

A Specialized Cache of Automata Matching for Content Filtering

Kuo-Kun Tseng, Chao-Shi Wang, Ya-Ying Liao

Abstract—Light-weight network gateways often employ a cost-effective embedded network processor and have received a strong demand for empowering content filtering services. In these regard, we were motivated to propose a specialized cache, fuzzy-updated cache automata matching (FCAM) circuit for accelerating the embedded network processors. Although automata matching algorithms are robust with deterministic matching time, there is still plenty of room for improving its average-case performance. The proposed FCAM employs cache to accelerate the root state and non-root state with the multiple characters matching, and applies the fuzzy decision to improve the cache performance. In our experiment, the FPGA implementation of FCAM can perform at the rate of 10.5 giga bit per second with the patterns of 25,642 bytes. This performance is superior to previous matching hardware in terms of throughput and pattern set.

Index Terms—String matching, Cache memories, Finite automata, Fuzzy control.

I. INTRODUCTION

IN recent years, deeper and more complicated content filtering has been required for applications dealing with intrusion detection, keyword blocking, anti-virus and anti-spam. In the content filtering applications, string matching usually occupies 30% to 70% of the systems' workload [1], [2]. Hence, as the transmission speed increases, it becomes very necessary to design an appropriate string matching accelerator to offload the work of string matching from the network processor.

To understand the necessary requirements of string matching algorithms, we surveyed real patterns from open source software which includes Snort [3] for intrusion detection, ClamAV [4] for anti-virus, SpamAssassin [5] for anti-spam, and SquidGuard [6] and DansGuardian [7] for Web blocking. Their requirements for string matching can be summed up as matching the variable-length, multiple simple patterns, and on-line processing of all content filtering systems. Since complex patterns can be converted to patterns composed of multiple simple patterns [8], they are optional in most applications.¹

Also, current existing on-line string matching algorithms can be classified into four categories, namely, dynamic programming, bit parallel, filtering and deterministic finite automata (DFA) algorithms. The dynamic programming [9] and bit parallel [10] algorithms are inappropriate for variable-length and multiple simple patterns, while the filtering algorithms [11] have poor worst-case time complexity $O(nm)$, where n and m are the

length of the text and patterns, respectively. Only automata based algorithms support the features of variable-length, multiple simple patterns and deterministic worst-case time complexity $O(n)$. Hence, the DFA based algorithm is a robust algorithm that can stand against malicious traffics and selected as the base to develop our new approaches.

Aho-Corasick (AC) [12] is a typical DFA base for string matching. However, there are several variations. Bitmap AC [13] uses bitmap compression to reduce the storage of AC states. AC_BM [1, 14, 15] 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 with the BM approach. AC_BDM [16] combines AC with backward dawg matching (BDM), and also improves the average-case time complexity of the conventional AC. Bit-split AC [17] splits the width of the input text into a smaller bit width to reduce the memory usage and the number of comparisons for selecting the next states. Since AC_BM, AC_BDM and bit-split AC are impractical for a large number of patterns. A scalable bitmap AC with space efficiency is more suitable for our purpose.

On the use of fuzzy-updated cache technique, two papers mentioned about the fuzzy technique in cache memory, and [18] used fuzzy replacement for generic cache memory. Also, [19] integrated the neuro and fuzzy logic for cache memory control. However, there is no related fuzzy-updated cache that was proposed for specific automata matching. Although bitmap AC has the good worst-case matching time complexity in $O(n)$, it is insufficient for high speed processing. In this paper, we present a fuzzy-updated cache automata matching (FCAM) with two novel fuzzy-updated cache, namely the root cache and the state cache techniques to accelerate automata based algorithms.

In addition, among the existing string matching hardware, the most prevalent is the finite automata (FA) based hardware. This is the case because of the support of the deterministic matching time and large patterns. FA based hardware can be divided into the deterministic FA (DFA) and non-deterministic FA (NFA) based hardware.

For DFA based hardware, there are three common designs in recently developed string matching hardware including Aho-Corasick (AC) based hardware [17, 20], Regular Expression (RE) based hardware [21, 22] and Knuth-Morris-Pratt (KMP) [23, 24, 25] based hardware. To save a great number of 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 supports a single simple pattern.

As for NFA based hardware, there are two variations, namely, the comparator NFA [26, 27] which uses the distributed comparators and the decoder NFA [28] which uses the character decoder (shared decoder) to build its NFA circuits. The other

Kuo-Kun Tseng Author is with Hungkuang University, 34 Chung-Chie Rd, Sha Lu, Taichung, 443, Taiwan, R.O.C. (Tel:886-937-789380 ; Fax:886-4-26310744; e-mail: kkseng@sunrise.hk.edu.tw).

Chao-Shi Wang and Ya-Ying Liao Authors are with Hungkuang University, 34 Chung-Chie Rd, Sha Lu, Taichung, 443, Taiwan, R.O.C. (e-mail: key135g@gmail.com and nacy90033@yahoo.com.tw, respectively)

existing non-DFA based hardware in our classification are the parallel comparator [29, 30, 31], Bloom filter [32], systolic array [33] and parallel and pipeline [34] hardware in our classification.

The rest of this paper is organized as follows: Section 2 describes our algorithm, architecture, and the detailed design of the fuzzy-updated cache. The objective evaluation and the analysis of space requirement and performance are presented in Section 3, while the FPGA implementation and comparison are described in Section 4. Finally, we draw our conclusion in Section 5.

II. ALGORITHM AND ARCHITECTURE

The proposed FCAM incorporates three operations in the algorithm, the state cache, root cache and AC matching operations, which are described in the first subsections. In the second subsections, we illustrate the parallel hardware architecture for our automata matching. For lower cache miss rate, a new fuzzy updated technique of automata cache is described in the last section.

Matching Algorithm

```

FA_matching(T, FA) {
    ML=0; FA.S_c = FA.S_0;
    for(i=0; i<=|T|; i++) {
        checkout(FA.S_c);
        t_current = t_current + ML;
        ML=0;
        if(FA.S_c == FA.S_0) {
            RC = root_cache_match(T[t_current : t_current + l_rc]);
            if(RC.Match == true)
                ML = RC.ML; FA.NS = RC.NS;
            else {
                FA.NS = AC_match(T[t_current]);
                ML = 1;
            }
            root_cache_update(T[t_current - l_rc - 1 : t_current - 1]);
        } else {
            SC = state_cache_match(T[t_current : t_current + l_sc]);
            if(SC.Match == true)
                ML = SC.ML; FA.NS = SC.NS;
            else {
                FA.NS = AC_match(T[t_current]);
                ML = 1;
            }
            state_cache_update(T[t_current - l_sc - 1 : t_current - 1]);
        }
    }
}
    
```

Fig. 1. The sequential version of FCAM matching algorithm is written in C-like language.

As shown in Fig. 1, the FCAM matching algorithm is written in C-like pseudo-code. The state cache is applied to each state, except for the root state, in order to avoid the bitmap AC operation and to match multiple bytes in a single operation. During the automata matching $FA_matching()$ process, for-loop feeds the text T into the matching functions from length zero to maximum text length $|T|$. If the current state $FA.S_c$ is set to a root state $FA.S_0$, $FA_matching()$ will use the result RC of root

cache matching $root_cache()$. If the root matched status $RC.Match$ indicates a matching hit, its next state $RC.NS$ is used for the next state transition $FA.NS$. If the current state is not the root state, it will perform state cache matching $state_cache()$ first. If the state matched status $SC.Match$ indicates a hit, the matched result of state cache $SC.NS$ is used for $FA.NS$ as well. Otherwise, the result of $AC_match()$ is used for $FA.NS$. A cache update is performed in each matching operation after $FA.NS$ is obtained by matched units.

After the matching, the current text position $t_current$ will advance the matched length ML for each matching iteration. ML is set to $RC.ML$ and $SC.ML$ if the root cache or state cache have a matching hit. Moreover, to select the text for the cache match and updated units, the maximum matched length of the root cache l_rc and the state cache l_sc are used to control the text windows. To output the matched result, $checkout()$ is performed after the next state is obtained. Although this C-like algorithm looks like a sequential flow, $checkout()$, $advance_text()$, $root_cache()$, $state_cache()$, $AC_match()$, $root_cache_update()$ and $state_cache_update()$, the units can be executed concurrently in the hardware.

Matching Architecture

The proposed FCAM hardware as depicted in Fig. 2 can be implemented as a coprocessor to assist the string matching. Its three units can perform their matching concurrently in the hardware. In this architecture, the AC matching processes only one-byte in a single matching iteration, but the state and root cache units can match multiple bytes in a single matching iteration.

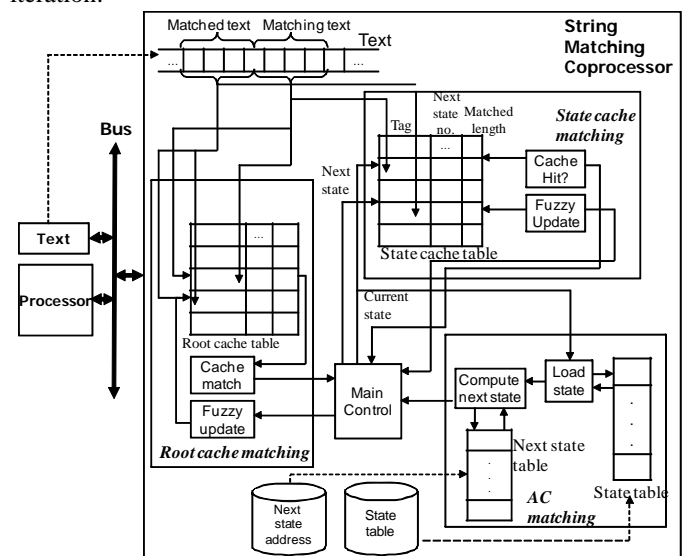


Fig. 2. The architecture of the string-matching co-processor, including the logic circuits and blocks of the root cache, state cache and AC matching for the FCAM co-processor.

In the state or root cache tables, each entry consists of the tag, next state and matched length. Since the state cache uses the partial state number and the matching text as its tag, the entry size of the state cache table will be larger than the root cache table.

An Fuzzy Updated Unit for Cache

A new fuzzy-updated cache updated technique to improve the cache matched probability is proposed. This fuzzy-updated cache can incorporate more automata matching features in order to improve the cache locality, such as the visiting frequency and the distance from the root state. Fig. 3 represents the updated unit for the root cache. The state cache updated unit is quite similar to the root cache although there are two major differences: (1) the state cache is indexed by a state number, and (2) it uses a fine-tuned parameter for producing a fuzzy update.

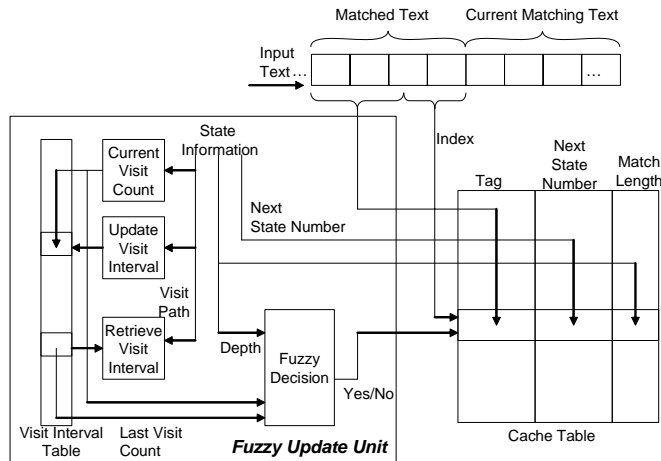


Fig. 3. Fuzzy updated unit for root cache.

In the implementation of the fuzzy updated mechanism, the fuzzy decision is based on two parameters, namely, the visit interval and the depth. Their fuzzy membership functions are illustrated in Fig. 4. (a) and (b), respectively. One special design feature that is used to reduce the entry of a visit interval for each state is to use the visit path to index the visit interval instead of the state number. Each visit path in the visit interval table keeps a last visit count LVC and the average visit interval VI_{avg} in the table. VI_{avg} is computed using a moving average approach as in (1) below

$$VI_{avg} = VI_{avg} \times F_{vi} + (CVC - LVC) \times (1 - F_{vi}), \quad (1)$$

where the visit interval represents the current visit count CVC subtracted from LVC , with a weight sensitive factor F_{vi} , such that a larger F_{vi} will be less sensitive to a new visit interval.

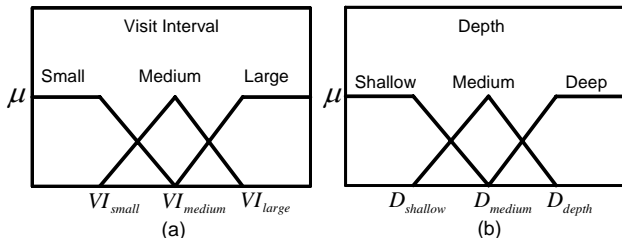


Fig. 4. (a) Membership functions of the visit interval parameter. (b) Membership functions of the depth parameter.

A fuzzy rule table is used to determine the updated decision. The Yes/No assignment of the fuzzy rule table is based on the experimental results that can be modified for different applications and situations. To compute the final decision, the Centroid method is applied in the fuzzy computation.

III. ANALYSIS

Space Requirement

A. Space Requirement

The space requirement can be determined by combing the bitmap AC, state cache and root cache spaces, as

$$SP_{total} = SP_{AC} + SP_{sc} + SP_{rc} \quad (2)$$

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

$$SP_{AC} = |S| \times SP_{state} \quad (3)$$

The state cache space SP_{sc} is determined by its entry amount EA_{sc} multiplied with its entry size ES_{sc} , as

$$SP_{sc} = EA_{sc} \times ES_{sc} \quad (4)$$

The size of each entry is summed by the size of state tag $|Tag_{sc}|$, state number SP_{state_no} and match length ML_{sc} , which can be obtained using

$$ES_{sc} = |Tag_{sc}| + SP_{state_no} + ML_{sc} \quad (5)$$

Since the state cache is indexed by the first partial state number $SP_{state_no_part1}$, the tag size of each entry $|Tag_{sc}|$ is the sum of the size of second partial state number $SP_{state_no_part2}$ and the text size $|T|$, then minus $SP_{state_no_part1}$.

$$|Tag_{sc}| = SP_{state_no_part2} + |T| - SP_{state_no_part1} \quad (6)$$

With the above equation, different configurations of the state cache can be obtained by a change in the address width of the index.

The root cache space SP_{rc} is quite similar to the state cache; only the root tag size $|Tag_{rc}|$ is different and is equal to the size of second partial text. Thus SP_{rc} and the entry size ES_{rc} of the root cache are (7) and (8), as

$$SP_{rc} = EA_{rc} \times ES_{rc} \quad (7)$$

$$ES_{rc} = |Tag_{rc}| + SP_{state} + ML_{rc} \quad (8)$$

TABLE I
COMPARISON FOR FUZZY-UPDATED CACHE PERFORMANCE

Address Width (Bit)	Entry Amount	Tag Size (Bit)	State NO. Size (Bit)	Match Length (Bit)	Root Cache Size (KByte)	State Cache Size (KByte)
16	65536	72	24	3	6336	4800
15	32768	73	24	3	3200	2432
14	16384	74	24	3	1616	1232
13	8192	75	24	3	816	624
12	4096	76	24	3	412	316

From the above-mentioned equations, the space root cache and state can be computed as in Table I. If the entry amount of cache table is from 4096 to 65536 for the state amount of bitmap AC from 1,048,576 to 16,777,216. The sizes of root cache and

state cache are from 412 to 6336 Kbytes, and 316 to 4800 Kbytes, respectively. Compared to the size of bitmap AC, the state amount of bitmap AC from 16,777,216 and 1,048,576 requires 4,992 and 312 Mbytes, respectively. The sizes of the cache tables are relative small and thus, it is feasible to implement the cache tables with the internal memories.

Performance Analysis

As mentioned earlier, the root cache, state cache and AC matching can be performed concurrently, so that the average matching time of the matching automata is computed as

$$T_{avg_time} = \frac{P_{rc} \times T_{rc} + P_{sc} \times T_{sc} + (1 - P_{rc} - P_{sc}) \times T_{AC}}{(ML_{rc} \times P_{rc}) + (ML_{sc} \times P_{sc}) + (1 - P_{rc} - P_{sc})} \quad (10)$$

where T_{avg_time} is the average time for matching one byte of the text, T_{rc} , T_{sc} and T_{AC} are the root cache, state cache and AC matching time, respectively. P_{rc} , P_{sc} and P_{AC} are the probabilities of using the root cache, state cache and AC matching, respectively. Since the state cache can advance multiple bytes in one single matching, ML_{sc} and ML_{rc} are the average matched length for both the state cache and root cache. Since the AC matching is the critical path, the worst-case time of the FCAM is equal to T_{AC} .

With the matching algorithm and architecture in Section 3, the proposed FCAM was implemented using the Verilog language, and then compared with the first in first out (FIFO) and the least recently used (LRU) cache update algorithms. In this comparative stage, the different cache entry amount configurations and the two patterns of major applications, namely, the URL block list and the virus patterns were also tested.

TABLE II
COMPARISON FOR FUZZY-UPDATED CACHE PERFORMANCE

Throughput(bps)	Fuzzy Update		FIFO Update		LRU Update		
	Virus	URL	Virus	URL	Virus	URL	
Cache	4096	6738	10121	2413	3617	2900	4618
Entry	8192	7546	10490	2535	4440	3613	5048
Amount	16384	8355	10563	3630	5536	4062	6194
	32768	9163	10526	4725	6770	6479	7196
	65536	9214	10665	6063	8003	6553	8544

For realistic evaluation, the URL blacklists and virus signatures from [6] and [4] were chosen. Since the URL blacklists and virus signatures have a lot of patterns as well as long patterns, these patterns are sufficient to evaluate the performance of our FCAM algorithm. The analyzed URL blacklists have 21,302 patterns and generate 194,096 states, while the virus signatures have 10,000 patterns and generate 402,173 states. Finally, the Google website and the ethereal protocol captures were used to test the URL and virus patterns, respectively.

Table II shows that the proposed fuzzy updated technique can archive around 10.5 Gbps for the URL patterns, which is superior to FIFO and LRU algorithms. Moreover, fuzzy update provides a better performance when the cache entry amount is not large.

Fig. 5 provides a more detailed view of cache performance, matching probability of root cache and state cache. Obviously, a

fuzzy-updated cache has much better matched probability than FIFO and LRU caches. A URL has better performance than a virus application by providing a higher matching probability. Another interesting finding is that the state cache matching probability does not always increase with an increase in cache size: especially, when root cache matched probability is high.

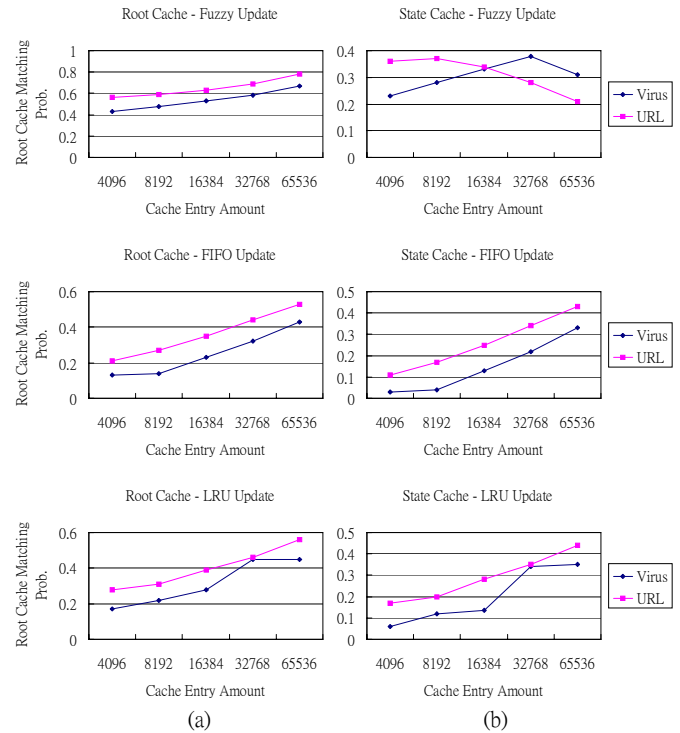


Fig. 5. Cache matching probability (a) for root cache and (b) for state cache.

IV. HARDWARE IMPLEMENTATION AND COMPARISON

When compared with various hardware algorithms, our FCAM showed better results. Since many string matching hardware [20, 22, 23, 31, 34] store their patterns in on-chip hardwired circuits and internal memories, we implemented the proposed FCAM using the FPGA internal memories for a fair evaluation. Besides, we also synthesized FCAM on various Xilinx FPGA devices to directly compare its performance with the other matching hardware.

TABLE II. The Comparisons of String Matching Hardware

Matching Hardware	Device	Pattern Size (Byte)	Throughput (Gbps)
RHAM ²	Virtex2P	25,642	10.5
	Virtex2 1000		6.2
	Virtex2 6000		8.8
	Spartan3 400		6.4
	VirtexE 2000		2.0
	Virtex 800 ²		1.6
Bit-split AC [17]	Xilinx FPGA	2,048	10.0
DFA+counter [21]	VirtexE 1000	11	3.8
Parallel Regular DFA [22]	VirtexE 2000E	420	1.2
KMP Comparators [23]	Virtex2P	32	2.4
Comparator NFA [26]	Virtex 100	29	0.5
Meta Comparator NFA [27]	VirtexE 2000	8,003	0.4
Approximate Decoder NFA [28]	Virtex2 6000	17,537	2.0
Offset Index Comparators [29]	Spartan3 400	20,800	1.9
Pre-decoded Comparators [30]	Virtex2 6000	18,032	9.7
CAM Comparators [31]	VirtexE 1000	640	2.2
Parallel Bloom Filter [32]	VirtexE 2000	9,800	0.6

In the comparison, we compared and analyzed about 11 major

recent works as shown in Table II. In addition, the architecture of the previous hardware often employed hardwired circuits and internal memories for storing their pattern sets and thus, their pattern sizes were limited by FPGA resources. Our FCAM architecture is scalable to support more patterns with higher performance because it can be implemented with external multiple banks memory.

V. CONCLUSION

In this paper, we present a fuzzy-updated cache automata matching (FCAM) with a novel state cache and root cache techniques. With the new architecture, the proposed FCAM technique avoids the time consuming bitmap AC matching and does the matching of multiple bytes in one single matching operation. Our FCAM has a distinguishing feature compared to previous fuzzy caches, i.e., it has a specialized cache for automata structure that is novel compared to all previous work. In addition, the result of FCAM implementation demonstrates that it surpasses all other existing hardware in terms of pattern set and throughput. The FCAM can support the largest pattern size of 25,642 bytes and run at the highest throughput of 10.5 Gbps. Moreover, since the proposed architecture works for both external and internal memories, and that the external ASIC memories can often run at a much higher clock rate than the FPGA memories, the architecture of FCAM is scalable for large pattern sets.

REFERENCES

- [1] F. Mike and V. George, "Fast Content-Based Packet Handling for Intrusion Detection," *UCSD. Technical Report CS2001-0670*, May 2001.
- [2] S. Antonatos, K. Anagnostakis and E. Markatos, Generating Realistic Workloads for Network Intrusion Detection Systems. *ACM WOSP*, 2004.
- [3] M. Roesch et al, "Snort: The Open Source Network Intrusion Detection System," <http://www.snort.org/>.
- [4] T. Kojm et al, "Clam Anti-virus," <http://www.clamav.net/>.
- [5] J. Mason et al, The Apache SpamAssassin Project. <http://spamassassin.apache.org/>.
- [6] T. D. Internordia et al, "SquidGuard filter," <http://www.squidguard.org/>.
- [7] D. Barron et al, "DansGuardian content filter," <http://dansguardian.org/>.
- [8] G. Navarro and M. Ranot, "Flexible Pattern Matching in Strings," *Cambridge University Press*, 2002.
- [9] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, 33(1):31-88, 2001.
- [10] S. Wu and U. Manber, "Fast Text Searching Allowing Errors," *Communication of the ACM*, 35:83-91.
- [11] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm." *Communications of the ACM*, 20, 10, 762-772.
- [12] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, pp.333-340.
- [13] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *IEEE Infocom*, Hong Kong, China, 2004.
- [14] C. Coit, S. Staniford and J. Mcalerney, "Towards Faster String Matching for Intrusion Detection," *DARPA Information Survivability Conference and Exhibition*, pp. 367-373, 2002.
- [15] N. Desai, "Increasing performance in high speed NIDS," <http://www.snort.org/>.
- [16] M. Raffinot, "On the Multi Backward Dawg Matching Algorithm (MultiBDM)," *Workshop on String Processing*, Carleton U. Press, 1997.
- [17] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection And Prevention," *ISCA*, 2005.
- [18] H. Diab, U. Furbach and H. Tabbara, "On the Use of Fuzzy Techniques in Cache Memory Management," *JCIS*, Atlanta, 2000.
- [19] O. Hammami, "Pipeline Integration of Neuro and Fuzzy Cache Management Techniques," *FUZZ-IEEE*, Barcelona, Spain, Jul. 1997.
- [20] M. Aldwairi, T. Conte and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," *ACM CAN*, 2005.
- [21] J. Lockwood, "An Open Platform for Development of Network Processing Modules in Reconfigurable Hardware," *IEC DesignCon*, Santa Clara, CA, Jan. 2001.
- [22] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," *IEEE FCCM*, 2003.
- [23] Z. K. Baker and V. K. Prasanna, "Time And Area Efficient Pattern Matching on FPGAs," *ACM/SIGDA FPGA*, California, USA, Feb 2004.
- [24] G. Tripp, "A Finite-State-Machine Based String Matching System for Intrusion Detection on High-Speed Network.," *EICAR*, May 2005.
- [25] L. Bu and J. A. Chandy, "A Keyword Match Processor Architecture Using Content Addressable Memory," *ACM VLSI*, April 26-28, 2004.
- [26] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching using FPGAs," *IEEE FCCM*, April 2001.
- [27] R. Franklin, D. Carver and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *IEEE FCCM*, Napa, CA, Apr 2002.
- [28] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," *IEEE FCCM*, 2004.
- [29] Y. H. Cho and W. H. Mangione, "A Pattern Matching Coprocessor for Network Security," *ACM/IEEE DAC*, California, USA, Jun 2005.
- [30] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," *IEEE FCCM*, 2004.
- [31] 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.
- [32] 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.
- [33] H. M. Blüthgen, T. Noll and R. Aachen, "A Programmable Processor For Approximate String Matching With High Throughput Rate," *IEEE ASAP*, 2000.
- [34] J. H. Park and K. M. George, "Parallel String Matching Algorithms based on Dataflow," *HICSS*, Hawaii, 1999.