A Parallel Automaton String Matching with Pre-Hashing and Root-Indexing Techniques for Content Filtering Coprocessor

Kuo-Kun Tseng, Ying-Dar Lin and Tsern-Huei Lee
National Chiao Tung University, Taiwan
{kktseng@cis, ydlin@cis and thlee@atm.cm}.nctu.edu.tw

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

Abstract

We propose a new Parallel Automaton string matching approach and its hardware architecture for content filtering coprocessor. This new approach can improve the average matching time of the Parallel Automaton with Pre-Hashing and Root-Indexing techniques. The Pre-Hashing technique uses a hashing function to verify quickly the text against the partial patterns in the Automaton, and the Root-Indexing technique matches multiple bytes for the root state in one single matching. A popular Automaton algorithm, Aho-Corasick (AC) is chosen to be implemented by adding the two techniques; we employ these two techniques in a memory efficient version of AC namely Bitmap AC. For the average-case time, our approach improves Bitmap AC by 494% and 224% speedup for URL and Virus patterns, respectively. Since Pre-Hashing and Root-Indexing techniques can be concurrently executed with Bitmap AC in the hardware, our proposed approach has the same worst-case time as Bitmap AC.

1. Introduction

In recent years, deeper and more complicated content filtering is required for internet security applications such as intrusion detection, keyword blocking, anti-virus and anti-spam. In a content filtering system, the string matching usually occupies a large workload. Therefore, it is necessary to design an appropriate string matching accelerator to reduce the workload in such a system.

For understanding the functional requirements of string matching algorithms, we surveyed the real patterns from the software of Open Source including Snort [1] for intrusion detection, ClamAV [2] for anti-virus, SpamAssassin [3] for anti-spam, and SquidGuard [4] and DansGuardian [5] for Web blocking. It is obvious that matching long and multiple patterns in real time is necessary for all content filtering systems.

Many existing real time string matching algorithms are also reviewed and classified into four categories, namely, Dynamic Programming, Bit Parallel, Filtering, and Automaton algorithms. The Dynamic Programming [6] and Bit Parallel [7] are inappropriate for long and multiple patterns, and the Filtering algorithms [8] have poor worst-case time complexity O(nm), where n and m are the length of text and patterns, respectively. Only the Automaton algorithm, e.g., Aho-Corasick (AC) [9], supports long and multiple patterns, and has the worst-case time complexity O(n). Therefore, the Automaton algorithm is a better choice for content filtering system, and selected as a base to develop our new approaches.

In this paper we employ two new techniques in Bitmap AC [9]; a variant of AC. AC is a space efficient Deterministic Finite Automata (DFA) through the failure links to reduce the number of next states. Bitmap AC further uses bitmap compression to reduce the storage of every state. However, it has the constant average-case time complexity O(n), not good enough for the high speed processing. Thus in this paper, Pre-Hashing and Root-Indexing techniques are added into Bitmap AC to speedup its processing time. The Pre-Hashing approach is a quick testing for avoiding the Bitmap AC matching, and the Root-Indexing approach is a parallel technique for matching multiple bytes simultaneously. In addition to the new proposed algorithm, the corresponding hardware architecture is developed as well. For measuring the performance, the space and time complexities are formally analyzed with real patterns. The results demonstrate that our proposed approach significantly outperforms Bitmap AC.

The reset of this paper is organized as follows: Section 2 includes the surveys of related AC algorithms and existing string matching hardware. Section 3 describes our idea, algorithm, detailed design of Pre-Hashing and Root-Indexing, and system architecture. The formal analysis and real patterns analysis are given in Section 4 to evaluate the performance of our proposed algorithm. Finally, we draw conclusion in Section 5.

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

2. Background

In this section, we survey previous related literatures for AC and string matching hardware, and then describe AC, bitmap AC and Bloom Filter algorithms for string matching.

The previous AC related algorithms are summarized in the followings.

1) AC [9] is an algorithm for processing multiple patterns. It searches the patterns in text by traversing an automaton. AC has good worst-case time complexity in O(n), but poor average-case time complexity in O(n).
2) AC_BM [10][11] combines the AC and Boyer Moore (BM) algorithm, and intends to improve the conventional AC from O(n) to the sub-linear time complexity with BM approach. However, the main drawback for AC_BM is that it has the worst-case time complexity O(nm).
3) Vectorized AC [12] implements AC in a vector processor and performs the string matching in parallel. This algorithm requires preprocessing the text, and thus is not suitable for real time matching.
4) Bitmap AC [13] uses the bitmap to locate the links of the subsequent next states, and thus improves the space usage from the conventional AC. However, the time of loading bitmap and calculating bitmap to locate the next state slow down the matching performance.
5) AC_BDM [14] combines AC with Backward Dawg Matching (BDM). This algorithm can also improve the average-case time complexity of the conventional AC algorithm, but it requires the double space and processing overhead for switching between AC and BDM.

We also investigated previous hardware technologies for string matching and summarized their pros and cons as follows.

1) Systolic array hardware [15] [16] implements Dynamic Programming for string matching. As we mentioned, the Dynamic Programming matching is only proper for short patterns and a short text, since the circuit size is proportional to the length of pattern and text.
2) Parallel and pipeline hardware [17] uses the naïve string matching and only accelerates processing time by increasing hardware circuit. Similar to systolic array, this approach also has the drawback of only suitable for short length patterns.
3) Reconfigurable hardware [18] [19] directly transforms the patterns into FPGA and is able to perform matching for the regular expressions. However, its main shortcoming is that FPGA is slower and more expensive than ASIC, causing the lack of competition in the commercial market.
4) Bloom Filter String Matching (BFSM) [20] [21] hardware uses Bloom Filter to accelerate the average-case time complexity for the exact matching algorithms. However, BFSM builds a single big bit vector for all patterns, making it infeasible.

The works most related to our approach is BFSM algorithms. The main philosophy of BFSM is that it uses multiple hashing functions to reduce the probability of false positive, which is the false match, but AND function reports a positive value. When BFSM chooses k independent hashing functions to hash N patterns into a vector with size M , where the probability of false positive P_{fp} is obtained as

$$P_{fp} = \left(1 - \left(1 - \frac{1}{M}\right)^{Nk}\right)^{k}, \qquad (1)$$

with referring to [20].

3. Algorithm and Architecture Design

We describe Pre-Hashing and Root-Indexing matching algorithms in these sections.

3.1. Pre-Hashing Matching

The Pre-Hashing method can test quickly the multiple partial patterns of the current state against the compared substring of text to avoid some slow AC matching. The AC matching can be skipped if True Negative is indicated in the Pre-Hashing matching. True Negative is the condition that the compared substring of text is absent in Pre-hashing vector of the Suffixes of the current state.

The Pre-Hashing approach can be described in Figure 1. β_i is a set of Suffixes for state S_i within the length $k_{pre-hash}$, that β_i also includes the failure links in the AC tree. When Suffixes are obtained, the Pre-Hashing algorithm hashes Suffixes into bit vectors. This procedure of building the bit vectors is illustrated in Figure 1 (a).

In the searching phase, Pre-Hashing is performed to match quickly for current state in the AC tree. Figure 1 (b) shows this searching process, the matching unit loads the current bit vectors V_c, then perform hashing operations to test whether each $w[1..j]$ is True Negative or not.

The Pre-Hashing idea is motivated by BFSM. However, there are two main differences between BFSM and our Pre-Hashing as follows.

1063-6862/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

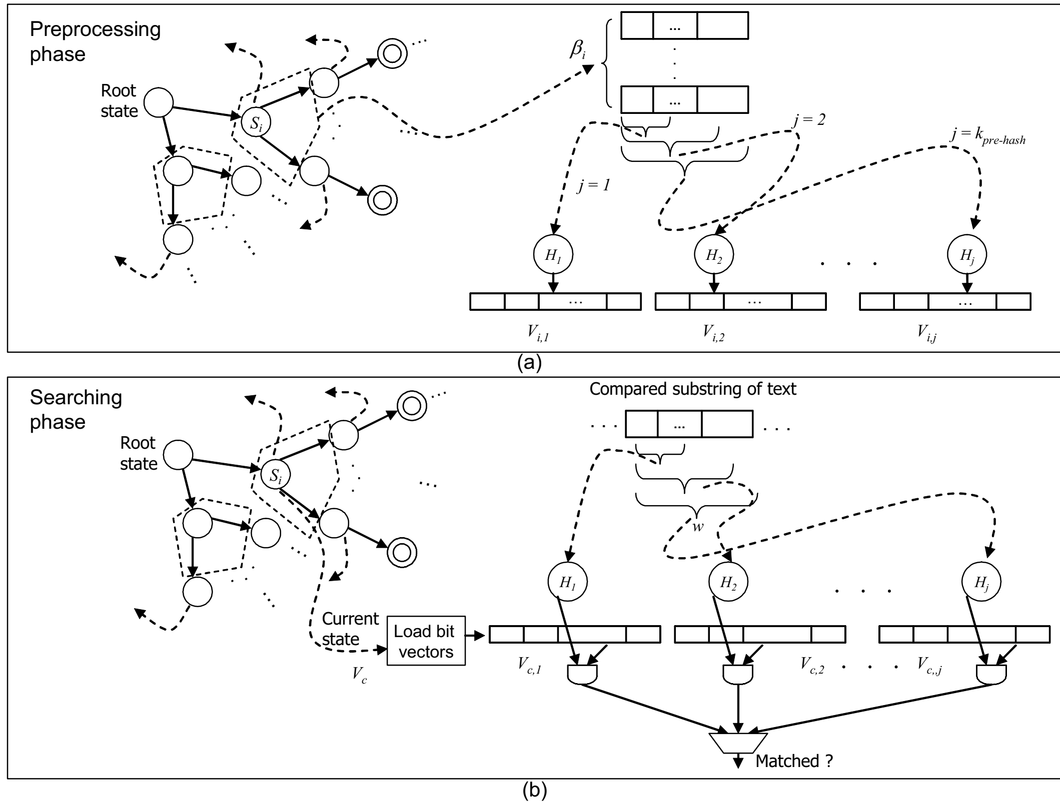

Figure 1. Pre-Hashing Matching for state S_i (a) Building the bit vector in the preprocessing phase (b) Load bit vector and compare text in the searching phase.

1) Since BFSM requires multiple Bloom Filters and builds the bit vector of each Bloom Filter from all patterns (which will need the large and multiple access memory for constructing bit vectors), it makes implementing bit vector impractical by using neither register nor SRAM. However our approach builds the bit vectors from the Suffixes of the state S_i only, the number of Suffixes is quite small, it makes implementing the bit vectors more feasible.

2) Because more hashing functions will set more one bit to one in bit vector, BFSM employs multiple hashing functions can reduce the probability of False Positive only, and cannot reduce the number of the exact matching. Our approach intends to improve the probability of True Negative for finding out the un-matching Suffixes. Thus using one hashing function is sufficient and that can significantly reduce the hardware cost and latency. The probability of True Negative P_{tn} is adapted from (1) as

$$P_{tn} = \left(1 - \frac{1}{M}\right)^{|\beta|}, \tag{2}$$

where $|\beta|$ is the number of Suffixes, and M is the size of bit vector.

3.2. Root-Indexing Matching

When the Pre-Hashing result is True Negative, the matching transition will return to the root state. Since most bytes of the text will visit the root state, the parallel Root-Indexing technique is worth to be used. Root-Indexing can process multiple characters of the text at the same time. In fact, the Root-Indexing matching is a compressed technique for parallel matching in the Automaton. In Figure 2 (a), Root-Indexing comprises k_{root} root index tables $IDX_{[1..k_{root}]}$ and a root next table $NEXT$, where k_{root} denotes the length of Root-Indexing matching. Each IDX stores the ordering number of the appearing letters for the corresponding byte in Prefixes, and each IDX has 256 slots for every binary

1063-6862/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

representation of the characters. $NEXT$ stores the next state addresses of the states, within the length k_{root} from root state S_0. The number of next addresses is equal to $\prod_{j=1}^{k_{root}} |IDX_j|$, that is the product of the size of each root index table.

In the example of Root-Indexing, we can obtain next state address NA for the patterns, "TEST", "THE", "HE" in parallel, as shown in Figure 2 (b). In the matching phase, 10_01_01_0, 10_01_10_1, 10_10_01_0 and 10_00_00_0 are NA to locate next states 2, 4, 6, 1 for input text z as "TEE", "TEST", "THE" and TT", respectively. For instance, Root-Indexing can lookup the $IDX_1[T] \circ IDX_2[E] \circ IDX_3[S] \circ IDX_4[T]$ to get the 10_01_10_1 to locate the text "TEST". Note that the zero value of IDX_j is mapped into the slot of the symbol (~), which is the termination symbol for the length of z is shorter than k_{root}. For example, NA of the text "TEE" is not $IDX_1[T] \circ IDX_2[E] \circ IDX_3[E]$, but $IDX_1[T] \circ IDX_2[E] \circ IDX_3[E] \circ IDX_4[\sim]$, because length of "TEE" is less than four. The other special case is the text "TT", because the second "T" is in IDX_2 as the NA of $IDX_1[T] \circ IDX_2[T]$ cannot be indexed, thus NA for "TT" will be indexed by $IDX_1[T] \circ IDX_2[\sim]$ to locate the next state.

4. Analysis

To evaluate the performance of our proposed algorithm, the formal formula, the real pattern analysis are given below.

4.1. Formal Analysis

Since Pre-Hashing, Root-Indexing and AC can be performed in parallel; the average time can be reduced as

$$T_{avg_time} = \frac{P_{root} \times T_{root} + (1 - P_{root}) \times T_{AC}}{(k_{root} \times P_{root}) + (1 - P_{root})}. \quad (3)$$

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.

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 True Negative $P_{tn_j'}$, which is the dependent probability of True Negative

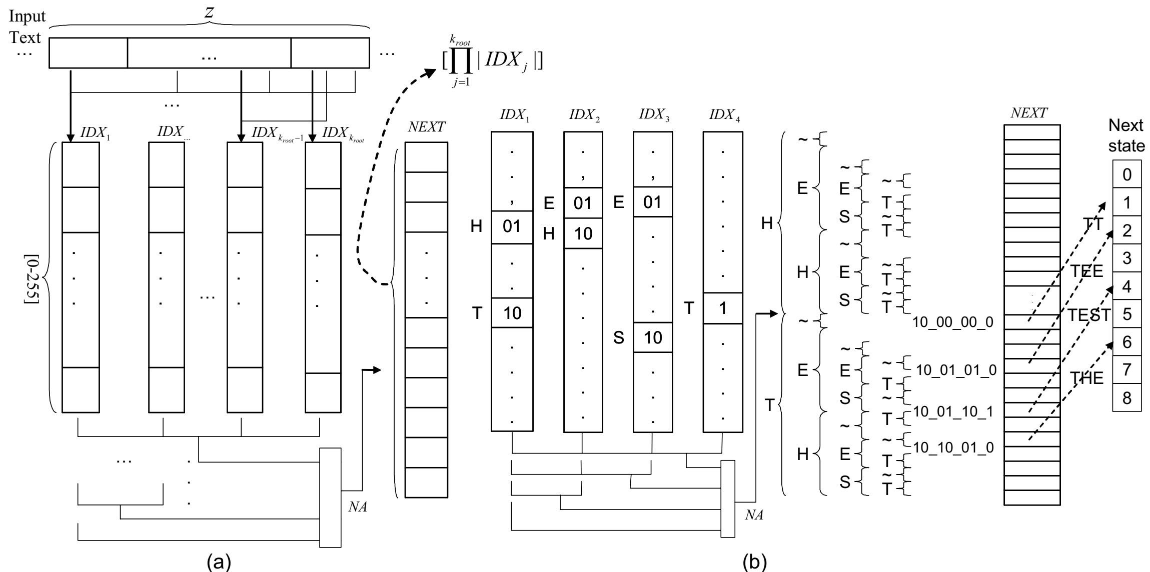

Figure 2. (a) Root-Indexing Architecture, (b) A Root-Indexing example for matching the texts "THE", "TEST", "TEE" and "TT" with the patterns "TEST", "THE" and "HE".

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

for length j. Because the ($j-1$)th Pre-Hashing is not matched, then the j th Pre-Hashing function can not be matched either. Thus $P_{tn_j'}$ is determined from the independent probability of True Negative P_{tn_j}, which can be obtained from (2). Thus $P_{tn_j'}$ is the first Pre-Hashing function, it can be obtained by

$$P_{tn_1'} = P_{tn_1} \times TH_1 , \quad (5)$$

and subsequent $P_{tn_j'}$ for length j is computed by

$$P_{tn_j'} = \left(1 - \sum_{y=1}^{j-1} P_{tn_y'}\right) \times P_{tn_j} \times TH_j . \quad (6)$$

As state before, a large number of Suffixes will need a big bit vector, a $Threshold_j$ parameter is applied to limit the rapid growth of the bit vector size. TH_j is the $Threshold_j$ rate for Suffixes of length j, and it can be obtained from

$$TH_j = \frac{N_j}{|S|} , \quad (7)$$

where N_j is the number of states in which the number of Suffixes is less than $Threshold_j$, and $|S|$ is the number of states.

The higher P_{tn} results in a better matching performance, since the low probability of text need to be matched for high P_{tn}. That less length can also achieve acceptable P_{root}. 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.

For the space evaluation, first of all, we need to determine the bit vector size M. Becasue the probability of True Negative is defined as (2), the M can be determined by given $|\beta|$ and P_{tn} as

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

In Section 4.2, our real pattern analysis shows $|\beta|$ is small for most of the states and the proper $Threshold_j$ value is about 8 for the URL and Virus patterns.

The space requirement can be determined by summing the Bitmap AC space $Size_{AC}$, the Pre-Hashing 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 AC, $Size_{AC}$, is mainly dominated by the state table, which is equal to the number of states $|S|$ multiplying with the state size $Size_{state}$,

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

The Pre-Hashing size $Size_{pre-hash}$ is the sum of bit vector size for the states which the number of Suffixes is smaller than $Threshold_j$. Because $Size_{pre-hash}$ is affected by $|S|$, TH_j, the bit vector size M_j for length j and the maximum length of Pre-Hashing $k_{pre-hash}$. Thus $Size_{pre-hash}$ is obtained from

$$Size_{pre-hash} = |S| \times \sum_{j=1}^{k_{pre-hash}} M_j \times TH_j . \quad (11)$$

$Size_{root}$ includes root index table and root next table. The size of root index table is 256 multiplying k_{root}, and the root next table is the number of next state address multiplying with the state address size $Size_{state_address}$. The number of root next state address is the cross product of the number of appearing letters for each length RN_j. Then $Size_{root}$ is formulated as

$$Size_{root} = 256 \times k_{root} + \prod_{j=1}^{k_{root}} RN_j \times Size_{state_address} \quad (12)$$

4.2. Real Pattern Analysis

In this analysis, we choose the URL blacklists and Virus signatures from http://www.squidguard.org/blacklist/ and http://www.clamav.net, respectively. Because the URL blacklists and Virus signatures have a lot of patterns and long patterns, these patterns are sufficient to evaluate the performance of our Parallel Automaton algorithm.

Our analyzed URL blacklist has 21,302 patterns and generates 194,096 states, and Virus signature has 10,000 patterns and generates 402,173 states. In the Suffix counting to obtain TH_j for the Suffixes of length 1 and 2. Our results show that, when the

IEEE COMPUTER SOCIETY

counting number is less than 8 for length 1, URL and Virus patterns have 68% and 49% states in using Pre-Hashing matching, respectively. Our results show 41% and 32% states for length 2 of URL and Virus patterns, respectively. These results show most states of the URL and Virus patterns have few numbers of Suffixes, so Pre-Hashing approach are useful to reduce the matching time.

5. Conclusion

A Parallel Automaton algorithm with the Pre-Hashing and Root-Indexing techniques is proposed in this paper. Our Pre-Hashing technique is used to verify quickly the text to avoid AC matching; it has two distinguished enhancements from previous BFSM. First, Bloom Filter uses all patterns to build a big vector, but our approach builds the bit vector from partial patterns. Second, BFSM uses multiple hashing key, but our approach uses only one hashing key. Therefore, our Pre-Hashing significantly reduces the hardware complexity, and makes hashing technique feasible in string matching.

Also our Root-Indexing matching is a size efficient and parallel matching technique for matching the multiple bytes in one single matching. Since root state is frequently visited in the string matching, it is an effective approach to accelerate the Automaton.

In the substantial evaluation, our Parallel Automaton can achieve at least the 494% and 224% speedup for Bitmap AC in the URL and Virus patterns. Since the Threshold is an upper bound for the number of Suffixes, many states will have the higher probability of True Negative. Thus, these results are the conservative evaluation. Moreover, our Parallel Automaton increases no worst-case time to Bitmap AC by performing the Pre-Hashing, Root-Indexing, and AC in parallel.

For the space requirement, our Parallel Automaton only increases 4 bytes in each state and one Root-Indexing size for root state. Therefore, the increased space 10.73 MB and 5.22 MB for URL and Virus patterns are quite acceptable with present technologies.

6. References

[1] The Open Source Network Intrusion Detection System, http://www.snort.org/.

[2] Clam AntiVirus , http://www.clamav.net/.

[3] The Apache SpamAssassin Project, http://spamassassin.apache.org/.

[4] DansGuardian content filter, http://dansguardian.org/.

[5] SquidGuard filter, http://www.squidguard.org/.

[6] G. Navarro, “A guided tour to approximate string matching,” *ACM Computing Surveys*, 33(1):31-88, 2001.

[7] S. Wu and U. Manber, “Fast text searching allowing errors,” *Communication of the ACM*, 35:83-91, 1992.

[8] R. S Boyer and J. S. Moore, “A fast string searching algorithm,” *Communications of the ACM*, 20, 10, 762–772, 1977.

[9] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Communications of the ACM*, pp.333–340, 1975.

[10] C. Coit, S. Staniford and J. McAlerney, "Towards Faster String Matching for Intrusion Detection," *In Proceedings of the DARPA Information Survivability Conference and Exhibition*, pp. 367-373, 2002.

[11] N. Desai, “Increasing Performance in High Speed NIDS,” www.snort.org/ docs/Increasing_Performance_in_High_Speed_NIDS.pdf, 2002.

[12] Y. Mishina and K. Kojima, “String matching on IDP: A string matching algorithm for vector processors and its implementation,” *In Proceedings of 1993 IEEE International Conference on Computer Design*, 1993.

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

[14] M. Raffinot, “On the multi backward dawg matching algorithm (MultiBDM),” *In Proceedings of the 4th South American Workshop on String Processing*, Carleton U. Press, pp.149-165, 1997.

[15] H. M. Blüthgen, T. Noll and R. Aachen, “A programmable processor for approximate string matching with high throughput rate,” *In Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp.309 -316, 2000.

[16] R. Sastry, N. Ranganathan and K. Remedios, “CASM: a VLSI chip for approximate string matching,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Volume:17 Issue: 8, pp. 824 -830, Aug 1995.

[17] J. H. Park and K. M. George, “Parallel String Matching Algorithms based on Dataflow,” 32nd Annual Hawaii International Conference on System Sciences, 1999.

[18] R. Franklin, D. Carver, and B. L. Hutchings, “Assisting network intrusion detectionwith reconfigurable hardware,” *In IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, Apr. 2002.

[19] J. Moscola, M. Pachos, J. W Lockwood and R. P. Loui, “FPsed: a streaming content search-and-replace module for an Internet firewall,” *11th Symposium on High Performance Interconnects*, Stanford, CA, pp. 122- 129, 2003.

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

[21] M. Attig, S. Dharmapurikar, and J. Lockwood, “Implementation Results of Bloom Filters for String Matching,” *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2004.

1063-6862/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY