國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

以 Bloom filters 硬體實作加速傳統次線性時間字串比對演算法: 設計、實作與評估

A sub-linear time string matching algorithm with Bloom filters acceleration: Design, Implementation and Evaluation

研 究 生：鄭伊君

指導教授：林盈達　教授

中 華 民 國 九 十 五 年 六 月

以Bloom filters硬體實作加速傳統次線性時間字串比對演算法：設計、實作與評估

A sub-linear time string matching algorithm with Bloom filters acceleration: Design, Implementation and Evaluation

研　究　生： 鄭伊君　　　　Student : Yi-Jun Zheng

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 Institute 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

June 2006

HsinChu, Taiwan, Republic of China

中華民國九十五年六月

以 Bloom filters 硬體實作加速傳統次線性比對時間字串比對演算法：設計、實作與評估

學生：鄭伊君　　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

目前針對封包內容作檢查的網路應用程式主要是用字串比對的方式來偵測封包中是否有入侵行為、病毒、廣告等惡意的傳輸資料。雖然目前針對字串比對演算法的研究已多不勝數，但用一般處理器跑軟體的運作方式上，由於其大計算量和頻繁的記憶體存取使得字串比對的處理速度存在一定的上限。所以在高速應用中已經走向使用硬體加速器來加速字串比對的運算。

本研究提出了一個應用 Bloom filter 的特性實作表格的查詢來達到次線性比對時間的硬體架構。此架構中利用二個機制來克服原本次線性時間演算法不適於硬體實作的因素，分別是：一、以平行詢問(parallel query)多個 Bloom filter 來取代原本演算法中需要存取在於外部記憶體的表格查詢動作。在次線性時間演算法中，普遍存在一個置於外部記憶體的大表格用以查詢判斷下一步掃描的動作；用 Bloom filter 的方式，不但可以模擬取代查詢表格的動作，更提供了比原本表格查詢更精確的資訊。二、設計一個非阻斷式(non-blocking) 的驗證介面，使得最差情況下的處理速度仍達到線性時間。次線性時間演算法在一般情形下的處理速度雖然是次線性時間，但是基於演算法原理，最糟情形下處理時間會比線性時間演算法長數倍。經由非阻斷式介面實作使得掃描與驗證的工作得以同時進行來達到最慢的處理速度會是線性時間演算法的處理速度。

本實作經過軟體模擬和 Xilinx FPGA 合成模擬後驗證無誤，最高速度可達將近 9.2Gbps；完全是病毒碼的驗證速度是 600Mbps。

關鍵字: 次線性時間、Bloom filter、字串比對、硬體、加速。

A sub-linear time string matching algorithm with Bloom filters acceleration: Design, Implementation and Evaluation

Student: Yi-Jun Zheng **Advisor: Dr. Ying-Dar Lin**

Department of Computer and Information Science

Nation Chiao Tung University

Abstract

Many network security applications heavily rely on string matching to detect malicious intrusions, viruses, spam, and so on. A software-based implementation may not meet the performance requirement of high-speed applications due to intensive computation and frequent memory accesses. A hardware solution to take advantage of hardware parallelism is a promising trend to inspect the packet payload at line rate.

In this work, we propose an innovative memory-based architecture using Bloom filters to realize a sub-linear time algorithm that can effectively process multiple characters simultaneously. The two key ideas to realize the sub-linear time algorithm in this architecture are (1) replacing the slow table lookup in the external memory with simultaneous queries to several Bloom filters and (2) designing a non-blocking verification interface to keep the worst-case performance in linear time.

The proposed architecture is verified in both behavior simulation in C and timing simulation in HDL. The simulation result shows that the throughput is nearly 10 Gbps for Windows executable files and 600 Mbps in the worst case.

Keywords: sub-linear time, Bloom filter, string match, hardware, acceleration

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Detecting and filtering proliferating attacks and viruses on the Internet requires content classification at the application layer, as opposed to traditional packet classification at the network and transport layers. Content classification involves string matching for patterns of malicious signatures, but string matching has been reported to be a bottleneck of network content security applications [1], [2], such as intrusion detection systems (IDS) and anti-virus systems. Consequently, the efficiency of string matching is critical to content processing. String matching algorithms implemented on general purpose processors is often not efficient enough to afford the escalating amount of Internet traffic, so several specialized hardware-based solutions have been proposed for high-speed applications.

A hardware accelerator can hardwire the signatures into logic cells [3-5] on the FPGA or store the signatures into memory [6]. The former can generally achieve higher performance because it is easier to implement pipelining and process multiple bytes per cycle with high area cost. The disadvantages are that the number of signatures is limited by the gate count, and that dynamic signature updates during matching is difficult due to long programming time. An ASIC implementation is also impossible since it is not reconfigurable. The latter is easily reconfigurable in signature updates. The number of signatures is limited by the memory size, but the limitation is a tiny problem if external memory is used. Moreover, the hardware can be implemented on an ASIC chip.

A study in [7] surveys and summarizes several architectures of string matching engines. As far as we know, existing solutions all implement linear time algorithms, say the well-known Aho-Corasick (AC) algorithm [8], that have to read every character in the text, and hence the time complexity is O(n), where n is the text length.

Although some designs can process multiple characters at a time at the cost of duplicating matching hardware components, the number of characters under simultaneous processing is quite limited in practice due to hardware complexity.

Unlike linear time algorithms, many existing sub-linear time algorithms can skip characters that can not be a match so that multiple characters are processed at a time in effect [9], [10]. Although they generally have higher performance than a linear time algorithm in software [10], [11], they are rarely implemented in hardware, probably because of two reasons. First, sub-linear time algorithms skip unnecessary characters according to some heuristics, typically by looking up a large table. The table may be too large to fit on the embedded memory, and thus be stored in external memory which is time-consuming to access. Second, although the average time complexity of the sub-linear time algorithms is roughly O(n/m), where m is the pattern length, the worst-case time complexity is O(mn), worse than O(n) in linear time algorithms. Therefore, sub-linear time algorithms are less resilient to algorithmic attacks that exploit the worst case of an algorithm to reduce its performance.

In this work, we propose an innovative memory-based architecture using Bloom filters [12] to realize a sub-linear time algorithm enhanced from the Wu-Manber (WM) algorithm [13], namely the Bloom Filter Accelerated Sub-linear Time (BFAST) algorithm herein. The proposed architecture stores the signatures in the Bloom filters that can represent a set of strings in a space-efficient bit vector for membership query, so this proposed architecture can accommodate a large pattern set and have low cost in pattern reconfiguration. The proposed architecture replaces the table lookup in the WM algorithm with simultaneous queries of several Bloom filters to derive the same shift distance of the search window. Thanks to the space efficiency of Bloom filters, the required memory space can fit into the embedded memory. To handle the worst case, a heuristic similar to the bad-character heuristic in the Boyer-Moore algorithm

[14] is adopted to reduce the number of required verifications in the WM algorithm. When a suspicious match is found in a position called the anchor, the anchor is passed to a verification engine without blocking the scan. The system also supports searching for a simplified form of regular expressions specified in terms of a sequence of sub-patterns spaced by bounded gaps, which can be found in some virus patterns [15].

The implementation result shows that this architecture can processes multiple bytes a time based on the theory of the sub-linear time algorithm to increase the throughput up to 9.2Gbps and utilize the efficient memory-usage Bloom filter to accommodate more than 10,000 patterns. The simplicity of the circuit design of this architecture makes this design can be integrated into the Xilinx XC2VP30 SOC platform to become a customized anti-virus chip.

The rest of this work is organized as follows. Chapter 2 reviews typical string matching algorithms and existing hardware accelerators. Chapter 3 introduces the architecture of the proposed sub-linear time algorithm. Chapter 4 discusses the detailed implementation issues, such as the interface between the search engine and the verification engine. This chapter also presents the system architecture as well as each component inside. Chapter 5 evaluates the proposed architecture and compares it with existing works. Chapter 6 concludes this work.

Chapter 2 Related Works

2.1 String Matching Algorithms

The single string matching problem is to search for all the occurrences of a string *p*, called the pattern, in the text $T = t_1t_2\ldots t_n$ on the same alphabet set, where *n* is the text length. The multiple string matching allows to simultaneously search for the patterns in the set $P = \{p^1, p^2\ldots, p^r\}$, where r is the number of patterns. Many algorithms have been proposed for various purposes, such as text searching, network content security, and bioinformatics. Some recent textbooks, such as [10] and [16], cover the algorithms for general applications. Lin et al. review and evaluate the algorithms particularly for network content security [9].

String matching algorithms can be linear time or sub-linear time in terms of time complexity. Linear time algorithms track every character in the text to decide whether a match occurs, so their running time is O(*n*). Sub-linear time algorithms can *skip* characters that can not be a match according to some heuristics, and thus can process multiple characters at a time in effect. The latter algorithms achieve the time complexity of roughly O(*n*/*m*) on average, so they are generally faster than the former.

2.1.1 Linear time algorithms

The Aho-Corasick (AC) algorithm [8] is a typical linear time algorithm found in some hardware designs. It constructs a finite automaton from the patterns, and then feeds the automaton with input characters one by one in the text for state transition. A match is claimed if one of the final states is reached. The time complexity of AC algorithm is O(*n*).

A state transition involves two memory accesses. One reads a character from the input text and the other accesses the transition table to determine the next state. Reading one character at a time limits the performance of AC. Modern data buses allow reading four or eight characters at a time, so the state transition from only one character is inefficient. Some hardware architectures allow processing more than one character at a time [17], [18]. Moreover, the transition table in AC is large for a large pattern set. Some compression methods have been proposed to reduce the memory requirement [19], [20].

2.1.2 Sub-linear time algorithms

The Boyer-Moore (BM) algorithm can skip characters that cannot be a match according to the bad-character and the good-suffix heuristics [14]. Some algorithms [13], [21], [22] based on similar heuristics extend the BM algorithm for multiple string matching. Among them, the Wu-Manber (WM) algorithm has been implemented in some popular open-source packages, such as Snort (*http://www.snort.org*) for intrusion detection systems and ClamAV (*http://www.clamav.net*) for anti-virus systems. Some variants of WM are also proposed for short patterns and longer skip distance [23], [24]. These algorithms are very efficient and their time complexity is sub-linear time on average.

The WM algorithm searches for the patterns by moving a search window of length *m* along the text, where *m* is the length of the shortest pattern in the pattern set. WM matches the rightmost block of the search window against the patterns. If the block *does not appear* in any of the patterns, the search window can safely shift *m*–*B*+1 characters without missing a match, where *B* is the block size. Otherwise, the shift distance is *m*–*j*, where *j* is the position of the last character in the rightmost occurrence of the block in the patterns. If the shift distance is 0, i.e., the block is the suffix of some patterns in the pattern set, the verification should follow to verify if a true match occurs. Without loss of generality, Fig. 1 illustrates an example with only one pattern for easy illustration. The search window can be shifted by 11–4+1 = 8 characters since the block ‘TEST’ does not appear in the patterns.

The WM algorithm calculates the shift values for every possible block of *B* characters and stores them in a shift table in the pre-processing stage. The larger the block size, the less likely the block appears in the patterns. However, a large block size also implies a large shift table for every possible block and limits the shortest pattern length that is allowed. The block size is 2 or 3 in practice. The mapping from the block to the table is not necessarily one-to-one. If more than one block is mapped to the same entry, the minimum shift value of them is filled in this entry. Such a mapping can save the table space at the cost of smaller shift values.

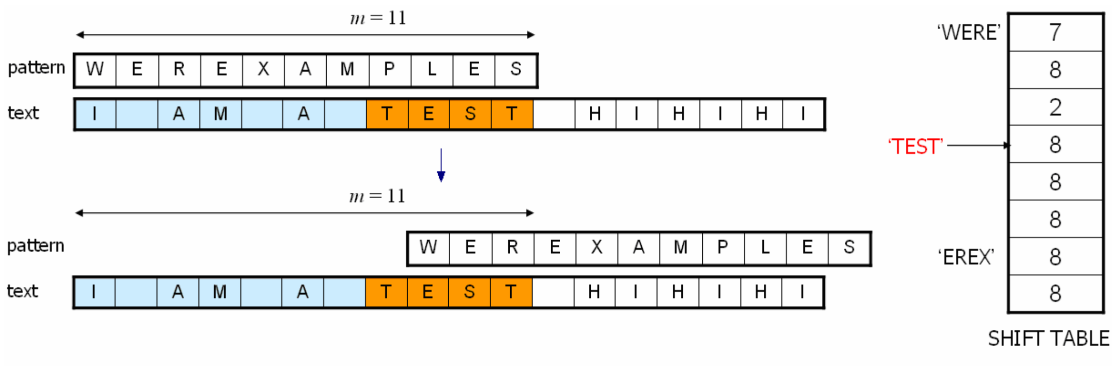

Figure 1. The search window can be safely shifted by *m*-*B*+1 characters by looking up the shift table according to the heuristic in the WM algorithm

Although the WM algorithm is fast on average, its worst-case performance is worse than linear time. For example, if a pattern in the pattern set is ‘aaaaa’ and the text is composed of all a’s, the search window cannot skip any characters. The time complexity becomes O(*mn*) because the verification takes O(*m*) in every position of the text.

2.2 Hardware accelerators

The patterns can be either hardwired into logic cells on FPGA or stored in the memory in existing hardware accelerators. Among these accelerators, we particularly pay more attention on Bloom filters which the proposed architecture is based on.

2.2.1 Hardwire-based accelerators

Although storing the patterns into the look-up tables (LUTs) on the FPGA is feasible, the reconfiguration is costly. Updating the patterns takes hours to regenerate a new bit-stream including the patterns and a few minutes to download it onto the chip. The cost can be reduced by incremental MAP and PAR [25] or by partial reconfiguration. Besides the reconfiguration cost, the size of the pattern set is limited by the number of available gate counts on the FPGA, so these accelerators are not scalable to a large pattern set. A few typical designs are introduced below.

Sidhu et al. [26] designed a low-cost solution for matching regular expressions. It represents the patterns in regular expressions as non-deterministic finite automata (NFA) or deterministic finite automata (DFA). Its performance is not high due to the state transition of only one single character per cycle. Moscola et al. duplicated matching modules to scan multiple packets concurrently. They quadrupled the throughput up to 1.184Gbps [27].

Cho et al. [3] designed a pipelining architecture of discrete comparators. Each pattern is matched in a pattern match unit. Each stage in the unit involves four sets of four 8-bit comparators to simultaneously match four consecutive characters each starting from four consecutive positions in each cycle. The matching results in each stage are fed to the next stage for the pipelining. This approach can operate at high frequency, so the performance can be fairly good. Sourdis [28] improved the architecture by fully pipelining the entire system and using a fast fan out tree to distribute the incoming data to each comparator. Thus, it can work at 344MHz on the Xilinx VirtexII-1000 FPGA, and the throughput is up to 11 Gbps. However, its area cost is high. Several following studies were devoted to area reduction, such as [5].

2.2.2 Memory-based accelerators

The memory-based accelerators store patterns in the memory, which can be either the embedded memory on the FPGA or the external memory connected through the system bus. The accelerators can be implemented on either ASIC or FPGA. When the patterns are updated, only the memory content is reloaded with the new pattern set and the processing logic remains unchanged. Hence the reconfiguration cost is low. Storing the patterns in the embedded memory is helpful to a high-speed design because of the high memory bandwidth. However, the size of embedded memory is usually too small to store a large pattern set. A memory-efficient mechanism, namely Bloom filtering, can be used in a high-speed matching engine [29].

A Bloom filter can represent a set of strings compactly in an m-bit bit vector for efficient membership queries [30]. Given a string x, the Bloom filter computes k hash functions $h_1(x)$, $h_2(x)$, …, $h_k(x)$ that produce k hash values ranging from 1 to m. The filter then sets k bits at positions $h_1(x)$, $h_2(x)$, …, $h_k(x)$ in the bit vector. The procedure repeats for each string in the string set to program the filter. A membership query for a string w looks up the bits at positions $h_1(w)$, $h_2(w)$, …, $h_k(w)$. If any one of the k bits is unset, it is impossible for w to be a member in the pattern set; otherwise, a match may occur. A verification phase follows a suspicious match to verify whether a true match occurs. Bloom filtering does not have false negatives, but may have false positives. The number of false positives can be reduced with proper choices of k and m.

Dharmapurikar et al. [29] designed a hardware architecture using Bloom filters to inspect packet content. Assume that the signature lengths range from L_{min} to L_{max}. The architecture groups the signatures by length and stores each group of signatures in an individual Bloom filters, so totally L_{max}-L_{min}+1 Bloom filters are in this architecture. A window of L_{max} characters reads one character from the text in each cycle. Each Bloom filter matches in parallel a substring of length i, $L_{min} \leq i \leq L_{max}$ in the window. If a suspicious match is declared by one of the Bloom filters, this match will receive further probing by an *analyzer* that verifies if it is indeed a true match. Otherwise, if no match is found, the window can safely advance to next character because no false negatives can occur. This architecture can be extended to advance G

characters at a time at the cost of duplicating *G* sets of parallel Bloom filters. The number of Bloom filters in this architecture becomes $G(L_{max}-L_{min}+1)$. Parallel access to the bit vector from so many Bloom filters is not always feasible, and thus *G* is quite limited in practice.

Chapter 3 Algorithm Design

3.1 The Bloom Filter Accelerated Sub-linear Time algorithm

This work designs the hardware architecture for the sub-linear time algorithm extended from the WM algorithm to accelerate multiple string matching. The key points to embody the design are avoiding the need of a large shift table and reducing the impact from the worst case on performance.

3.1.1 Drawbacks of using a shift table

The WM algorithm looks up the shift values in the shift table by indexing the block in the suffix of the search window during scanning stage. A block of fewer than three characters is very likely to appear in a large pattern set, say that of virus signatures, and thus the shift distance will be mostly short and the verifications will be frequent according to the WM heuristic. A larger block of at least three characters can improve the situation, but it also leads to a large shift table. For example, 256^3 entries in the shift table are required to store the shift value of every block of three characters. It amounts to memory space of 16 MB if each entry takes one byte. A block size of larger than three is almost impractical due to the huge table size. Although compressing the table by mapping more than two blocks to an entry is possible, the shift distance will be reduced because the shift value in an entry is the minimum of all the blocks mapped to that entry. The shift distance will be reduced and the number of verifications will be increased significantly if the table is compressed too much.

A large table is unable to fit into the embedded memory, but if the table is stored in the external memory, the slow memory access will slow down the overall performance. Moreover, the shift values in the shift table can be indexed only from the rightmost block of the search window. If a shift value of zero happens frequently, the frequent verifications will slow down the overall performance. The BFAST algorithm keeps the positions of the blocks in the patterns so that not only the rightmost block, but also the other blocks in the search window can derive their position in the patterns. Therefore, the algorithm can use a heuristic similar to the bad-character heuristic in the Boyer-Moore algorithm to determine a better shift value. This benefit will be demonstrated in the next sub-section.

3.1.2 Implicit shift table using Bloom filters

Let B_o be the rightmost block in the search window. The shift distance is a function of the positions of B_o or its suffix in the patterns [24], so separately storing the blocks in each position of the patterns is sufficient to derive the shift distance. Fig. 2 shows an example of this derivation. Assume current block of the text is ‘XAMP’ , it appears in the fourth last block of the pattern ‘EXAMPLE1’, and thus the shift distance of ‘XAMP’ should be 4 to fetch the block ‘PLE1’ and check if its shift distance is 0 as illustrated in the Section 2.1.2. The shift value is derived formally from the Equation (1)

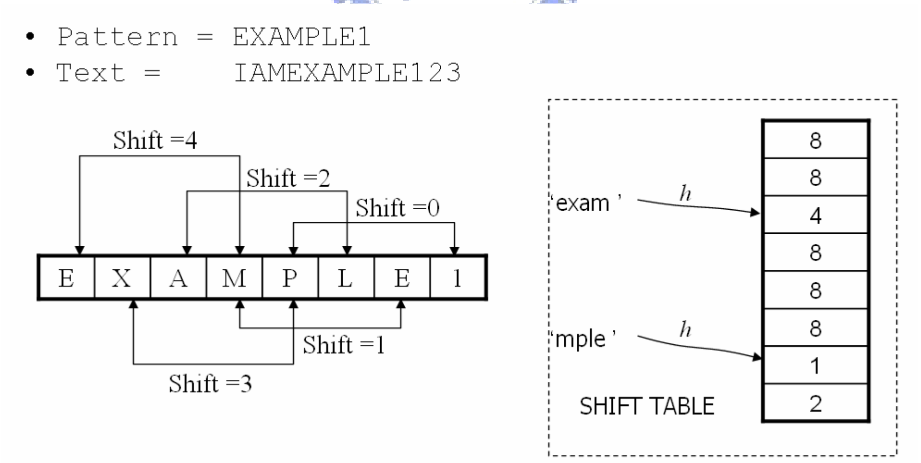

Figure 2. Shift distance of a block can be derived from its position in the patterns

$$Shift(B_0) = \begin{cases} m-j, & \text{if the rightmost occurrence of } B_0 \text{ is in position } j \text{ of some patterns} \\ m-k, & \text{if } B_0 \text{ does not appear in the patterns, but its longest suffix of length } k \text{ is the prefix of some patterns} \\ m, & \text{otherwise} \end{cases} \quad (1)$$

With this derivation of shift distance, we can replace the shift table lookup operation with membership query of parallel Bloom filters. Bloom filtering is a space-efficient approach to store strings in the same length for membership query, i.e. to check if one string belongs the string set or not. By grouping the blocks in *different positions* of the patterns and storing these groups in separate Bloom filters, we can know whether a block belongs to the pattern or not and its position by querying these Bloom filters in parallel.

Fig. 3 illustrates how to establish an implicit shift table using Bloom filters. Assume the pattern set is {P_1, P_2, P_3}. After dividing by position, the Group 0 is {efgh,mnop,vuts}, Group 1 is {defg,lmno,wvut}, and so on. If the block of text is “cdef”, the query result will be Group 2 hit, so the shift distance is 2. If there is no hit reported, then it means there is no such block in the patterns, we can safely shift maximum shift distance or 8 in this example.

- P_1 = abcdefgh
- P_2 = ijklmnop
- P_3 = zyxwvuts
- Grouping:
  - G_0 = {efgh, mnop, vuts}, G_1 = {defg, lmno, wvut},
  - G_2 = {cdef, klmn, xwvu}, G_3 = {bcde, jklm, yxwv},
  - G_4 = {abcd, ijkl, zyxw}, G_5 = {abc, ijk, zyx},
  - G_6 = {ab, ij, zy}, G_7= {a, i, z}

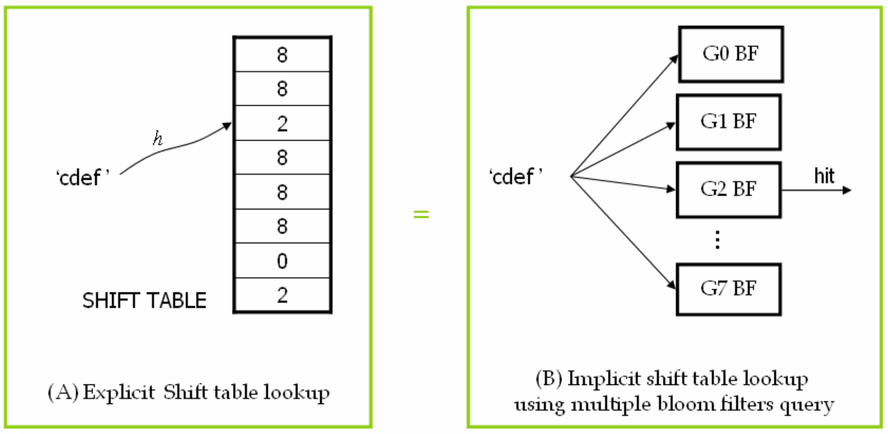
'cdef'
h
8
8
2
8
8
8
0
2
SHIFT TABLE
(A) Explicit Shift table lookup
=
'cdef'
G0 BF
G1 BF
G2 BF
hit
G7 BF
(B) Implicit shift table lookup using multiple bloom filters query

Figure 3. Grouping of blocks in the patterns for deriving the shift distance from querying Bloom filters. The shift table in the WM algorithm becomes implicit in the Bloom filters herein.

The grouping is defined formally in Equation 2:

$$G_i = \begin{cases} \{p_{m-i-B+1} \ldots p_{m-i} \mid p \in P\}, 0 \le i \le m - |B_0| \\ \{p_1 \ldots p_{m-i} \mid p \in P\}, m - |B_0| + 1 \le i \le m - 1 \end{cases} , P: \textit{the pattern set}. \qquad (2)$$

The membership query of the Bloom filters may have false positives. In other words, a block may not exist in a group, but the corresponding Bloom filter of that group may be hit. The shift distance will be smaller than it should be as the false positives happen, but the search is still safe: no match will be missed. As long as the number of false positives is controlled within a small value with proper parameter setting of the Bloom filter, say the length of the bit vector, the false positives will not be an issue.

3.1.3 Additional checking in the Bloom filters

Although G_0 is rarely hit for random samples, i.e. the block is not in the rightmost block of the pattern, this is not always the case in practice such as the reason illustrated in Section 3.2.1. Therefore, unlike the original WM that verifies the possible match immediately, the BFAST algorithm continues the checking the block $B_1, B_2, \ldots, B_{m-|B|}$ like the bad-character heuristic in the Boyer-Moore algorithm, where B_j stands for the $|B|$ characters that are j characters away from the rightmost character backward in the search window. If the Bloom filter of G_i is hit, where $i > j$, the shift distance can be $i - j$. The reason is much like the bad-character heuristic in the Boyer-Moore algorithm. A shift less than $i - j$ cannot lead to a match because B_j cannot match any blocks in groups from G_{i-1} to G_j. The verification procedure will follow to check whether a true match occurs only if every block from B_0 to $B_{m-|B|}$ is in Bloom filters of G_0 to $G_{m-|B|}$. For example, Assume the text is ***abcdefghijk****lmn*.... When the querying result of a block ***hijk*** is reported hit in the group 0, i.e. the shift distance equals to 0, we take the preceding block ***ghij*** to query the bloom filter of group 1. If it still hit, we continue to use the preceding block ***fghi*** to query group 2, otherwise, we declaim verification end and move on to scan the block ***ijkl*** which is the next block of the one caused the verification, i.e. the block which shift distance is 0. This verification procedure repeats until querying the last group. If all the groups are hit, Anchored AC verification is involved. This further verification can reduce significantly the number of verifications in the WM algorithm. In the simulation using 10,000 patterns, this approach can reduce the number of verifications by around 50%.

3.1.4 Worst case handling

The performance of a sub-linear time algorithm, say the WM algorithm, may be low in some cases. First, when the pattern length is close to the block size, the shift distance of m - $|B_0|$ + 1 will be very short, given $m \geq |B_0|$. The BFAST algorithm can process at least four characters in each shift of the search window, while the shift distance in the WM algorithm can be as short as one or two characters in the same case. Second, the worst case time complexity can be as high as O(mn) if the patterns occur in the text frequently. Consider the extreme case that the characters both in the text and in some patterns are all a's, verification is required after each shift of only one character. To increase the performance in the worst case, this work uses a linear time algorithm, Anchored-AC, to co-work with this sub-linear time algorithm for the verification. The verification result is reported to software (upper-layer applications) directly by the verification engine. The interface between the search engine and verification engine communicates through a descriptor buffer. As long as the buffer is not full, the search engine can always offload the verification and move on to scan the next block without blocking after finding a potential match.

3.1.5 Advantages of the proposed architecture

This architecture can successfully process multiple characters at a time with the number of Bloom filters on the order of at most O(m). Compared with other Bloom-filter-based architectures, such as [29], which demands the Bloom filters on the order of O(*ms*), where *s* is the allowed shift distance, the proposed architecture has the two major advantages. First, the number of Bloom filters required is reduced for the same purpose of processing multiple characters at a time. Second, the proposed architecture allows long shift distance. For example, if the shortest pattern length is 10, the proposed architecture allows shifting as many as 10 characters at a time. This is not feasible in the architecture of [29] because the number of Bloom filters is large and simultaneous accesses to the bit vector from so many Bloom filters are difficult. Moreover, as far as we know, no other hardware architecture can have such long shift distance so far.

3.2 Design issues

Besides the algorithm itself, the implementation impacts on the performance significantly. This section focuses on the practical design issues in the implementation.

3.2.1 Characteristics of the pattern set

Because this design benefits the application with a large pattern set, we choose anti-virus as the target application. Therefore, we analyze Windows executable files as the text. The block distribution in the Windows executable files is non-uniform. Table 1 presents the top 10 appearing blocks in 1,000 Windows files we selected.

Table 1. Top 10 appearance blocks in 1,000 windows files.

	block	%		block	%
1	000000	48.7%	**6**	202020	0.6%
2	ffffff	4.7%	**7**	020000	0.4%
3	909090	4.3%	**8**	00008b	0.4%
4	010000	1.1%	**9**	0083c4	0.3%
5	cccccc	0.7%	**10**	404040	0.3%

The ‘000000’ appears very frequently in the text, up to 48.7%. This non-uniform distribution leads to the high frequent hit in the Bloom filter of G_0 when there is a pattern that ends with an all zero block.

3.2.2 Hash functions

The formula of false positive rate in the Bloom filter is based on the universal hash functions. If the hash function is not universal, the false positive rate will be higher. The more false positives happen, the more verification procedures are required. Therefore, the selection of proper hash functions is critical to the system performance. This works uses the class of universal hash functions which are easily implemented in hardware proposed by [34]. For any bit in the block X with b bits represented as $X= <x_1, x_2, x_3 \ldots x_b>$, the hash function is calculated as $h(x) = d_1 \cdot x_1 \oplus d_2 \cdot x_2 \oplus \ldots.. \oplus d_b \cdot x_b$, where " $\cdot$ " is a bitwise AND operator and $\oplus$ is a bitwise OR operator, where d_i are pre-generated random number ranging from 0 to the bit vector size of m. After simulating the Bloom filters by randomly generated text, we find this hash function can achieve lower false positive rate.

Chapter 4 Implementation details

4.1 String match architecture

The string matching architecture includes two main components: (1) the scanning module, which is the main block performing the proposed algorithm that queries Bloom filters and shifts the text according to this querying result, and (2) the verification module and interface. When the scanning module finds a potential match, it instructs a verification job by filling an entry in verification job buffer in the verification interface. Fig. 4 shows the block diagram of the entire architecture. Each component in this architecture is described in following sections.

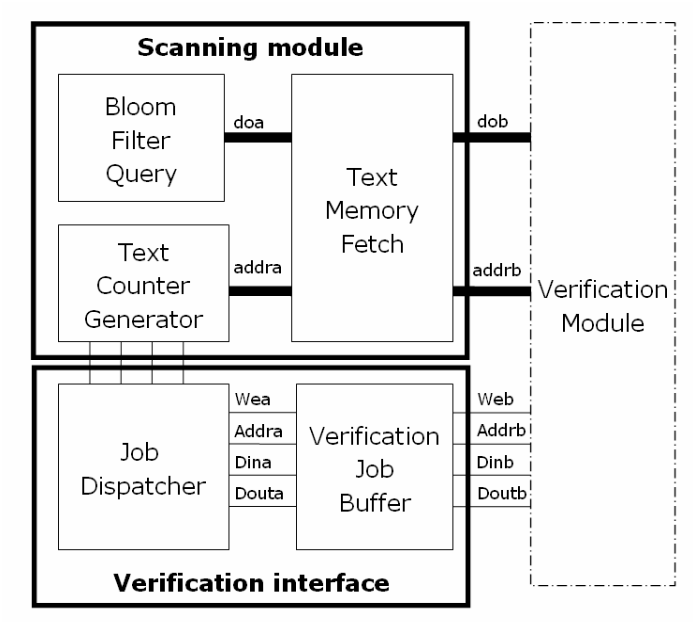

Figure 4. Overview of the string matching architecture

Each shift in the text includes three operations implemented in three separate sub-modules in the scanning module.

1. *TextMemoryFetch* fetches the suffix block of the search window in the text memory.
2. *BloomFilterQuery* queries the Bloom filters to find which group(s) the block belongs to.
3. *TextPositionController* calculates the location (address) of the search window in

the text memory on the next round according to querying result from the Bloom filters.

4.1.1 Text memory fetching

The block size is set to a word of four bytes for accessing memory efficiently and reducing matching probability of a random block. For parallel accessing four continuous memory bytes, the text memory is divided into four interleaving banks. Fig. 5 illustrates an example of fetching a word of ‘BCDE’ starting with the byte addresses $\underline{00}01_2$. Note that the characters in the text are interleaved in each memory bank and the first character to fetch locates in bank1. The underlined bits in the address except the last two are word address. The byte offset is decoded to fetch the correct byte in each bank. The fetched word is rotated according to the byte offset from a multiple of four.

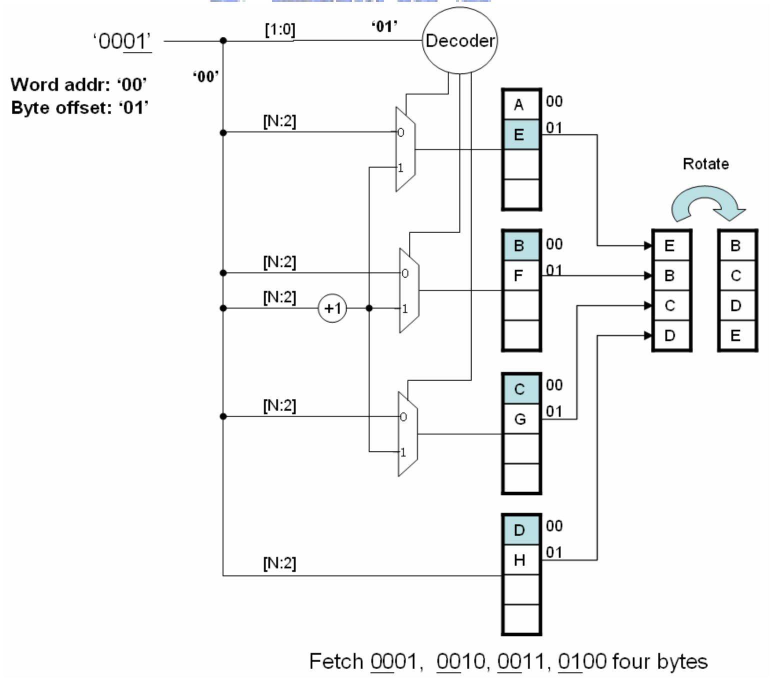

Figure 5. An example of fetching four bytes in $\underline{00}01$, $\underline{00}10$, $\underline{00}11$ and $\underline{01}00$.

4.1.2 Bloom filter querying

There are *N* independent Bloom filters storing different block sets in the patterns grouping with their positions in the patterns, where *N* corresponds to the group number. The block fetched by the *TextMemoryFetch* module queries these *N* Bloom filters in parallel to get the membership information. After the query, the priority encoder in *TextPositionController* encodes the membership information into the shift distance as illustrated in Chapter3. The block diagram of the *BloomFilterQuery* module is presented in Fig. 6.

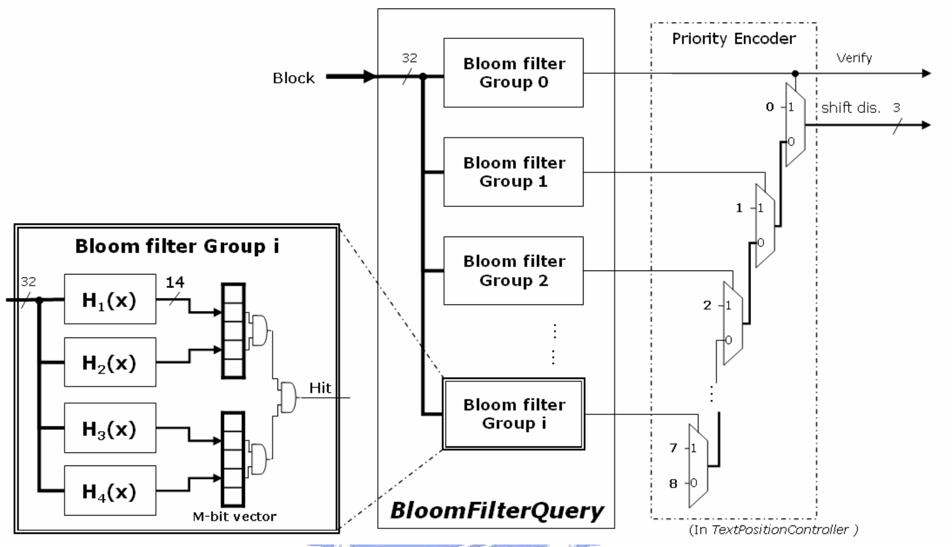

Figure 6. *BloomFilterQuery* module architecture

Because the bit vector has to be long enough to reduce the false positive rate, the on-chip dual-port block RAM is a lower cost way to implement it than flip-flops. Fig 6 is a example using 16kb block RAM on Xilinx XC2VP30 to implement one Bloom filter. Each block RAM is configured as a single bit wide and 16kb long bit array, and can be read write on two port simultaneously to support two hash function. Thus, the false positive rate *f* of ONE block memory is $\left(1 - e^{-\frac{2n}{16384}}\right)^2$ [29], where n is the number of pattern blocks stored in that bit vector. Using k block memory can reduce this rate to f^k, it is very close to the false positive rate of one k*16kb memory of 2k ports. The hash functions are independent, so they can be calculated and fetch the M-bit bit vector in parallel.

4.1.3 Text position controller

The *TextPositionController* maintains the position of the suffix block in the search window of the text and calculate the next position according to the membership information of the *BloomFilterQuery* and current matching state. A finite state machine keeps five states to control how the position is calculated. Fig. 7 illustrates the state transition diagram.

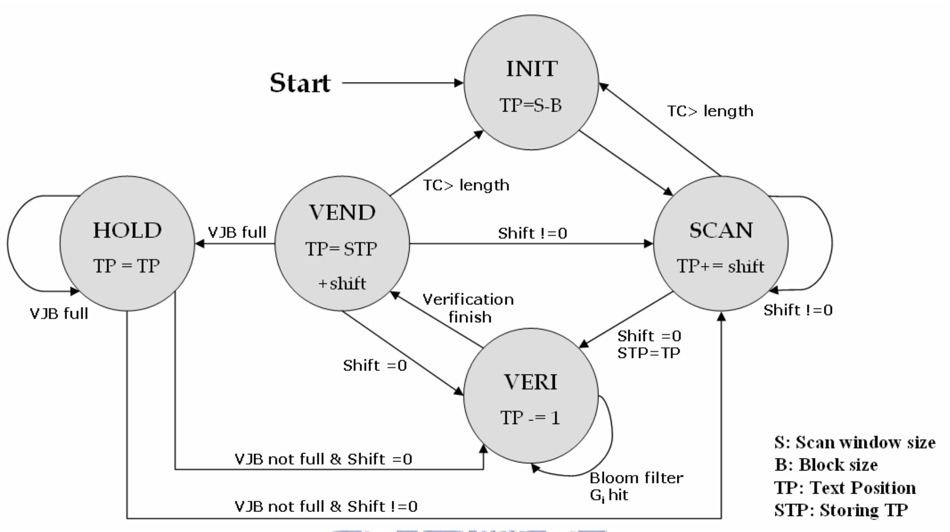

Figure 7. Text position controller state transition diagram.

1. In the beginning, set the initial address according to the scan window size and block size. For example, the scan windows size is *l* and the block size = *b*, the initial text position is *l* - *b*
2. When the shift distance is non-zero, i.e. no potential match, it adds the shift distance to the text position to get the next one.
3. When the shift distance is zero, it substrates 1 from the text position to get the preceding block in the text to take additional checking illustrated in Section 3.2.2 and stores the text position of this hit block for going to next block as verification finished.
4. When the additional checking finished, it shifts by the shift distance of the non-hit block if no match or report a match and just shift one byte to find next match.

5. When there is a potential match, i.e. additional checking reporting match, but the verification job buffer is full and thus there is no space for instructing a verification job. *TextPositionController* halts to wait for a free entry to be filled, so the text position is not change in this state.

4.1.4 Verification interface

This work defines a flexible verification interface rather than implements a specific verification mechanism to let verification mechanism to be replaceable according to different applications. The verification mechanism is beyond this work, so we just briefly describe the advantages of the approach we take in this work.

This work takes anchored Aho-Croasick algorithm to verify the suspicious data for two reasons. (1) Its data structure allows high compression rate. It compresses the original AC date structure to 1 Mb that stores 1000 patterns, almost 0.2%, that can be put into the Virtex-II Pro platform we used for experiment. (2) Its time complexity is linear in the worst case. Due to the potential match is very possible to be a true match, i.e. a virus; a linear worst case time algorithm is efficient to discover it.

There are two parts in the verification interface: *JobDispatcher* and *VerificationJobBuffer (VJB)*. When the scanning module discovers a potential match, it instructs the *JobDispatcher* to fill the *verification job descriptor (VJD)*, composed of text position, length and other related information to the VJB. The format of VJD is illustrated in Fig. 8. The most significant bit of the VJB is set when it is allocated. The verification module should test this bit to know if there is a new verification job and decode the text position and length and other information it needs to verify. After it finishes the verification, it should clear the entry it verified to free the entry.

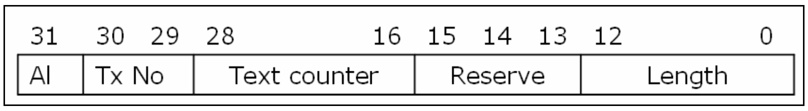

Figure 8. Verification Job Descriptor format

The VJB is implemented with one 16 kb dual-port block RAM too, so there are 16k/32 or 512 entries in it. The *JobDispatcher* keeps a VJB pointer. After scanning modules allocated one entry, the pointer is moved to the next entry. When it finds a potential match next time it reads the allocation bit to tell if it is empty. The pointer will rotate when it comes to bottom of VJB, so if it finds allocation bit is set; which means the VJB is full. This can be improved by more than one verification module to verify the jobs in the VJB to balance the verification and scanning speed.

4.1.5 Pipeline the design

In the original multi-cycle design, the location of next block is decided by the shift values derived from the Bloom filters to continue the next round of matching, so there is only one active module at a time like the Fig. 9(a). We pipeline the design by dividing the text into four independent segments like the example of Fig. 9(b). If the length of text is *m*. The segments are the $0 \sim \frac{m}{4}-1$, $\frac{m}{4}-s+1 \sim \frac{m}{2}-1$, $\frac{m}{2}-s+1 \sim \frac{m}{4}*3-1$, $\frac{m}{4}*3-s+1 \sim m-1$. The range of one segment is decided by dividing the original text by four plus the scan-window-size to avoiding the pattern across the segments. For example, assume length of text is 40. The four segments will be 0~9, 3~19, 13~29, 23~39. In this way of dividing the text to four independent segments, the *TextPositionController* can assign four start addresses at every cycle of four in the beginning, and calculate the second block position of the first segment at the fifth cycle: 7+S_1, and the second block position of the second segment at the sixth cycle: 10+S_2, and so on, where S_1 is the shift distance of first segment at the first time query, and S_2 is the shift distance of second segment at the first time query.

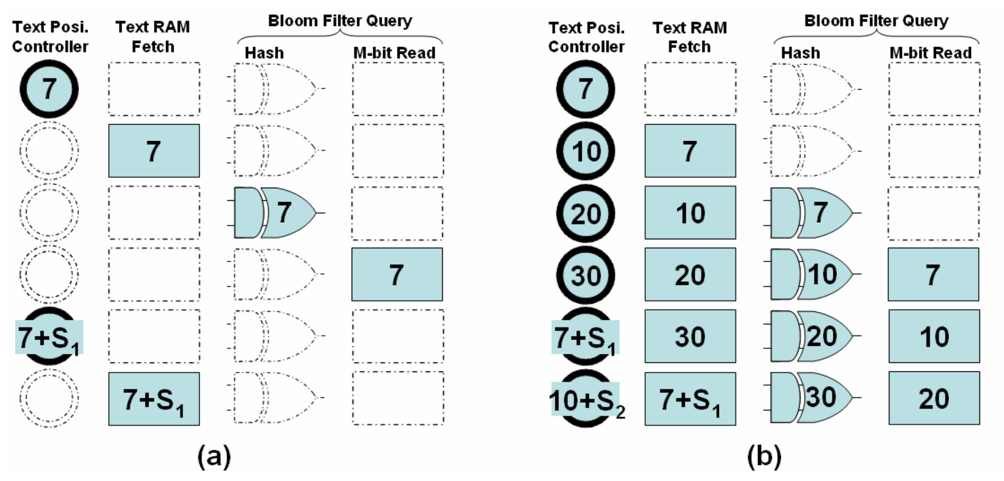

Figure 9. (a) State diagram of pipelining

4.2 Integration to Vertex II pro platform

Besides the string match module, this work needs additional efforts to integrate it into the system to test its functionality on FPGA chip. This work is implemented on a Xilinx SOC FPGA development platform XC2VP30. The well-tested soft IP supported by Xilinx with this chip can be used to quickly integrate the user-defined logic using Xilinx development tool EDK to become a complete and customized system. The user-defined logic only needs to use a generalized IP interface (IPIF) as a wrapper to communicate with the other components in the system without dealing with the timing. The functions in the IPIF can also been customized using the EDK, such as interrupt supporting, S/W register supporting, address range supporting, or DMA supporting. Therefore, we only need to define the communication interface, use the template files generated by development tool, and connect the I/O between IPIF and our designs.

The interface between processor and string matching module in this work defined as figure 10.

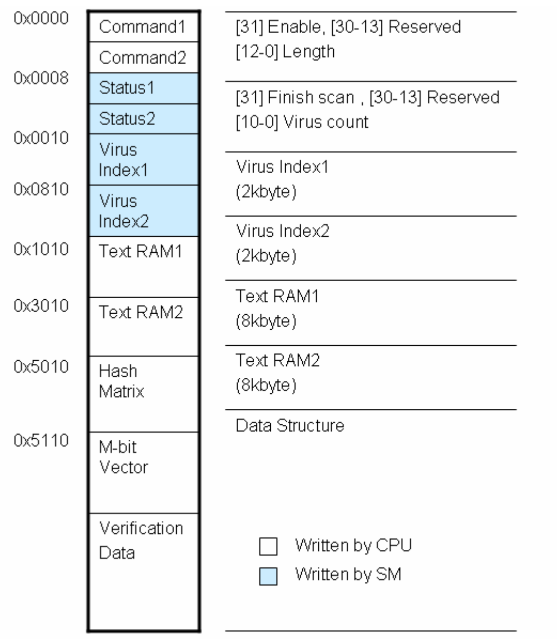

Figure 10. Addressing space of string matching

There are two copies of the communication data to scanning one text segment and transferring another segment at the same time. One communication data includes (1) a *command register* that sets up the length and enable bit of one transaction, (2) a *status register* to report the matching virus count, (3) a block of v*irus index memory* to report all the matching virus identifications and (4) a block of *text memory* to store the text to be scanned. Besides the data communication in scanning time, scanning data like m-bit vector and hash functions or verification data like transition states of Anchored-AC are needed to be updated before scanning.

Chapter 5 Experimental Results

5.1 Simulation and synthesizing result

To verify the design, this work runs a behavior simulation in C to measure the performance in different pattern count. Besides, this work also runs a timing simulation in HDL to find the critical path delay to estimate the clock rate.

5.1.1 C Simulation result

Setting the same false positive rate of each group of Bloom filter by let the *m/n* and *k* are all the same in the Bloom filter false positive rate formula $f = \left(1 - e^{-\frac{nk}{m}}\right)^k$ [29], where m is memory size, n is pattern count, k is hash function number, we measured the shift distance in different pattern count showed in Fig 11. We use both windows executable files and random generated data as scanning text to run simulations. Although the shift distance becomes smaller as the pattern set going larger, it maintains at greater than 5 that means five times faster than traditional linear time algorithm as the pattern count is 52k larger than 30k in Anti-Virus.

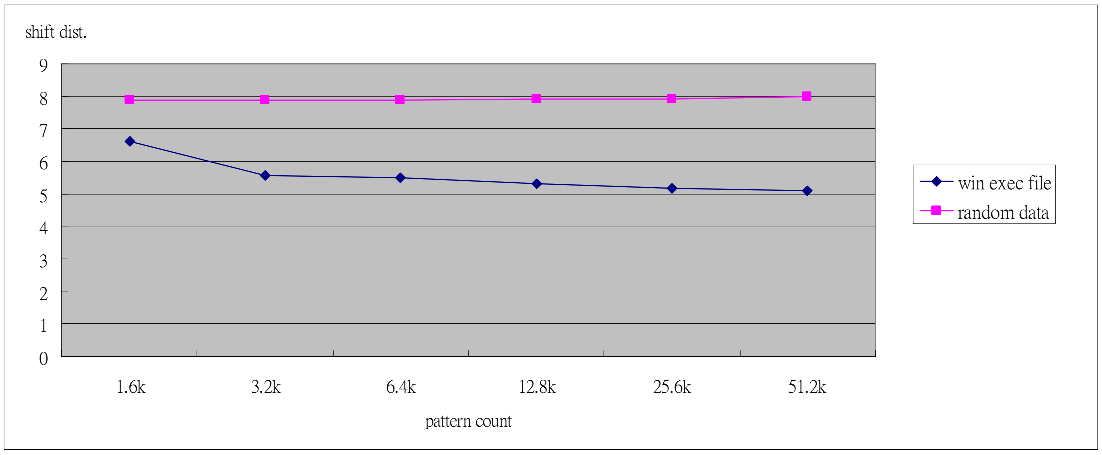

Figure 11.. Shift distance in different pattern counts (read/random)

The positive rate in the scanning module and the positive rate of a single Bloom filter for different pattern counts are showed in Fig. 12. The false positive rate of a single Bloom filter is set to constant as before. There are three observations in this false positive simulation result:

1. Positive rate of the result confirms to theoretical false positive rate when the text is *random generated files*, but goes much higher when the text is *Windows executable files* because the same reason mentioned in Section 3.2.1. Therefore, we can simply take the curve of random generated files as false positive rate curve and take the curve of Windows executive files as the true positive plus false positive rate in this figure.
2. The positive rate of scanning Windows executable files increases as the pattern set goes larger, but keeps almost the same in scanning the random generated files whatever the single Bloom filter or entire scanning module. Taking the observation in 1, we verify the positive reported and we find that this difference is also induced by many true positive happening as the pattern count growing.
3. The additional check of checking the preceding block of the first hitting block is useful as the pattern count growing. When the pattern count is 52k, it almost filters the verification to 50%.

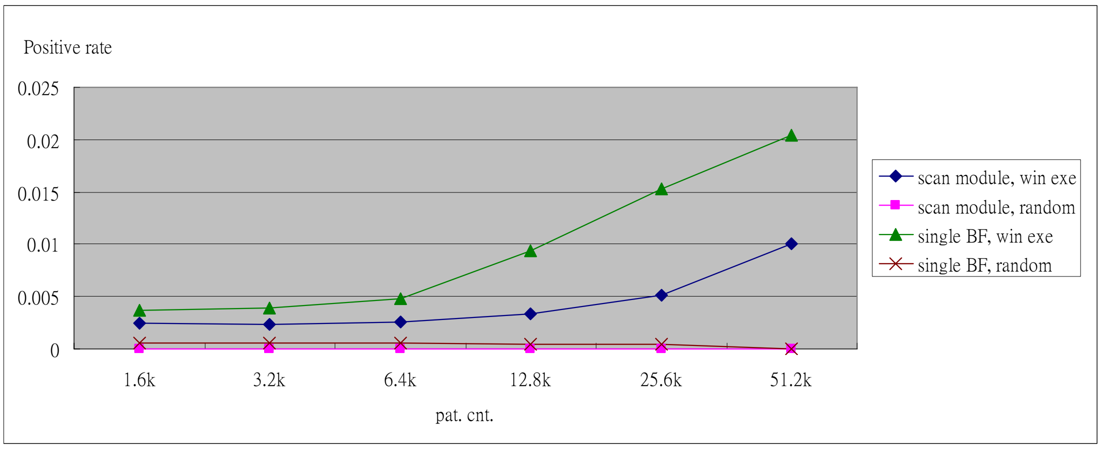

Figure 12. Scanning module false positive rate in different pattern counts

5.1.2 HDL simulation result

Xilinx XCVP30 FPGA has 136 dual-port embedded block memories. Each of it can be configured single bit wide, 16384 bit long array. We use 16 block memories which implement 8 Bloom filter groups, two for each that supporting 4 hash functions and storing 1000 blocks. The theoretical positive rate of each Bloom filter is 0.017%, and becomes 0.498% in the simulation of the windows executive files and 0.0015% of random generated files. Besides the m-bit vector of Bloom filters, this works takes 2*4 or 8 to implement two four-bank text memories. Furthermore, for fast prototyping, the state transition data needed by verification module, Anchored AC, is moving into the FPGA embedded memory. That costs almost 1 Mb to store 1000 patterns which is the reason why the pattern count of this HDL simulation is limited in 1000. The total account of memory cost is about 1.4 Mb in this implementation. If the verification data keeps in external memory, the pattern set of scanning module using this platform can be scale to about 10k and maintaining the scanning performance. The penalty is the longer worst case running time, i.e. verification time.

This architecture implementation consumes 16% usage of LUT of XC2VP30 and the system operates at 150MHz, and the average shift distance is 7.71 bytes. If the scanning module is not blocked by the verification module, i.e. the VJB is always not full; the throughput can be up to 150*7.71*8 or 9.26 Gbps. In our simulation of clean windows executive files, the verification module needs 5 cycles to verify one entry is not virus in average, but the scanning module issues a verification job every 26 cycles in average. Therefore, the assumption of not being block is established in average case.

The worst case performance occurring when the text are full of viruses is depends on the virus ratio in the text, the signature length and the matching policy, i.e. one match or multiple matches. The throughput in different parameters combination is aimed to be implemented in the future work and not being analysis in this work. We only simply measure the worse case performance in one condition: when the VJB is full. Verification speed is the bottleneck of entire system, which is one character for two clock cycles equaling to 150/2*8 or 600Mbps.

5.2 Comparisons

5.2.1 Compare with the Dharmapurikar et al.'s Bloom filter

The Bloom filters have been used to inspect packet in the research of Dharmapurikar et al. in 1997 [29]. They hash the m-bit vector with entire patterns, not only a fix-length (scan window size) prefix of patterns in this work. Therefore, there are L_{max}-L_{min} Bloom filter engines in their design, where L_{max} is the longest pattern length, and L_{min} is the shortest pattern length, i.e. the number of Bloom filter engine depends on the pattern length distribution. On the other hand, in this work, the Bloom filter number is decided flexibly by the scan window size. The bigger scan window size, the larger shift distance, and the less verification rate in general. Of course, the scan window size has to be shorter than the shortest pattern length following the way of WM algorithm works. Besides, the Dharmapurikar's design scans the packets from the beginning straightforwardly and shifts one byte if no match. It duplicates scan engines to process multiple bytes a time unlike this design utilizing the algorithm advantage to meet this requirement. Altogether, the low circuit cost and the high throughput are the main advantages of this work comparing with the Dharmapurikar's design. The disadvantage of this design is the higher verification rate. This design only matches the fix-length prefix of patterns, and thus the matching rate is higher then the rate of matching entire pattern of the Dharmapurikar's design, especially when the block distribution is very non-uniform like Windows executive file. Thanks to the implementation of verification interface in this design, the verification job can be finished on time before the next verification job is assigned to complement this defect in average. Table3 lists the comparison results in more detail. One thing of this table needs to be addressed. The false positive rate of [29] is simulated with the unknown signatures, and the false positive rate of this work is simulated with ClamAV virus database. The distribution of these two kinds of signature is different, so the number of false positive rate is just for information; the ratio between them is meaningless.

Table 2. Comparisons between Dharmapurikar's Bloom filter and BFAST.

Name	**Pattern count**	**Signature**	**Text**	**False positive rate**	**Memory usage**
Bloom	35,475	N/A	live traffic	0.39%	629,146
Filter[6]	(1419/BF)	length 2~26			
BFAST	1,000	clamav virus	Win.exec.files	0.323%	262,144
(1,000)			random	0.0%	
BFAST	10,000	clamav virus	Win.exec.files	0.390%	1,048,576
(10,000)			random	0.0%	
Name	**Implement**	**Circuit Size (LC)**	**Bloom filter engine #**	**Frequency (Mhz)**	**Throughput (Mbps)**
Bloom	Xilinx	23328	25	80	502
Filter	Virtex 2000e				
BFAST	Xilinx	3325	≦8	156	9,631
(1,000)	Vertex2p30				
BFAST	Xilinx	5438	≦8	140	5,775
(10,000)	Vertex2p30				

5.2.2 Compare with the other related works.

Fig.13 and Fig.14 show the pattern size and the throughput of several accelerators of string matching. The architecture of the researches may have high performance like Tan's Bit-split AC [18] or accommodate large pattern set like Dharmapurikar's [29], but there are few architectures can both accommodate large pattern set and maintain a high throughput like BFAST. One thing need to be mentioned in this figure is the throughput of the BFAST is depending on the verification rate. We assume the verification rate is low enough (<0.4%) herein so that the scanning module will not be blocked by the verification module, and thus the throughput is full-speeding 9.2Gbps.

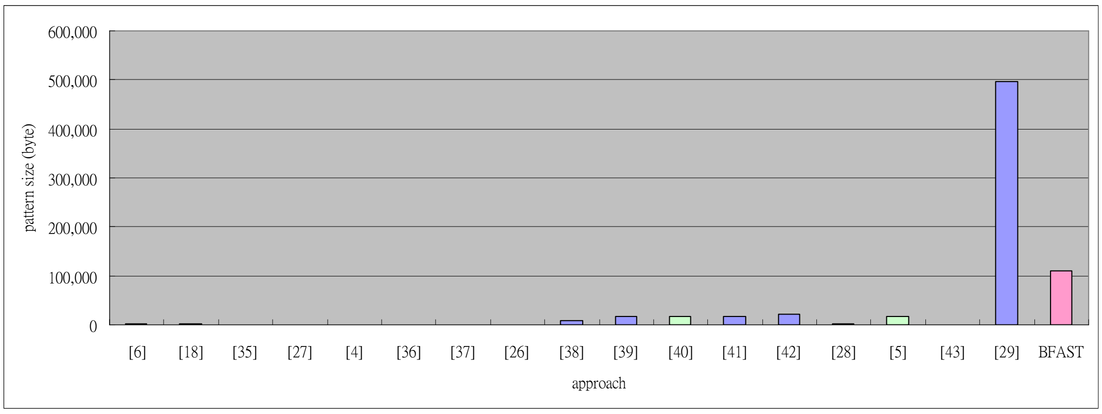

Figure 13. The pattern size comparison of different hardware architecture

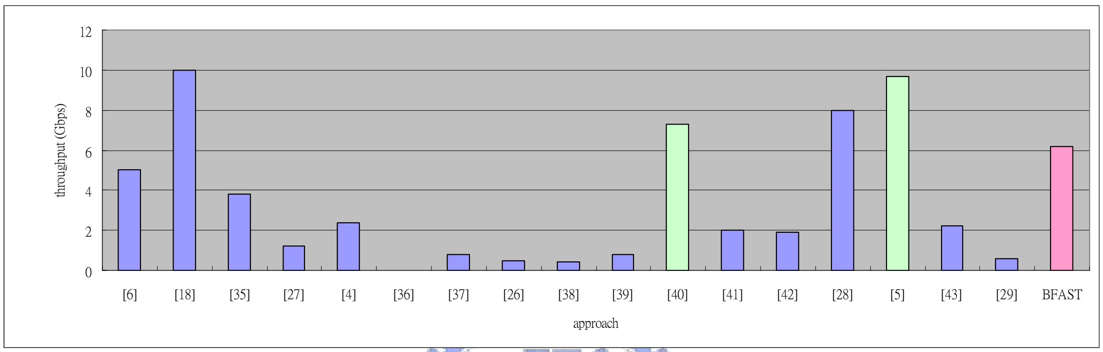

Figure 14. The throughput comparison of different hardware architecture

Besides the large pattern capacity and the high throughput, another advantage of the BFAST is the simple circuit design. Table3 compares with the INOSIDICE and the clarks' design which have more little pattern capacity but higher throughput, Table the circuit size of BFAST is much smaller with storing the patterns in memory instead of flip-flops.

Table 3. Comparisons of circuit sizes of Multi-character decoder NFA, Pre-decoded CAM comparator and BFAST

Type	**Approach**	**Circuit Size (LC)**
Decoder NFA	**Clark's Multi-character decoder NFA [29]**	29,281
Parallel Comparator	**Sourdis et al.'s Pre-decoded CAM Comp. [33]**	64,268
Bloom Filter	**Implicit shift table using Bloom filters**	5,438

Chapter6. Conclusion and future works

This work implement an implicit shift table using Bloom filters to realize a sub-linear time algorithm with hardware. It processes multiple bytes a time based on the theory of the sub-linear time algorithm to increase the throughput up to 9.2Gbps and utilize the efficient memory-usage Bloom filter to accommodate more than 10,000 patterns. The simplicity of the circuit design of this architecture makes this design can be integrated into the Xilinx XC2VP30 SOC platform to become a customized anti-virus chip. After coordinating the packet flow and the other processor communication, it can become a complete security system.

After the implementation, we find that although the performance of this design is good in average case, but it will decrease when the verification rate going higher, i.e. when the virus appearing more often. The slow verification speed will slowdown overall system performance. It can be fixed by utilizing more than one verification engine to balance the speed between verification and scanning. Analysis of the speed differencing in various virus appearing ratio and the different packet lengthes is also interesting in this topic. Although this work implements only a heuristic like Bad-Character in the BM algorithm, the Good-Suffix heuristic is designed with this architecture and illustrated in Appendix too. The improvement of this heuristic is predictable.

References

[1] S. Antonatos, K. G. Anagnostakis and E. P. Markatos, “Generating realistic workloads for network intrusion detection systems,” *ACM Workshop on Software and Performance (WOSP)*, Redwood, CA, Jan. 2004.

[2] Ying-Dar Lin, Chih-Wei Jan, Po-Ching Lin and Yuan-Cheng Lai, “Designing an Integrated Architecture for Network Content Security Gateways,” *IEEE Computer*, June 2006.

[3] Y. H. Cho, S. Navab and W. H. Mangione-Smith, “Specialized hardware for deep network packet filtering,” *Proc. of 12th International Conference on Field Programmable Logic and Applications (FPL)*, La Grand Motte, France, Sept. 2002.

[4] Z. K. Baker and V. K. Prasanna, “Time and area efficient pattern matching on FPGAs,” *Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, Feb. 2004.

[5] I. Sourdis and D. Pnevmatikatos, “Pre-decoded CAMs for efficient and high-speed NIDS Pattern Matching,” *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, April 2004.

[6] M. Aldwairi, T. Conte and P. Franzon, “Configurable string matching hardware for speeding up intrusion detection,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 99-107, Mar. 2005.

[7] I. Sourdis, “Efficient and high-speed FPGA-based string matching for packet inspection,” *MS Thesis, Dept. Elec. Comput. Eng., Tech. Univ. Crete*, Jul. 2004.

[8] A. Aho and M. Corasick, “Efficient string matching: An aid to bibliographic search,” *Comm. of the ACM*, vol. 18, issue 6, pp.333-343, Jun. 1975.

[9] P. C. Lin, Z. X. Li, Y. D. Lin, Y. C. Lai and F. C. Lin, “Profiling and Accelerating String Matching Algorithms in Three Network Content Security Applications,” *IEEE Communications Surveys and Tutorials*, to appear.

[10] G. Navarro and M. Raffinot, Flexible pattern matching in strings, Cambridge Univ. Press, 2002.

[11] M. Norton and D. Roelker, “High performance multi-rule inspection engine,” [Online]. Available: *http://www.snort.org/docs/*.

[12] A. Broder and M. Mitzenmacher, “Network applications of Bloom Filters: A survey,” *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, 2004.

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

[14] R. S. Boyer and J. S. Moore. “A Fast String Searching Algorithm,” *Comm. of the ACM*, vol. 20, issue 10, pp.762-772, Oct. 1977.

[15] ClamAV document, “Creating signatures for ClamAV,” [Online]. Available: *http://www.clamav.net/doc/0.88/signatures.pdf*.

[16] M. Chrochemore and W. Rytter, Jewels of Stringology, World Scientific Publishing, 2003.

[17] Y. Sugawara, M. Inaba and K. Hiraki, “Over 10Gbps String Matching Mechanism for Multi-Stream Packet Scanning Systems,” *Proc. of 14th International Conference on Field Programmable Logic and Applications (FPL)*, Antwerpen, Belgium, Aug. 2004.

[18] L. Tan and T. Sherwood, “A High Throughput String Matching Architecture for Intrusion Detection and Prevention,” *Proc. 32nd Annual International Symposium on Computer Architecture (ISCA)*, Madison, WI, June 2005.

[19] N. Tuck, T. Sherwood, B. Calder and G. Varghese. “Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection,” *Proc. of the IEEE INFOCOM*, Hong Kong, Mar. 2004.

[20] M. Norton, “Optimizing Pattern Matching for Intrusion Detection,” [Online]. Available: *http://www.snort.org/docs/*.

[21] C. J. Coit, S. Staniford and J. McAlerney, “Towards faster string matching for intrusion detection or exceeding the speed of Snort,” *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, Anaheim, CA, June 2001.

[22] M. Fisk and G. Varghese. “Fast Content-Based Packet Handling for Intrusion Detection”. *Tech. Rep. CS2001-0670, UC San Diego*, May 2001.

[23] R. T. Liu, N. F. Huang, C. H. Chen and C. N. Kao, “A fast string-matching algorithm for network processor-based intrusion detection system,” *ACM Trans. Embedded Computing Systems*, vol. 3, no. 3, pp. 614-633, Aug. 2004.

[24] P. C. Lin, Y. D. Lin, Y. C. Lai, T. H. Lee, “A Multiple-String Matching Algorithm with Backward Hashing for Generic Network Content Security”, __.

[25] Xilinx, “Two flows for partial reconfiguration: Module based and difference based,” [Online]. Available: *http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf.*

[26] R. Sidhu, V. K. Prasanna, “Fast regular expression matching using FPGAs,” *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM),* Rohnert Park, CA. Apr. 2001.

[27] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. “Implementation of a Content-Scanning Module for an Internet Firewall,” *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, April 2003.

[28] I. Sourdis and D. Pnevmatikatos. “Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System,” *Proc. of 13th International*

Conference on Field Programmable Logic and Applications (FPL), Lisbon, Portugal, Sep. 2003.

[29] S. Dharmapurikar, P. Krishnamurthy, T. Sproull and J. Lockwood. "Deep packet inspection using parallel bloom filters," *11th Symposium on High Performance Interconnects*, Stanford, CA, Aug. 2003.

[30] B. Bloom. "Space/Time Tradeoffs in Hash Coding with Allowable Errors," *Comm. of the ACM*, vol. 13, issue 7, pp 422-426, July 1970.

[31] R. A. Baeza-Yates and G. H. Gonnet, "A New Approach to Text Searching," *Comm. of ACM*, vol. 35, issue 10, pp.74-82, Oct. 1992.

[32] Z. Galil, "On improving the worst case running time of the Boyer-Moore string searching algorithm," *Comm. of the ACM*, vol. 22, issue 9, pp. 505-508, 1979.

[33] M. Attig, S. Dharmapurikar, and J. Lockwood. "Implementation results of bloom filters for string matching." In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2004.

[34] M.V. Remekrshna, E. Fu and E. Bahcekapili, "A performance study of hashing function for hardware applications," In Proc. of Int. Conf. on Computing and Information, pages 1621-1636,1994

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

[36] G. Tripp. "A Finite-State-Machine Based String Matching System For Intrusion Detection on High-Speed Network." *Proc. of EICAR 2005*, p. 26-40, May 2005.

[37] L. Bu, J. A. Chandy. "A Keyword Match Processor Architecture Using Content Addressable Memory." *Proc. of the 14th ACM Great Lakes Symp. on VLSI*, Apr. 2004.

[38] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proc. 10th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 02),* IEEE CS Press, 2002, pp. 111-120.

[39] C. R. Clark, D. E. Schimmel "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns." Lecture Notes in Computer Science, Vol. 2778, Jan 2003

[40] C. R. Clark, D. E. Schimmel. "Scalable Pattern Matching for High Speed Networks." *Proc. of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, 2004

[41] C. R. Clark, D. E. Schimmel. "A Pattern-Matching Co-Processor for Network Intrusion Detection Systems." *Proc. of IEEE International Conference on Field-Programmable Technology (FPT),* Tokyo, Japan, Dec 2003

[42] Y. H. Cho, W. H. Mangione-Smith, “A Pattern Matching Coprocessor For Network Security.” *Proc. of the 42nd Annual Conference on Design Automation*, California, USA, Jun 2005.

[43] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. “Granidt: Towards Gigabit Rate Network Intrusion Detection Technology.” Lecture Notes in Computer Science, Vol. 2438, Jan 2002.