Scalable Automaton Matching for High-speed Deep Content Inspection

Ying-Dar Lin, Kuo-Kun Tseng and Chen-Chou Hung
National Chiao Tung University, Taiwan
{ydlin@cis, kktseng@cis and bry@cis}
.nctu.edu.tw

Yuan-Cheng Lai
National Taiwan University
of Science and Technology, Taiwan
laiyc@cs.ntust.edu.tw

***Abstract*—String matching plays a central role in content inspection applications such as intrusion detection, anti-virus, anti-spam and Web filtering. Because they are computation and memory intensive, software matching algorithms are insufficient in meeting the high-speed performance. Thus, offloading packet content inspection to dedicated hardware seems inevitable. This paper presents a scalable automaton matching (SAM) design, which uses Aho-Corasick (AC) algorithm with two parallel acceleration techniques, root-indexing and pre-hashing. The root-indexing can match multiple bytes in one single matching, and the pre-hashing can be used to avoid bitmap AC matching, which is a cycle-consuming operation. In the implementation of the Xilinx Vertex4P FPGA platform, the proposed hardware architecture can achieve almost 10.7 Gbps and support the largest pattern set, which is 7.65 times faster than the original bitmap AC in the average case. Further, SAM is feasible for either internal or external memory architecture. The internal memory architecture provides high performance, and the external memory architecture provides high scalability of patterns.**

***Index Terms*—Coprocessor, String matching, Hashing, Finite automata, Content filtering.**

I. INTRODUCTION

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

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

In this paper, we propose a scalable automaton matching (SAM) which is based on the Aho-Corasick (AC) algorithm with external memory architecture. AC is a common algorithm with the following key features. First, it has a linear time performance in the worst case. Second, it is robust for large and long patterns. Third, it can perform multi-pattern matches. However, the most critical defect of AC algorithm is its large memory usage. Another AC-based algorithm, bitmap AC, improves memory utilization by using a 256-bit bitmap to replace the 256 word-size pointers of each state in AC. Therefore, bitmap AC is the alternative which we adopted in this work. We also developed two acceleration techniques to make our architecture have a sub-linear matching time. The first technique is root-indexing, which comes from the observation of AC's high frequency root-visiting behavior. The second technique is pre-hashing, which comes from the observation of time-consuming operation in bitmap AC, which also has a high cost on the x86-platform. Thus, to reduce this kind of operation, pre-hashing can test quickly to avoid bitmap AC matching. For scalability, our architecture uses either internal or external memories to store the whole pattern database of SNORT or even ClamAV.

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

II. RELATED WORKS

A. Selecting Matching Algorithm for Content Filtering

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

Although the complex patterns, such as class, wildcard, regular expression and case sensitive patterns might increase the expression power of the patterns and has been used in some applications, they can be converted to be composed of multiple simple patterns [8], they are optional for matching algorithms.

Current existing on-line string matching algorithms for content filtering can be classified into four categories, namely, dynamic programming, bit parallel, filtering, and automaton algorithms. In the summary, dynamic programming [3] and bit

21st International Conference on
Advanced Information Networking and Applications Workshops (AINAW'07)
0-7695-2847-3/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

parallel [4] algorithms are inappropriate for variable-length and multiple patterns, and the filtering algorithms [5] have poor worst-case time complexity O(nm), where n and m are the length of the text and patterns, respectively. Only the automaton based algorithms such as Aho-Corasick (AC) [6] support variable-length and multiple patterns, and also have the deterministic worst-case time complexity O(n). Therefore, the automaton based algorithm is a better choice for the content filtering system, and is selected as a base to develop new approaches.

B. AC Related Algorithms

AC works by constructing a state machine from the patterns to be matched, and is composed of three component functions. The first is the *Goto* function, which is used to traverse from one node to the other node. The second is the *Output* function, which outputs the match pattern for the current state. The third is the *Failure* function, which is traversed when there is no next state.

AC is a typical deterministic finite automaton (DFA) based on string matching. However, there are several variations of it. Bitmap AC [7] uses bitmap compression to reduce the storage of AC states. AC_BM [8] is a combination of the AC and Boyer Moore (BM) algorithms, and aims to improve the conventional AC from O(n) to the sub-linear time complexity. AC_BDM [9] combines AC with backward DAWG matching (BDM), which improves the average-case time complexity of the conventional AC. Bit-split AC [10] splits the width of the input text into a smaller bit width to reduce memory usage and the number of comparisons when selecting next states. Since AC_BM has the worst-case time complexity O(nm) and overhead for switching between AC and BDM, and since bit-split AC requires large match vector for each bit-split state, they are impractical for large patterns. Thus, a scalable bitmap AC with space efficiency is more preferable for our embodiment.

Bitmap AC, is a compromise between table and link list approaches. It maintains a 256-bit bitmap for each state to indicate whether a traverse with a given character is valid or invalid, which requires traversing along the failure pointer path.

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

C. Hardware-based String Matching

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

For DFA based hardware, there are three common designs in recent string matching hardware: Aho-Corasick (AC) based hardware [11, 12] Regular Expression (RE) based hardware [13, 14] and Knuth-Morris-Pratt (KMP) [15, 16, 17] based hardware. To save more states, KMP and AC are simplified from RE DFA by disabling their regular expression patterns. Each AC DFA supports multiple simple patterns, and each KMP DFA only support single simple patterns. Thus, many KMP DFAs use duplicated hardware to support multiple patterns.

As for NFA based hardware, there are two variations: comparator NFA [18, 19], which uses the distributed comparators, and decoder NFA [20], which the uses the character decoder (shared decoder) to build their NFA circuits. In our comparison, it seems that the comparator NFA is more scalable and has a higher performance compared to the decoder NFA.

Other existing non-DFA based hardware are the parallel comparator [21, 22, 23], Bloom filter [24], systolic array [25] and parallel-and-pipeline [26] hardware in our classification. Parallel comparator based hardware improves the performance of brute force algorithm by exploiting architecture parallelism and pipelining. Bloom filter based hardware uses multiple hashing keys for quick approximate matching. Using systolic array implementing dynamic programming for string matching is only proper for short patterns and text, since the circuit size is proportional to the length of the patterns and text. Parallel and pipeline hardware uses naïve string matching and only accelerates processing time by increasing the hardware circuits. Like the systolic array, this approach also has the drawback of only being suited for short-length patterns.

III. Architecture and Techniques

A. Architecture

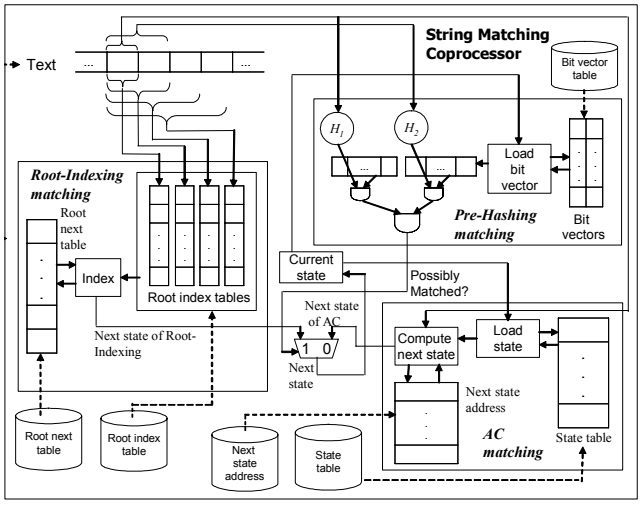

Fig. 1. Architecture of the string matching coprocessor

IEEE COMPUTER SOCIETY

A preferable searching architecture is suggested in Fig. 1, where a string matching coprocessor performs three independent matching units in parallel. Therefore, the control logic coordinates pre-hashing, root-indexing and bitmap AC matching for parallel processing. Also each matching function has its individual memory interface to access its pre-processing data. Since the design methodologies of System On a Chip (SOC) are popular and have matured in recent times, this specific component is quite feasible for use in modern IC technology.

In the SAM coprocessor, the three units can read the text in different lengths and perform their matching concurrently. This example processes a one-byte substring for AC matching, a two-byte substring for pre-hashing matching, and a four-byte substring for root-indexing matching in a single matching iteration. The root-indexing and bitmap AC are used to locate the next states, and the pre-hashing matching is used to decide on which next state is to be used in the next matching iteration.

B. Root-indexing Matching

In the AC tree, most of failure links point to the root state–that is, it will always go back to the root state when there is no any next state for a given character. Thus, it is efficient to apply root-indexing in the root state, where it can match multiple characters simultaneously. In Fig. 2, root-indexing comprises *k* index tables *IDX[1…k]* and a root next table *NEXT*, where *k* denotes the maximum length of root-indexing matching in the same time. Each entry of *IDX* stores a partial address for locating the next state in *NEXT*.

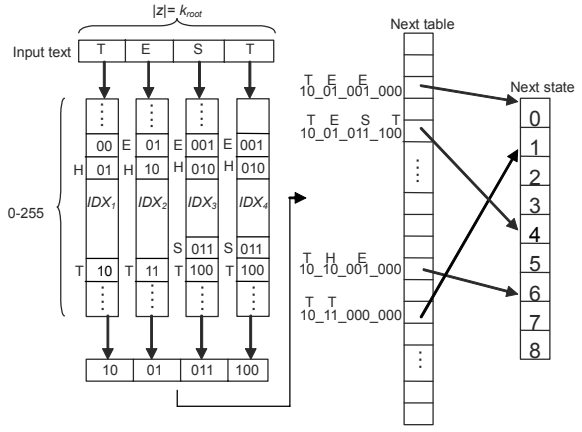

Fig. 2. Root-indexing architecture and example for the input text “TEST” with the patterns “TEST”, “THE” and “HE”.

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

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

C. Pre-hashing Matching

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

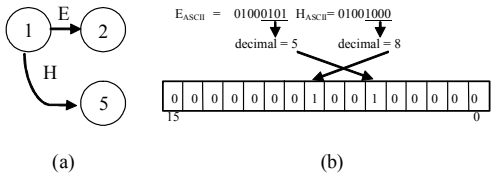

Fig. 3. (a) AC tree of state 1 for building the bit vector. (b) example of building the bit vector for state 1 in the preprocessing phase.

When suffixes are obtained, the pre-hashing algorithm hashes suffixes into bit vectors. This procedure of building the bit vectors for state 1 is illustrated in Fig. 3 (a). In Fig. 3 (b), the mask of the rightmost four bits of the characters and the transformation from binary to one-hot representation are used as the hash function in our design. However, better mask position is adjustable for a lower false positive, according to the characteristics of the patterns.

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

IV. Implementation

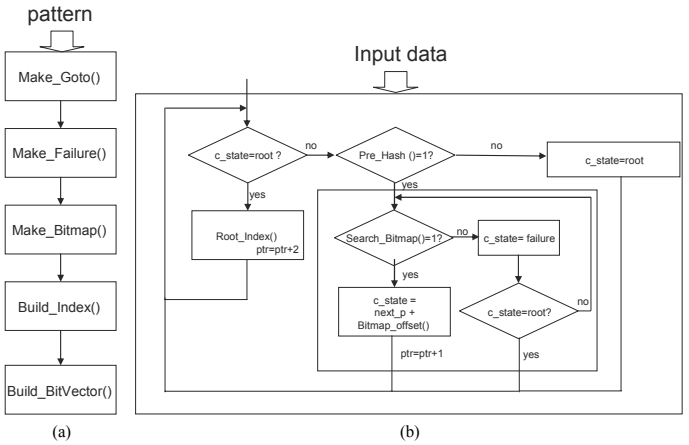

Fig. 4. (a) The pre-processing procedure. (b) The flow of C simulation model.

The pre-processing procedure generates essential data structures for the proposed hardware, as shown in Fig.

IEEE COMPUTER SOCIETY

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

After the pre-processing procedure is finished, the simulation of the proposed SAM can perform matching according to the flow in Fig. 4 (b). For each matching iteration, the first current state is checked. When the current state is in the root state, the *Root-Index()* matching is performed, otherwise *Pre-Hash()* is performed.

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

The proposed architecture is a highly parallel design where all modules are working at the same time, and this architecture is also flexible for either internal or external memory-based platforms. We use the Xilinx Vertex4P FPGA as our development system, with Xilinx EDK and Synplicity's SynplifyPro as the basic development tools. The EDK can generate the software and the bit stream file for our system design. The software includes the mapping address define files and the drivers of all peripherals needed for building the complete RTOS image. The RTL code design for string matching hardware, ModelSim and Debussy, were the simulator and debugger tools we used, respectively.

V. EVALUATION

A. Formal Analysis

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

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

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

However, in the hardware, the pre-hashing, root-indexing and AC can be performed in parallel, and the computation of the next states in these three units are independent. Thus, the average time can be reduced to

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

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

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

The probability P_{root} is calculated by

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

where P_{root} is computed by summing the dependent probabilities of a true non-hit $P_{tn\_j'}$, which is the dependent probability of a true non-hit for length j. Because if the $(j-1)$th pre-hashing is not matched, then the j th pre-hashing function cannot be matched either. Therefore, $P_{tn\_j'}$ is determined from the independent probability of a true non-hit P_{tn_j}, which can be obtained from (2). Therefore, $P_{tn\_1'}$ is the first pre-hashing function, and can be obtained by

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

where P_{tn_1} is the first independent probability of a true non-hit and TH_1 is $Threshold_1$ rate for suffixes of length one. The subsequent $P_{tn\_j'}$ for length j can be computed by

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

When the shorter suffix indicates a true non-hit, the longer suffix definitely outputs a true non-hit too. Therefore, $P_{tn\_j'}$ is computed by subtracting previous summing probability $\sum_{y=1}^{j-1} P_{tn\_y'}$, and multiplying corresponding P_{tn_j}.

Our approach intends to improve the probability of a true non-hit by ascertaining the non-matching suffixes. Thus, using one hashing function for each bit vector is sufficient and can significantly reduce hardware cost and latency. The probability of a true non-hit P_{tn} is referred from [13] as

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

where $|\beta|$ is the number of suffixes, and M is the size of the bit vector. In our observation, a short length of suffixes can also achieve acceptable P_{root} whose value is larger than 0.4. Therefore, setting the maximum suffix length $k_{pre-hash}$ to 2 is sufficient. For example, when P_{tn} is set to 0.6 and $k_{pre-hash}$ is set to 2, P_{root} is equal to 0.84.

IEEE COMPUTER SOCIETY

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

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

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

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

The original space requirement of bitmap AC, $Size_{AC}$, is mainly dominated by the state table, which is equal to the number of states $|S|$ multiplied by the state size $Size_{state}$,

$$Size_{AC} = |S| \times Size_{state}. \quad (10)$$

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

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

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

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

$Size_{root}$, which includes all root-indexing tables and the root next table. The size of all root-indexing table is 256, multiplied by k_{root}, and the root next table is the number of next state addresses, multiplied by the state address size $Size_{state\_address}$. The number of root next state addresses is the cross product of the numbers of appearing alphabets in the index tables IDX_j and one zero entry. $Size_{root}$ is formulated as

$$Size_{root} = 256 \times k_{root} + \prod_{j=1}^{k_{root}} (|IDX_j| + 1) \times Size_{state\_address}. \quad (13)$$

B. Simulation Analysis

This simulation analysis can determine the performance of our simulation software. In our analysis, the test contents are execution files in Linux and Windows, as well as normal text files. The 32-bit bit vector and 1,000 virus patterns are used to evaluate the proportion of root-indexing matching and bitmap AC matching.

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

After analyzing these two key factors, the non-hit rate for different sizes of the bit vector and the number of patterns in the three different data types are shown in Fig. 5. As the pattern set increases, the 32-bit bit vector has better relative improvement than 16-bit bit vector. In addition to the hit rate, the false positive rate of pre-hashing matching is also affected by the size of the bit vector. The false positive will lead to a little penalty in the clock cycles in the internal SRAM architecture, as well as great penalty in the bus contention in the external DRAM architecture.

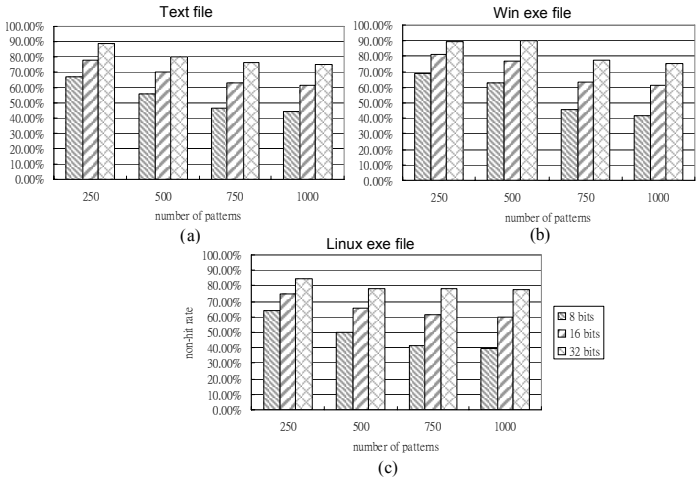

Fig. 5. The non-hit rate of 8-bit, 16-bit and 32-bit bit vectors for (a) text files, (b) Windows execution files, (c) Linux execution files.

C. Hardware Implementation and Comparison

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

The operating frequency of the synthesis result for our internal SRAM architecture is 350 MHz which is reported by SynpilifyPro. The root-indexing module takes 2 clock cycles to index a mapping state. The throughput in the average case, depending on the average proportion of the root-indexing matching and the bitmap AC matching, can be estimated at 5.37 Gbps. For the worst case, all bytes are matched in the text buffer. The throughput is 1.56 Gbps. It is obvious that the average case has very high performance, which is very close to that in the best case, and also has moderate performance in the worst case. This result demonstrates that our pre-hashing and root-indexing techniques are robust for high-performance content filtering applications.

IEEE COMPUTER SOCIETY

Since many previous matching hardware [11, 12, 14, 24, etc.,] employed duplicated hardware for parallel processing, our two engine architecture should be fair in this comparison. Our optimally utilizing dual port block RAM of Xilinx FPGA not only doubles the performance, it also increases no extra block RAM. The results demonstrate that our design has throughput at 10.7 Gbps and a support pattern of 21,563 bytes.

We compare and analyze about 12 major hardware from recent related works, as shown in Fig. 6. The common goals for this kind of hardware are to pursue a higher throughput and a larger pattern size, which are the major evaluated factors in this comparison. Pattern sizes are used for measuring scalability with unit in byte, and the throughput factor is used for measuring performance with unit in giga bit per second (Gbps).

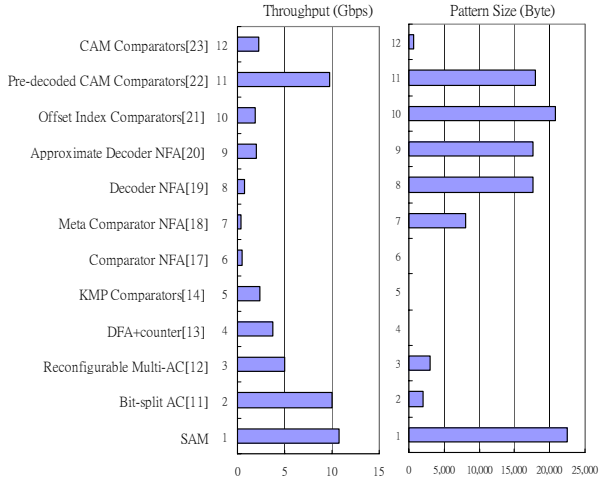

Fig. 6. SAM comparison with the other string matching hardware.

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

VI. Conclusion

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

The proposed internal SRAM architecture is implemented on the Xilinx Virtex4P FPGA-based platform. The string matching function of the target application ClamAV is also modified to set up the string matching engine. We tuned the hardware design according to the analysis results of our software simulation, and also built a complete system solution for content filtering applications such as IDS, URL blocking and ClamAV.

References

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

[2] G. Navarro and M. Ranot, "Flexible Pattern Matching in Strings," *Cambridge University Press*, 2002.

[3] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, 33(1):31-88. 2001.

[4] S. Wu and U. Manber, "Fast Text Searching Allowing Errors," *Communication of the ACM*, 35:83-91.

[5] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, 20, 10, 762–772.

[6] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, pp.333–340.

[7] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *IEEE Infocom*, Hong Kong, China, 2004.

[8] C. Coit, S. Staniford and J. Mcalerney, "Towards Faster String Matching for Intrusion Detection," *DARPA Information Survivability Conference and Exhibition*, pp. 367-373, 2002.

[9] N. Desai, "Increasing performance in high speed NIDS," *http://www.snort.org/*.

[10] M. Raffinot, "On the Multi Backward Dawg Matching Algorithm (MultiBDM)," *Workshop on String Processing*, Carleton U. Press, 1997.

[11] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection And Prevention," *ISCA*, 2005.

[12] M. Aldwairi, T. Conte and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," *ACM CAN*, 2005.

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

[14] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," *IEEE FCCM*, 2003.

[15] Z. K. Baker and V. K. Prasanna, "Time And Area Efficient Pattern Matching on FPGAs," *ACM/SIGDA FPGA*, California, USA, Feb. 2004.

[16] G. Tripp, "A Finite-State-Machine Based String Matching System for Intrusion Detection on High-Speed Network," *EICAR*, May 2005.

[17] L. Bu and J. A. Chandy, "A Keyword Match Processor Architecture Using Content Addressable Memory," *ACM VLSI*, April 26-28, 2004.

[18] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching using FPGAs," *IEEE FCCM*, April 2001.

[19] R. Franklin, D. Carver and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *IEEE FCCM*, Napa, CA, Apr. 2002.

[20] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," *IEEE FCCM*, 2004.

[21] Y. H. Cho and W. H. Mangione, "A Pattern Matching Coprocessor for Network Security," *ACM/IEEE DAC*, California, USA, Jun. 2005.

[22] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," *IEEE FCCM*, 2004.

[23] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," *LNCS*, Volume 2438, Jan. 2002.

[24] S. Dharmapurikar and P. Krishnamurthy, T. S. Sproull and J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro*, Vol. 24, No. 1, Jan. 2004.

[25] H. M. Blüthgen, T. Noll and R. Aachen, "A Programmable Processor For Approximate String Matching With High Throughput Rate," *IEEE ASAP*, 2000.

[26] J. H. Park and K. M. George, "Parallel String Matching Algorithms based on Dataflow," *HICSS*, Hawaii, 1999.

IEEE COMPUTER SOCIETY