# 行政院國家科學委員會專題研究計畫　期中進度報告

## 子計畫三:網路內容分類的系統架構：設計、實作與評估 (2/3)

計畫類別： 整合型計畫
計畫編號： NSC94-2213-E-009-037-
執行期間： 94 年 08 月 01 日至 95 年 07 月 31 日
執行單位： 國立交通大學資訊科學學系(所)

計畫主持人： 林盈達

報告類型： 精簡報告
報告附件： 出席國際會議研究心得報告及發表論文
處理方式： 本計畫可公開查詢

中　華　民　國 95 年 6 月 1 日

## 一，計畫

| | |
|---|---|
| 計畫名稱 | 以內容為基礎之網路安全-子計畫三:網路內容分類的系統架構：設計、實作與評估(2/3) |
| 計畫編號 | 94-2213-E-009-037- |
| 主持人 | 林盈達 教授 交通大學資訊科學系 |
| 執行機關 | 交通大學資訊科學系 |
| 執行期限 | 2005.08.01 至 2006.07.31 |

## 二、關鍵詞

本文關鍵詞—次線性時間、Bloom filter、字串比對、硬體

*Keywords—sub-linear time, Bloom filter, string matching, hardware*

## 三、中英文摘要

一般處理器跑字串比對演算法，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走向使用硬體加速器來加速字串比對的運算。我們提出了一個應用 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。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。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。

Because of the intensive computation and memory access on a general-purpose processor, software-based implementations may not meet the high-performance requirements. Instead, adopting hardware to take advantage of the parallelism is a promising solution recently to inspect the packet payload at line rate. This work proposes an innovative memory-based architecture using Bloom filters to realize a sub-linear time algorithm. The two key ideas to adopt the sub-linear time algorithm into the proposed architecture are (1) replacing the slow table lookup in the external memory with simultaneous queries of 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 up to almost 10Gbps when the text using windows executable files and 600 Mbps in the worst case.

## 四、計畫目的

String matching algorithms implemented on general purpose processors are 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 study in [1] 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 [2], 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 [2], [3]. Although they generally have higher performance than a linear time algorithm in software [4], [5], 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.

This work proposes an innovative memory-based architecture using Bloom filters [6] to realize a sub-linear time algorithm enhanced from the Wu-Manber (WM) algorithm [7], 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 [8] 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 [9].

## 五、研究方法及結果

### I. The rationale of the BFAST algorithm

A large shift table in the WM algorithm 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.

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 position of the patterns and storing these groups in separate Bloom filters, we can know a block belongs to the pattern or not and its position by querying these Bloom filters in parallel.

Fig. 1 is a diagram describing how to establish an implicit shift table using Bloom filters. Assume the pattern set is {$P_1$, $P_2$, $P_3$}. After dividing by the position, the Group 0 is {efgh,mnop,vuts}, Group 1 is {defg,lmno,wvut}, etc. 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.

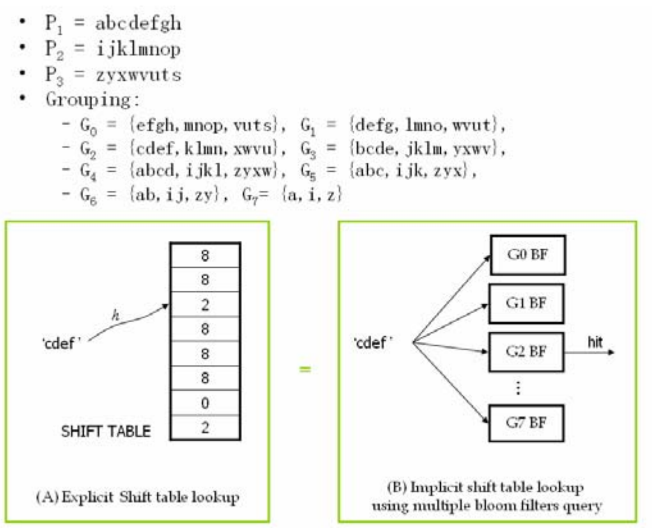
• P1 = abcdefgh
• P2 = ijklmnop
• P3 = zyxwvuts
• Grouping:
- G0 = {efgh, mnop, vuts}, G1 = {defg, lmno, wvut},
- G2 = {cdef, klmn, xwvu}, G3 = {bcde, jklm, yxwv},
- G4 = {abcd, ijkl, zyxw}, G5 = {abc, ijk, zyx},
- G6 = {ab, ij, zy}, G7= {a, i, z}
'cdef'
h
8
8
2
8
8
8
0
2
SHIFT TABLE
(A) Explicit Shift table lookup
=
'cdef'
G0 BF
G1 BF
G2 BF
hit
G7 BF
(B) Implicit shift table lookup using multiple bloom filters query

**Figure 1. 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.**

### II. Additional checking and worst-case handling

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. Therefore, unlike the original WM that verifies the possible match immediately, the BFAST algorithm continues checking the block $B_1$, $B_2$, …, $B_{m-|B|}$ 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 is still a hit, we continue to use the preceding block ***fghi*** to query group 2; otherwise, we claim failure and move on to scan the block ***ijkl*** which is the next block of the one causing the verification, i.e. the block which shift distance is 0. The 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%.

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.

## III. String matching 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. 2 shows the block diagram of the entire architecture. Each component in this architecture is described in following sections.

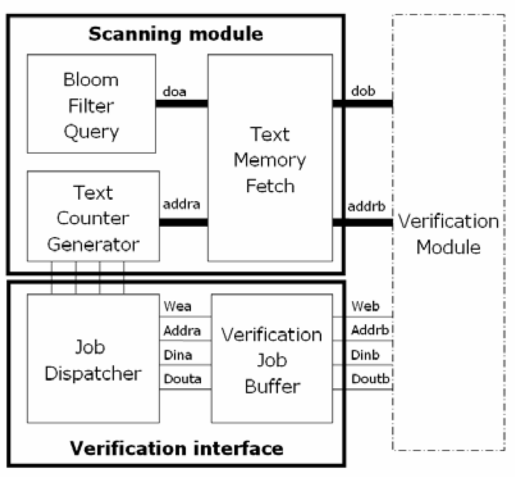

**Figure 2. Overview of the string matching architecture**

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.

A. 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. 3 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.

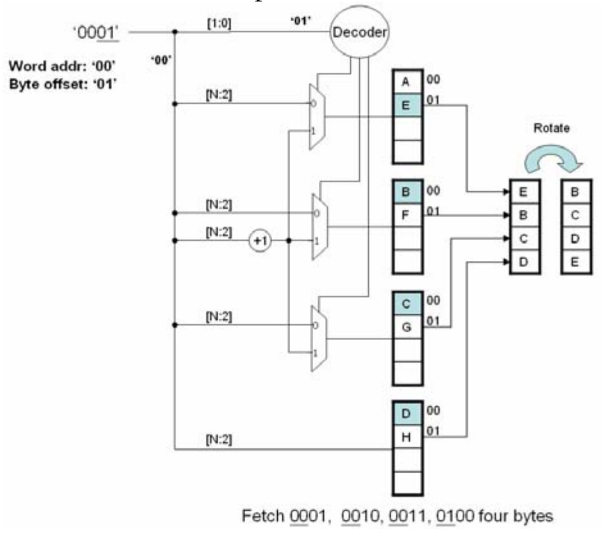

**Figure 3. An example of fetching four bytes in $\underline{00}01$, $\underline{00}10$, $\underline{00}11$ and $\underline{01}00$.**

B. 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. The block diagram of the *BloomFilterQuery* module is presented in Fig. 4.

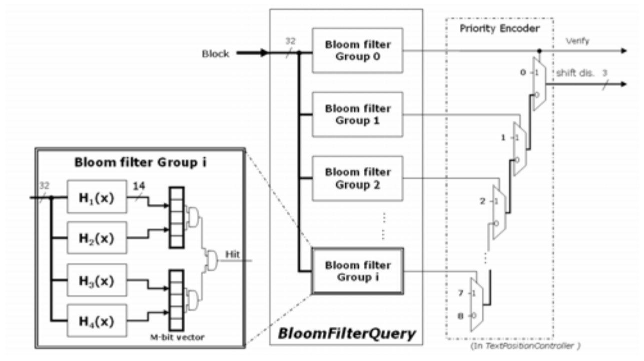

**Figure 4.** ***BloomFilterQuery*** **module architecture**

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 XCV2P30 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 $\left(1 - e^{-\frac{2n}{16384}}\right)^2$, 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.

## C. 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.

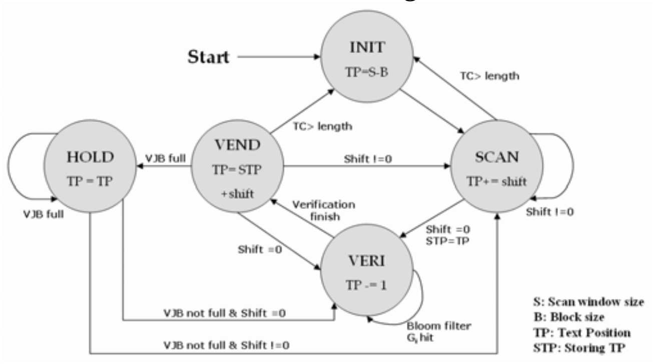

**Figure 5. Text position controller state transition diagram.**

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.

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.

## D. Verification module

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.

## E. Integration with Xilinx Virtex II Pro

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 XCV2P30. The well-tested soft IP supported by Xilinx with this chip can be used to quickly integrat 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.

## IV. Experimental results

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.

### A. 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 $f = \left(1 - e^{-\frac{nk}{m}}\right)^k$, 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. 6. We use both windows executive files and random generated data as scanning text to run simulations. Although the shift distance becomes smaller as the pattern set going larger decreasing the performance, it maintains in 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.

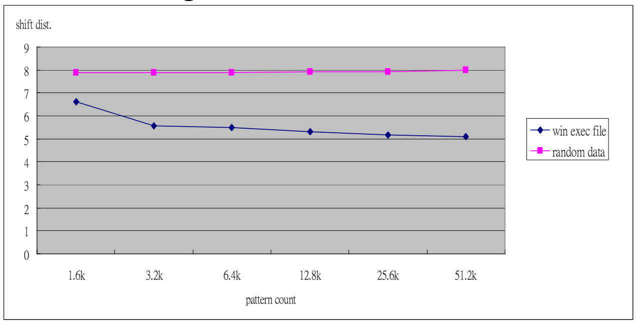

**Figure 6. Shift distance in different pattern counts (read/random).**

The scanning module positive rate and the single Bloom filter positive rate of different pattern count is showed in Fig. 7. The single Bloom filter false positive rate 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 executive 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 going 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%.

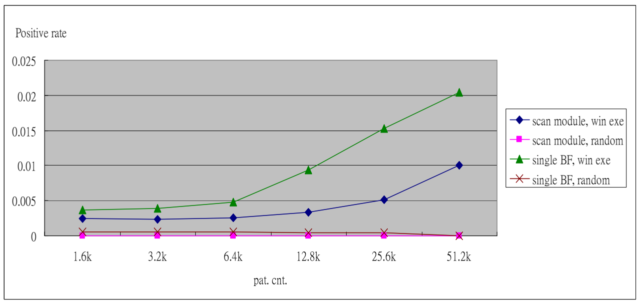

**Figure 7. Scanning module false positive rate in different pattern counts**

### B. 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 1Mb 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 is about 1.4Mb 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.

This architecture implementation takes 16% usage of LUT of XCV2P30 and the system clock is 150 MHz, and the average shift distance is 7.71 bytes. If the scanning module is not blocked by the verification module, 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. The assumption of not being blocked is established in average case.

The worst case that occurs when the text is full of viruses 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 parameter combinations is aimed to be implemented in the future work and not being analyzed in this work. We 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.

C. Comparison with other architectures

Fig. 8 and Fig. 9 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 or accommodate large pattern set like Dharmapurikar's 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.

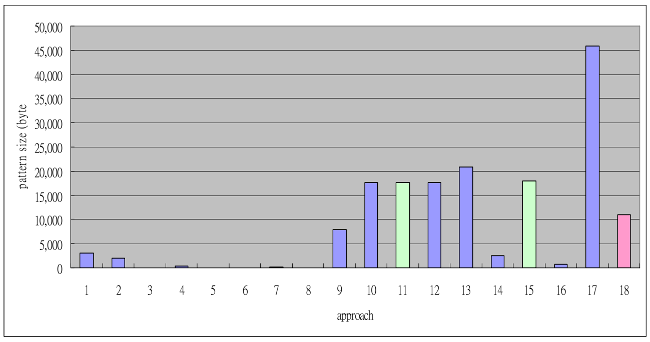

**Figure 8. The pattern size comparison of different hardware architectures.**

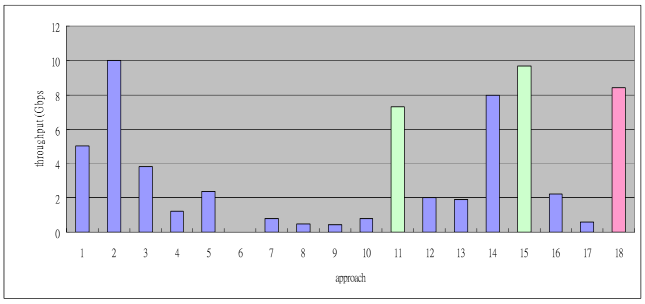

**Figure 9. The throughput comparison of different hardware architectures.**

**V. Conclusion**

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 performance and utilize the efficient memory-usage Bloom filter to increase the pattern capacity. 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 lengths is also interesting in this topic.

## References

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

[2] 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.

[3] 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.

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

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

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

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

[8] 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.

[9] ClamAV document, "Creating signatures for ClamAV," [Online]. Available: *http://www.clamav.net/doc/*.