno,wvut}, and so on. Each group is stored in its respective Bloom filters, e.g., G_0 in BF(G_0). We set the block size to be 4, and window size to be 8. The rightmost block in the search window, “cdef”, hits G_2, so the shift distance is 2 without missing any pattern. If there is no hit reported, the shift distance could be 8.

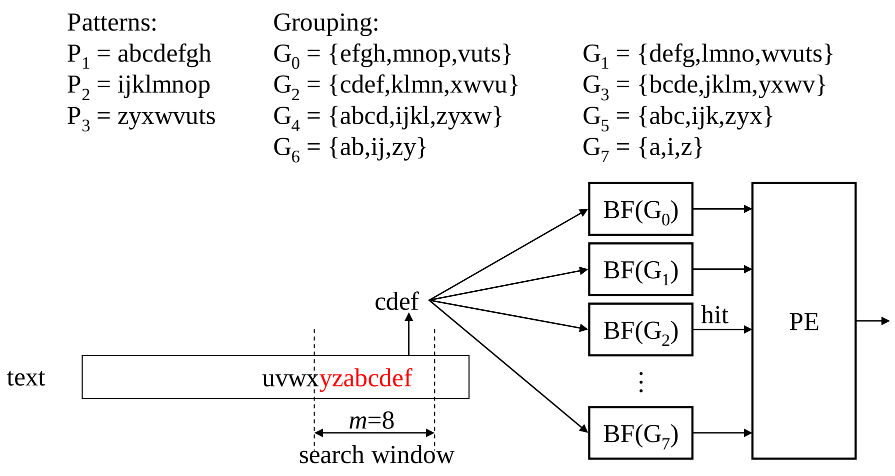

Fig. 1 Implicit shift table using Bloom filters

2.2.1 String matching architecture

Fig. 2 shows the string matching architecture. This architecture includes two main components: (1) the scanning module that fetches text located by *TextCounterGenerator* from *TextMemoryFetch*, feeds it into Bloom filters, and then shifts the text according to this query result, and (2) the verification module and interface that pass the location of a possible match to *VerificationJobBuffer*, and verify the text on the location.

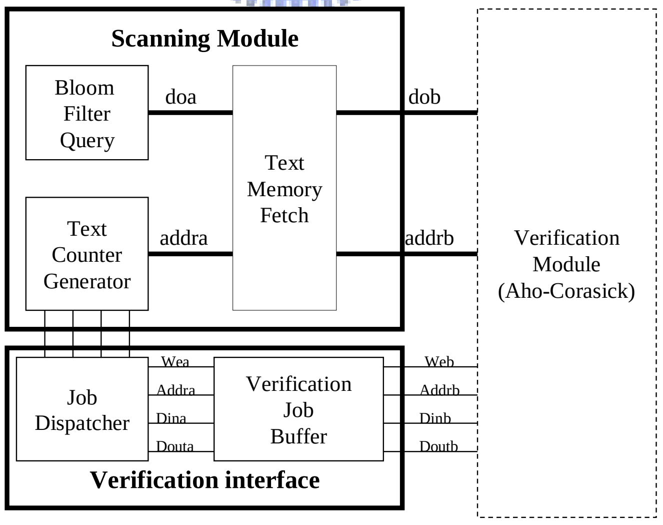

Fig. 2 Overview of the string matching architecture

2.2.2 Text memory fetching

,The BFAST architecture cuts the text into four interleaving banks for four-byte blocks to access four continuous memory bytes in parallel. The block is rotated according to the byte offset after a block is fetched.

2.2.3 Bloom filter querying

Eight independent Bloom filters stores a group between G_0 and G_7. The block fetched by *TextMemoryFetch* module queries the Bloom filters in parallel to get membership information. If more than one group is hit, the shift distance is the number of the smallest number of group in the hitting groups; otherwise, the shift distance is the pattern length.

2.2.4 Text position controller

If the shift distance is longer than 0, the scanning module will shift the search window; otherwise it means this block matches some pattern and scanning module will start to verify the whole search window. The scanning module verifies the whole search window by querying the Bloom filter for every block in the search window.

2.2.5 Verification interface

Two components are in verification interface: *JobDispatcher* and *VerificationJobBuffer*. If the query result gets corresponding shift distance, *JobDispatcher* passes the location of search window to *VerificationJobBuffer* and then notifies Verification Module start to verify it is a virus or not.

Chapter 3 Designing the offloading scheme

3.1 Hardware/Software partition

The first and most important step is hardware and software partition. According to Amdahl's law—"make the common case fast", the most critical part should be implemented in hardware. Because string matching is critical in virus-scanning and IDS, it is made into a hardware module to accelerate the scanning. We intend to accelerate the open source software ClamAV to increase the throughput. Fig. 3 shows the hardware and software partition in ClamAV.

The software part of ClamAV works with BFAST* to quickly filter files without viruses. ClamAV performs exact string matching and matcher-bfast* calls driver to enable DMA for transferring data and enable BFAST* to scan the file. The file is infected if there is no possible virus, and if BFAST* finds a possible virus, ClamAV pass the file to matcher-bm for verification.

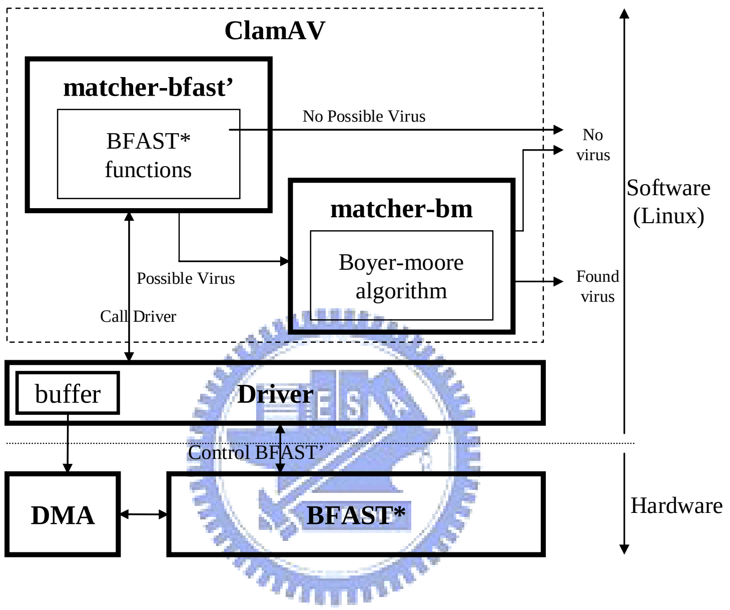

Fig. 3 Partitioned string matching: hardware scanning and software verification.

3.2 Hardware interface design

BFAST* hardware has five modules: *TextPoint, TextRam, HashGenerator, BloomFilterQuery* and *TPController*. (1) TextPoint stores the address in TextRam where we want to scan. (2) TextRam stores the text to be scanned. (3) HashGenerator generates the hash values of the block. (4) BloomFilterQuery is reported the shift distance, according to the hash value generates from TextHashGenerator. (5) TPController controls these modules, shifts the window and stores the status into registers.

We design a five-stage pipeline for BFAST*, so TPController can shift the window per cycle. Fig. 4 shows the five stages:

(1)*TextPosition* (*TP*): it gets an address from TextPoint.

(2)*TextRead*: TPController gets the text from TextRam.

(3)*Hash*: TPController gets the hash value of HashGenerator.

(4)*ShiftDistance*: TPController gets the shift value from BloomFilterQuery.

(5)*WB*: TPController computes the shit distance and update the address of TextPosition.

The stages of TextRead and ShiftDistance are memory access, so the memory access time restricts the clock rate.

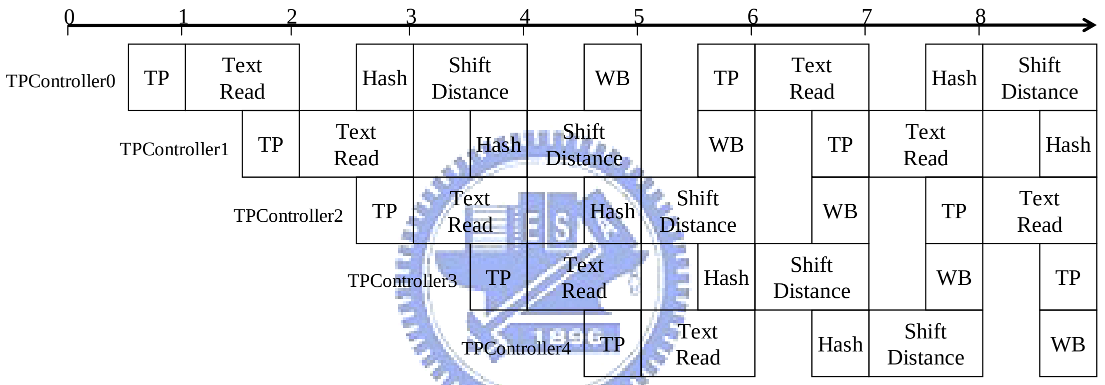

Fig. 4 Five stage pipeline.

Five TPControllers are instantiated because there are five pipeline stages. Each TPController scans 1600 bytes of data in TextRam. We make each TextPoint in TPController points to the address of TextRam at 0, 1600, 3200, 4800, 6400, and TPController0 scans data in TextRam addressed from 0 to 1599 and TPController1 scans the data in TextRam from addressed 1600 to 3200 and so on.

Fig. 5 shows the state machine of TPController. There are four states: INIT, SCAN, CHECK and HOLD.

1. In the beginning, TPController stay in the state INIT until BFAST* is enabled, and then current state transits to SCAN state.
2. In SCAN state, TPController gets the shift distance, if the shift distance is zero, it

means that the text should be additional checking, and then current state transits to CHECK state. Otherwise, if the shift distance is not zero then TPController update TextPoint by adding shift distance and stay in SCAN state until all data in TextRam is scanned finished.

3. While state transits to CHECK state, TPController check that every blocks in window is hit. If every blocks is hit with corresponding position then current state transits to HOLD and reports that there is a possible match
4. The HOLD state keeps the TPController status.

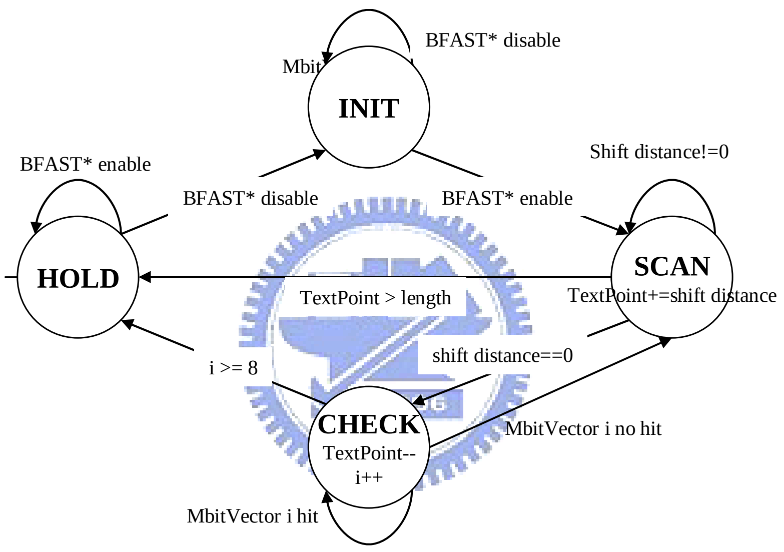

Fig. 5 The state transition diagram of TPController

There are two TextRams. When one TextRam is being scanned, BFAST* stores data into another TextRam. The two TextRam act alternately.

3.3 Software interface design

Two mechanisms for ClamAV gets information from BFAST*: interrupt and polling. If BFAST* finds a possible virus for ClamAV to verify, BFAST* interrupts the CPU and reports ClamAV that a virus needs to be verified, this kind mechanism of getting information from BFAST* calls interrupt. The benefit of interrupting is CPU may do other tasks, while BFAST* quickly filters no-match cases until BFAST* finds a possible virus or the scan is finished. If an interrupt occurs, the CPU should switch to ClamAV, and context switch adds the overheads of scanning. If many possible viruses appear, the context switch overhead is huge.

On the other hand, the mechanism that ClamAV keeps detecting the status of BFAST* is called polling. Because polling lets ClamAV know a possible virus immediately and the CPU does not need to do context switch, it performs better than interrupting. Polling increases the throughput of virus scanning but decreases the performance of whole system unless CPU does not need to do any other tasks. We adapt the polling mechanism for higher throughput.

ClamAV scans a file using Boyer-Moore algorithm and Aho-Croasick algorithm, Boyer-Moore algorithm is implemented as matcher-bm library and it performs exact string match. Aho-Croasick algorithm is implemented as matcher-ac library, and it performs regular expression string match.

In our design, BFAST* performs exact string matching and filers the no-match cases first. If a possible virus is detected by BFAST* then matcher-bm is invoked for verification. On the other hand, if there is no possible virus is detected, it means the file is not infected. Most of network traffic and files in system belongs to the no-match case, so it scans quickly.

3.4 Hardware/Software interface design

The driver functions include memory writing, scanning module behavior and getting status. The data structure in the memory of BFAST* is memory mapped, so we just write data to corresponding memory address. Several data structures should be written: (1) the hash functions (2) blocks in the patterns and (3) the text to be scanned. After the data are well prepared, TPController in BFAST* will start to scan the text and write the status to registers.

Direct Memory Access (DMA), it transfer data from physical memory address to

TextRam in BFAST*. The problem is that data from user space can’t declare a continued physical memory space, so driver create a continued physical memory space and copy the data from ClamAV into this continued physical memory. This continued physical memory space can’t be cached, because if DMA transfer data and the memory didn’t update yet, then DMA transfers the old data into TextRam.

Chaper 4 Implementation Details

We modify the ClamAV to co-work with BFAST smoothly. ClamAV scans the files for the patterns to recognize infected ones. ClamAV implements two libraries for string matching: matcher-bm for Boyer-Moore algorithm and matcher-ac for Aho-Croasick algorithm. It reads a file in batch into a buffer of 131,072 bytes. The library in matcher-bfast* stores the buffer into driver and enables DMA to transfer the buffer into TextRam in BFAST*. The TPController then scans the data in TextRam. If no virus is possible in the data, BFAST* loads the next batch of text into TextRam; otherwise, TPController reports the possible virus, and matcher-bm verifies the buffer contains a virus or not.

Fig 6 shows the entire system architecture. We attach BFAST* on PLB Bus. The OPB Bus and PLB Bus operates at 100MHz, but the maximum data width of PLB is 64bits, twice the width of OPB. DMA in BFAST* transfers the data from DDR SDRAM to BFAST*. PPC405 is 300Mhz

To simplify observation and analysis the bottleneck, we discard the regular expression case. If regular expression is necessary in whole system, the pattern which is regular expression form should be cut off. Take a pattern “abcdefg{-10}hijkl” for example, the pattern is divided into two parts, “abcdefg” and “hijkl”, then add these two parts as two patterns into BFAST*. BFAST* filters the no-match case and library matcher-ac verifies the match case.

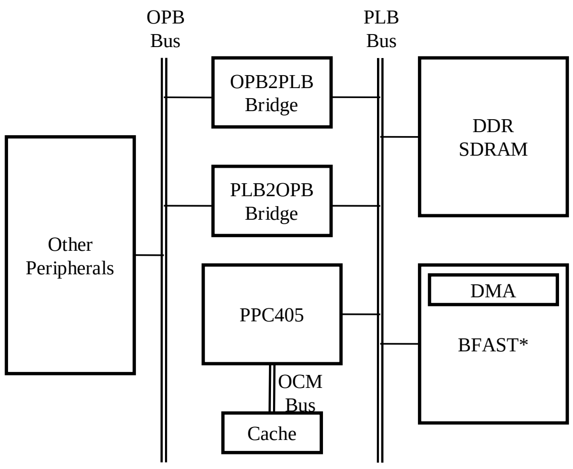

Fig. 6 System architecture

4.1 Overview of the co-design platform

The system is implemented on MontaVista Linux 2.4.20-8 and ClamAV 0.88 on Xilinx ML310 VirtexII Pro Board. We illustrate the scan flow and explain the function of each module. Before starting scanning the files, ClamAV programs the signatures from ClamAV database files into the BloomFilterQuery module of BFAST* and matcher-bm. After the data structure is well prepared and loaded, ClamAV reads the file and the matcher-bfast* library invokes the driver in BFAST* to scan the content. If BFAST* does not find any virus, it reports the file is not infected; otherwise, ClamAV will pass this file to matcher-bm for verification.

4.2 Hardware Implementation

Figure 7. shows the five modules in BFAST*.

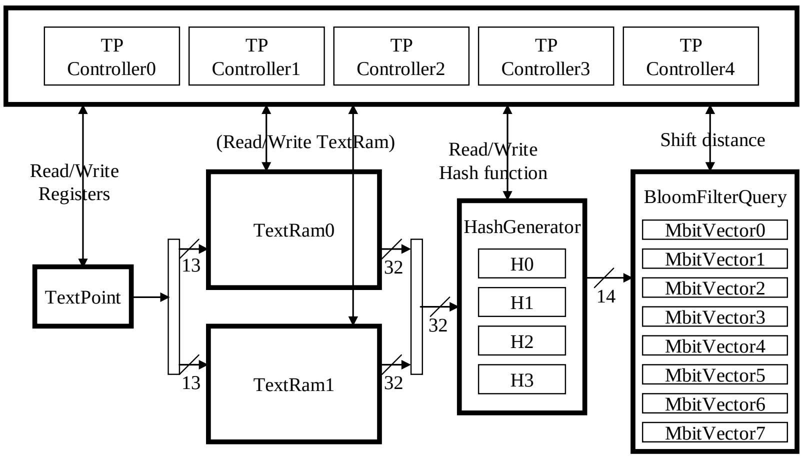

Fig. 7 Components in the BFAST* architecture.

1. TextRam0 and TextRam1 modules

Each TextRam has 8KB, indexed by 13 bits address line and 8 bits data line. Four bytes can be fetched from TextRAM each time for the heuristics. ClamAV scans the file content in TextRam.

2. HashGenerator module

HashGenerator module includes four Hash function: H0, H1, H2, and H3. The data in TextRam is fetched, then the data will be hashed by these four hash function and gets 4 14bits hash values.

3. BloomFilterQuery module

BloomFilterQuery module includes eight MbitVectors. Each MbitVector is 4KB with 14bits address line and 1bit data line. MbitVector stores the Bloom filter data. MbitVector0 stores the BF(G0) and MbitVector stores BF(G1) and so on. When MbitVector n is hit, then TPController shift n. For example, if MbitVector3 is hit, so we shift window 3 bytes.

4. TPController module

Five TPController is instantiated corresponding five pipeline stages. Each TPController scans 1,600bytes of data in TextRam. For example, TPController0 scan the data address from 0 to 1,599 and TPController1 scans the data from address 1,600 to 3,199.

5. Registers

There are two register for enable the BFAST* and 1 register stores the status of BFAST*. Fig. 8 shows the format of these register. Register *EnableTextRam0* enables TPController scans TextRam0 with start address and the size of length and *EnableTextRam1* enables TPController scans TextRam1 with start address and the size of length. Register2 stores the status of BFAST*.

The information stores in register *StatusRegister* is describe here. The *BFAST*enable* single means that TextRam0 or TextRam1 is scanning, *TextRam0finished* and *TextRam1finished* means which TextRam scans completely, *TextRam0scanning* and *TextRam1scanning* means which TextRam is scanning and *TextRam0Error* and *TextRam1Error* means which TextRam is error while reading or writing. *VirusAddress* reports which TPController finds a possible virus. *TextPointer* stores the value that point to the address of TextRam. *FoundVirus* reports it finds a possible virus in TextRam.

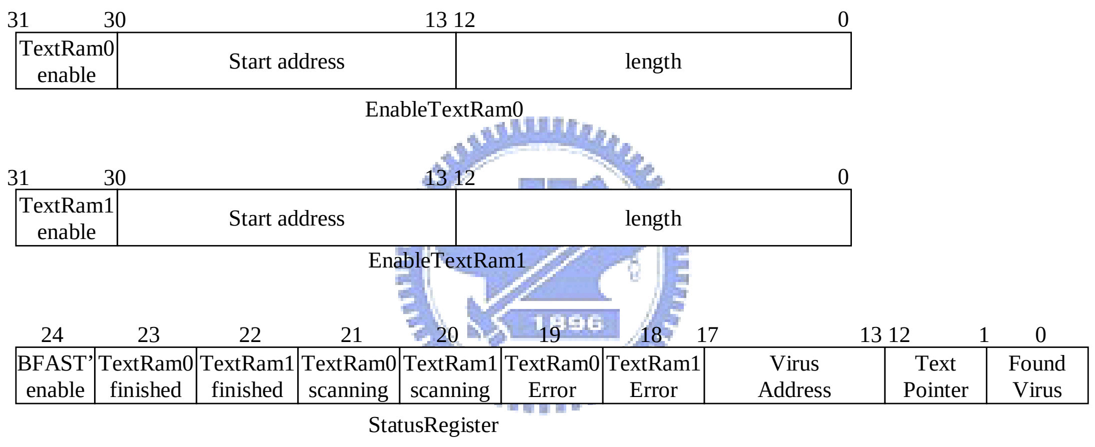

Fig. 8 The format of registers.

4.3 Driver Implementation

We implement the driver on Linux for BFAST* controlling. ClamAV invokes matcher-bfast*, and then matcher- matcher-bfast * calls driver to get the status of BFAST* by accessing StatusRegister register, or enable BFAST* to scan data in TextRam. Driver also provides the function of accesses the data in TextRam0, TextRam1, HashGenerator and MbitVectors in BloomFilterQuery modules.

Driver has several functions to perform behavior of BFAST*. (1) *TEXTRAM* loads the data from ClamAV and copy these data into DMA buffer which is a physical continue memory space and then enables DMA to transfer data from DMA buffer to TextRam in BFAST*. (2) *REG_SELECT* selects which register to operate, there are 3 register in BFAST*. (3) *REG_IOC_IN* reads data from register where REG_SELECT points to and (4) *REG_IOC_OUT* writes data into registers. (5) *AR_SELECT* selects which memory to operate, the memory is TextRam0, TextRam1, H0 to H3, and MbitVector0 to MbitVector7 in BFAST*. (6) *AR_LSEEK* selects the address of memory. (7) *AR_IOC_IN* reads data from memory, and the final function (8) *AR_IOC_OUT* writes data into memory. For example, if we want to enable BFAST* start to scan TextRam0 and the length is 0x1000, so we needs to use the function ioctl and pass it the parameter REG_SELECT and data of 0, then driver will transform to operate the EnableTextRam0 and then pass the parameter REG_IOC_OUT and the data of 0x80001000, then it start to scan TextRam0 with length 0x1000.

4.4 Software Implementation

We implement ClamAV 0.88 on MontaVista Linux 2.4.20-8. In order to enable the hardware for scanning files, we modify the ClamAV library. We modify two modules readdb, matcher and add a new library matcher-bfast*. The library readdb reads patterns (signatures) from ClamAV database files, so we need to modify it and transform the signature into BFAST* data structure and store into MbitVectors. We modify the library matcher for BFAST* scanning virus. We add four function in matcher-bfast* to operate BFAST*. The First function is bfast*_init, this function opens the device and return the file descriptor of BFAST*. Second function is bfast*_addpatt, this function reads signatures which library readdb reads from ClamAV database files and stores the pattern into MbitVectors. The third function is bfast*_scanbuff, this function scans the buffer where the library matcher reads from files. BFAST* has two TextRam with capacity of 8KB, so we can scanning the TextRam and store the buffer into another TextRam at the same time, Fig. 9 shows the

Pseudocode of loading text into MbitVectors while BFAST* scanning virus.

These two TextRam interleaving acts, so the buffer have to keep 10 bytes to prevent that the virus from cutting off between the two TextRam.

```
Disable TextRam0;
Disable TextRam1;
Current TextRam = TextRam0;
do{
    if(buffer size > (TextRamMaxSize+scanned))
        Copy the TextRamMaxSize of data from buffer into text;
    else
        Copy the size of (buffer size –scanned) of data from buffer into text;
    scanned+=the size of text;
    load buffer into Current TextRam;
    Wait for another TextRam until it scans completely or finds a possible match;
    If(Another TextRam finds a possible match) {
        Disable another TextRam;
        Return Found a virus;
    }
    else{
        if(buffer size > scanned) {
            scanned-=10;
            Disable another TextRam;
            Current TextRam = another TextRam;
        }
    Enable Current TextRam;
    }
} while(buffer size > scanned)
Wait for current TextRam until scans completely or finds a possible match;
If(Currrent TextRam finds a possible match) {
    Disable Current TextRam;
    Return Found a virus;
}
Else {
    Disable Current TextRam;
    Return No virus;
}
```

Fig. 9 Pseudocode of loading text while BFAST* scanning virus

Finally, the library matcher reads every 131072 bytes from files into buffer and then calls bfast*_scanbuff and matcher-bm to scan the buffer, we also have to keep the last 10 bytes to prevent the virus from cutting off.

If bfast*_scanbuff found a virus, bfast*_scanbuff report a the address of virus, and matcher-bm start to scan the data after the address.

Chapter 5 Experiment results

We benchmark the performance in various environments: ClamAV on ML310 board, BFAST* hardware simulation, BFAST* hardware implementation on ML310 board and ClamAV with BFAST*

BFAST* is attached on Process Local Bus (PLB), whose data width is twice that of On-Chip Peripheral Bus(OPB).

5.1 Pure Software

We implement ClamAV 0.88 with MontaVista Linux 2.4.20-8 on Xilinx ML310 Vertex II Pro Board. In this experiment, the number of signatures is 42108, We scan the data of 1KB and 1MB with and without viruses. ClamAV scans files with matcher-bm implements Boyer-Moore.

The matcher-bm reports whether a file is infected or not. If matcher-bm reports that this file is not inflected then ClamAV passes the file to matcher-ac for scanning multi-part signatures. Because we just program exact string, so only matcher-bm is concerned. Table. 1 shows the time of ClamAV scans files of 1KB and 1MB. If matcher-bm found a virus, it reported the file is infected and stops to scan the file. So ClamAV scans a 1KB file with virus which virus at the position 359, it needs 539 microseconds, because ClamAV found a virus and then it stops to scan data after that. The throughput of matcher-bm is (8*1,024*1,024)/1,559,306 or 5.37Mbps.

Table. 1 Scan time of ClamAV on ML310 (microseconds)

	1KB (virus at the position of 359 bytes)	1MB (virus at the position of 738,663 bytes)
With virus	matcher-bm: 539	matcher-bm: 1,102,765
Without virus	matcher-bm: 1,545	matcher-bm: 1,559,306

5.2 Hardware simulation and synthesis

We Implement the BFAST* in Verilog and synthesis it. BFAST* consumes 2,749 Slices out of 13,696 total Slices and 24 BRAMS out of 136 total BRAMS in Xilinx board and the maximum clock rate of system operates at 142.507 MHz. We know that the average shift distance is 7.71 from [9], so the average throughput is 142.507*7.71*8 or 8.79Gbps and the maximum throughput is 142.507*8*8 or 9.12Gbps when TextRam is full. When data size in TextRam is less than 1600 bytes, the throughput is fifth of the throughput of data is full with TextRam. And when data size in TextRam is larger than 1600 bytes but less than 3200 bytes, the throughput is two fifth of the 8.79Gbps and so on.

5.3 SoC without OS

We implement the BFAST* on Xilinx ML310 Vertex II Pro Board with firmware and benchmark the throughput. Our design is attached on the PLB. Xilinx ML310 only support 4 different clock rate: 25Mhz, 33Mhz, 50Mhz, 100Mhz. Our design can operate at 142Mhz, so we choose 100Mhz as PLB clock rate. Fig. 10 shows the scan time for different sizes of data in TextRam. When data size is less than 1,600 bytes, TPController 0 is enabled and each shift needs 5 cycles. On other hand, if the size of data is more than 1,600 bytes, the scan time is about 1000 clock time, because that when TPController0 is scanning the window at address 0, TPController1 is scanning the window at address 1,600, TPController2 is scanning the window at address 3,200 and so on at the same time. The maximum throughput is 8000 bytes/ (1004*10ns) or 6.37Gbps when data size is 8000 bytes which means the TextRam is full.

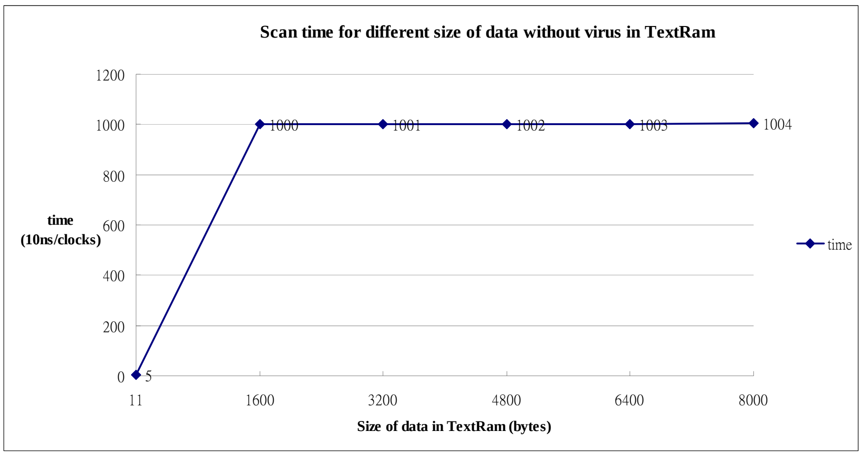

Fig. 10 Scan time for different size of data without virus in TextRam.

We wrote a virus at different of address of TextRam and evaluate the throughput. Fig. 11 shows the scan time for different address with virus in TextRam. When the last block of window is match, the TPController will enter the verification state, and verify all block in window then entering hold state. It costs 55 cycles including 5cycles for checking the last block is matching or not, 40 cycles for 8 block verification each costs 5 cycles, 5 cycles for check these block is all at appropriate position and 5 cycles for report it is matched or not.

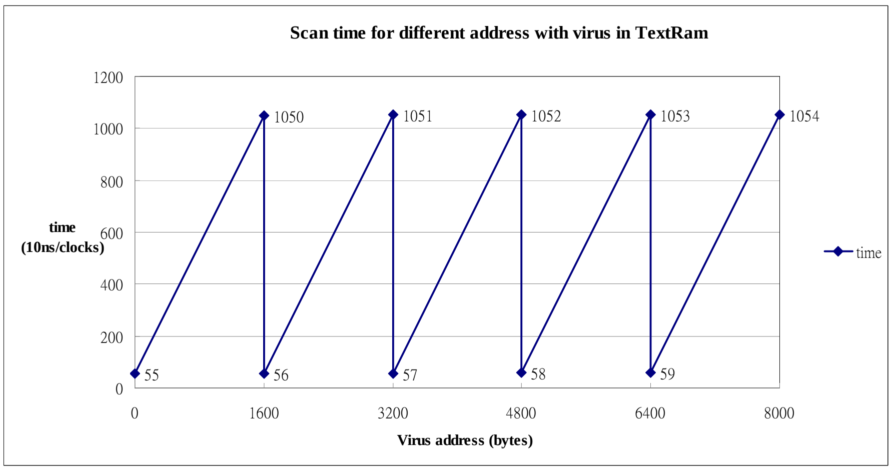

Fig. 11 Scan time for different address with virus in TextRam.

5.4 SoC with Linux

We have 3 configurations according to where the data to be scanned is:

1. **Data to be scanned is located in hard disk.**

Table. 2 shows the throughput of ClamAV with BFAST* scans files which are 1KB and 1MB with virus and without virus. When ClamAV scans a file without virus, only matcher-bfast* is invoked. And if this file is infected, matcher-bm is launched for verification. The total throughput of ClamAV is about 146.612Mbps. The time of writing data into TextRam occupies about 90% of string matching time.

Table. 2 Scan Time of ClamAV integrated with BFAST* on ML310 (microseconds)

	1KB (virus at the position of 359 bytes)	1MB (virus at the position of 738663 bytes)
With virus	matcher-bfast*: 270 (write data into TextRam: 94)	matcher- bfast*: 40,810 (write data into TextRam: 36,656)
	matcher-bm: 17	matcher-bm: 26
Without virus	matcher- bfast*: 273 (write data into TextRam: 95)	matcher- bfast*: 55,552 (write data into TextRam: 49,891)

The throughput of the integrated design is 146.612Mbps. The significant disparity between the pure hardware throughput and the co-design throughput is due to the following reasons. Although scanning on the FPGA is fast, ClamAV must pass the text to be scanned to the device driver (from ther user space to the kernel space), and the driver in turn copies the text to the DMA buffer. The DMA then transfers the text in the buffer into TextRam. These steps of data passing occupies nearly 90% time of total processing, and slows down the pure hardware throughput. Fig.12 shows the time distribution in these steps. The other 10% overhead of in the text processing is monitoring the status of BFAST* and keeping 10 bytes to prevent missing the virus that spans two contiguous batches of text.

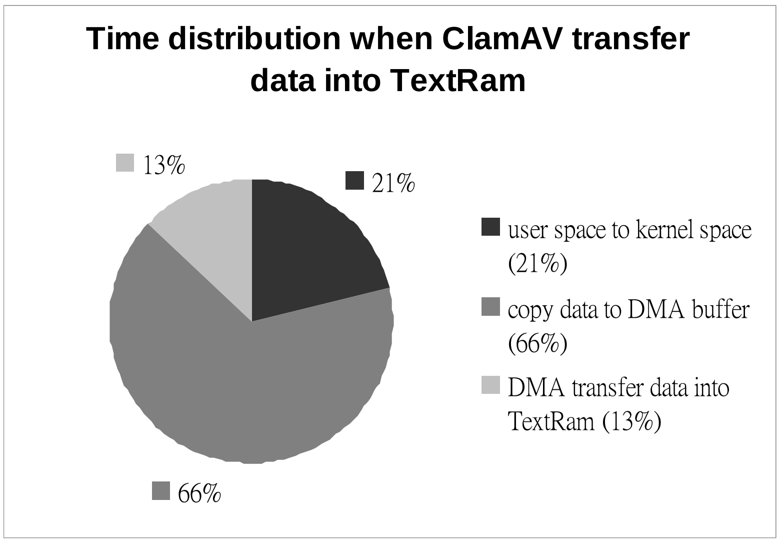

Fig. 12 Time distribution when ClamAV transfers data into TextRam

2. Data to be scanned is located in memory:

According to the above observation, pursuing a fast hardware design of a scanning engine is insufficient.

Two steps are fundamental and inevitable in the design: (1) having the text to be in the memory somehow and (2) transferring the text from memory into the FPGA with DMA. Assuming the incoming text could be somehow directly stored in the memory without the aforementioned overheads, the other possible bottlenecks in (1) become the speed of the network interface or the disk I/O, depending on where the text is from: packet content or disk files. Step (2) is not fast enough in the current platform. It's only 1.3 Gbps, and total system throughput is 912.7Mbps, so the throughput is restricted by the DMA. If the DMA throughput could be improved, the design could be much faster.

3. Data to be scanned is located in memory and DMA throughput is improved:

Fig.13 shows that throughput of ClamAV can reach up to 2.176 Gbps, while the throughput of DMA is 7.8 Gbps and data is well prepared in the memory.

When ClamAV scans 1MB of data in Ram, it takes 2789 microseconds for monitor the status of BFAST* and 6402 microseconds for loading data into TextRam. So the throughput of ClamAV is 8*1024*1024/ (2789+6402) or 912.7Mbps, and if the throughput of DMA is 2.6Gbps, the throughput of ClamAV is 8*1024*1024/(2789+(6402/2)) or 1.4Gbps, and so on.

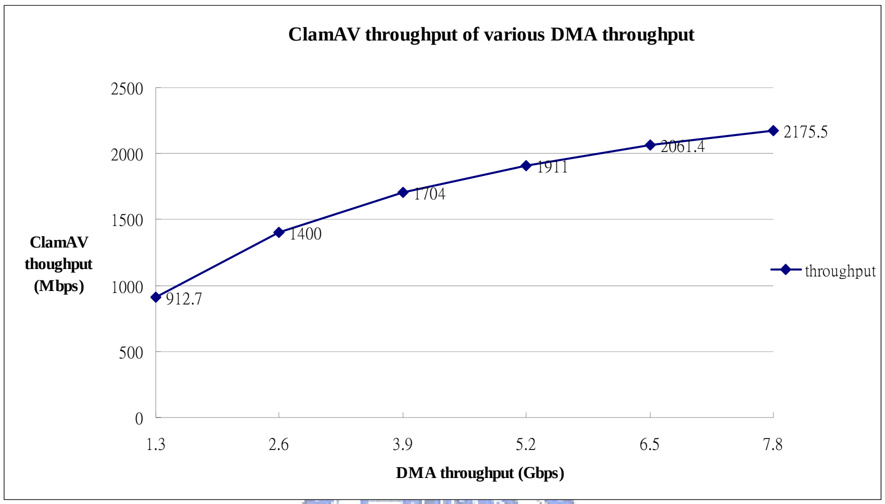

Fig. 13 ClamAV throughput of various DMA throughput.

Chapter 6 Conclusions and Future Works

This work implements a hardware BFAST* with ClamAV in MontaVista Linux on ML310 board. It increases the throughput of ClamAV. The throughput of the design scans a file is about 146.612 Mbps. We improve about 27.3 factors of original ClamAV when scanning the large of files.

In hardware simulation, the hardware clock rate reaches 142.507MHz after synthesis. The average shift distance is 7.71 bytes [9], so the average throughput is 7.71*8*142.507 = 8.79 Gbps. The best case is that the window shift 8 bytes every cycle, so the best case throughput is 8*8*142.507 = 9.1Gbps.

When we implement the design on the ML310 board, on which the bus clock rate is up to 100MHz, the best case throughput becomes 8*8*100 = 6.4Gbps and the average throughput is 7.71*8*100 = 6.17Gbps. The bus clock rate of the ML310 board lowers the throughput that is possible in a simulation environment.

If the data to be scanned is well prepared in memory, the throughput of ClamAV is 912.7Mbps, and if DMA throughput is higher then the throughput of ClamAV is higher too. When DMA throughput reaches 7.8Gbps the throughput of ClamAV is 2.1755 Gbps.

If the ClamAV were implemented in the kernel space, the data passing between the user space and the kernel space could be removed. Copying the text to the DMA buffer is heavy, but this step is necessary for two reasons: (1) to ensure the text is indeed written into the memory, rather than only in the CPU cache, unless the cache is write-through, and (2) to ensure the text is stored contiguously in the physical addresses for the DMA to fetch. Note that the addresses of the text may be contiguous in virtual memory space, but not in physical memory space. The DMA must somehow know the physical addresses, perhaps through the Memory Management Unit (MMU), to fetch the text. Therefore, copying the text into the non-cacheable DMA buffer is a safe solution. If the buffer to store incoming text could be made non-cacheable, and the DMA can fetch the text in contiguous addresses or through the MMU, for example, the heavy overhead could be eliminated.

The data transfer from ClamAV into TextRam slows down the overall system performance. If ClamAV ran in the Linux kernel, dropping the text transfer from the user space to the kernel space is possible. The text must also be somehow directly stored in the memory without being cached in the CPU, and the DMA must be able to derive the physical addresses of the text. If the text to be scanned is well prepared in RAM, the throughput is restricted by the DMA, and raising the DMA throughput is critical.

Reference

[1] R. Boyer and J. Moore, “A fast string searching algorithm,” In Communications of the ACM, vol.20, no 10, pp762–772, October 1977.

[2] M. Aldwairi, T. Conte, and P. Franzon, “Configurable String Matching Hardware for Speeding up Intrusion Detection,” ACM SIGARCH Computer Architecture News, 33(1):99–107, 2005.

[3] B. L. Hutchings, R. Franklin, D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *fccm*, p. 111, 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02), 2002

[4] Zachary K. Baker , Viktor K. Prasanna, “A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs,” Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, p.135-144, April 20-23, 2004

[5] M. Attig, S. Dharmapurikar, and J. Lockwood, “Implementation results of bloom filters for string matching,” In Proceedings of the Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposiumon (FCCM’04), pages 322–323. IEEE Computer Society, 2004.

[6] Z. K. Baker and V. K. Prasanna, “Time and area efficient pattern matching on FPGAs.” In Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, pages 223–232. ACM Press, 2004.

[7] S. Dharmapurikar, M. Attig, and J. Lockwood, ”Deep packet inspection using parallel bloom filters,” Micro, IEEE, 24(1):52–61, January -February 2004.

[8] I. Sourdis and D. Pnevmatikatos, “Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System,” In Proceedings of FPL2003, 2003.

[9] Yi-Jun Zheng, Po-Ching Lin and Ying-Dar Lin, “Realizing a Sub-linear Time String-MatchingAlgorithm with a Hardware Accelerator UsingBloom Filters,” IEEE TVLSI, 2008.

[10] A. Aho and M. Corasick, “Efficient string match-ing: An aid to bibliographic search,” In Communications of the ACM, vol. 18, no. 6, pp.333-343, June 1975.

[11] Chen-Chou Hung and Ying-Dar Lin, “Automaton Based String Matching Hardware with Root-Indexing and Pre-Hashing Techniques: Design, Implementation and Evaluation,” Master’s thesis, National Chiao Tung University, 2006.

[12] A. C. Tao, “The complexity of pattern matching for a random string,” SIAM Journal on Computing, 8(3):368-387, 1979.

[13] S.Wu and U. Manber, “A fast algorithm for multi-pattern searching.” Tech. Rep. TR94-17, Dept. Comput.Sci., Univ.Arizona, May 1994.