Towards High-performance Pattern Matching on Multi-core Network Processing Platforms

Yaxuan Qi[1], Zongwei Zhou[3], Yiyao Wu[4], Yibo Xue[2] and Jun Li[1, 2]
[1]Department of Automation, Tsinghua University, Beijing, 100084, China.
[2]Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, 100084, China.
[3]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA.
[4]Information Networking Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

ABSTRACT

With the continual growth of network speed and the increasing sophistication of network applications, keeping network operations efficient and secure becomes more challenging. Pattern matching is one of the key technologies for content-ware network processing, such as traffic classification, application identification and intrusion prevention. In this paper, we propose a hybrid pattern matching algorithm optimized for multi-core network processing platforms. As a system-level solution, our scheme focuses on both performance stability and hardware/software co-design. To verify the effectiveness of our design, the proposed algorithm is implemented on a state-of-art 16-MIPS-core network processing platform and evaluated with real-life data sets. Experimental results show that, when compared with the traditional Aho-Corasick algorithm, our hybrid solution saves 60~95% memory space while guarantees stable performance on large pattern sets and against adverse test traffic.

I. INTRODUCTION

Due to the ever-increasing bandwidth and sophistication of the Internet, keeping network operations efficient and secure becomes more challenging than ever. To build high-performance network devices with holistic protection, efficient and in-depth packet processing features need to be integrated into modern network devices. For example, modern security gateway is often integrated with intrusion detection, application-identification, and anti-virus. Pattern matching is one of the key technologies for these content-aware network security applications [1] [2] [3].

To reach high-performance pattern matching, a large number of algorithms have been proposed in recent years. However, most of the proposed algorithms are point-solution based on particular hardware and therefore cannot be integrated to the emerging multi-core network processing platforms. This is mainly due to the lack of:

- **Performance stability:** Many existing algorithms work well under normal conditions, but cannot guarantee stable performance in special situations. For example, data sets with large number of patterns might exponentially increase the memory usage, and traffic with a lot of attack patterns might significantly affect the matching performance. In practice, unstable performance means vulnerability to deny of service (DoS) attacks.
- **Software hardware co-design:** On the one hand, many existing software algorithms stay in mathematical analysis or software simulation stage, and only a few of them exploit the architectural advances of modern multi-core platforms [8] [9]. On the other hand, hardware solutions based on ASIC/FPGA have poor portability and scalability, making it hard to be integrated into commercial network processing platforms.

Therefore, the need for more general and flexible algorithmic solutions motivates our research today. In this paper, we solve these problems from the general line of thought. Main contributions of this paper are:

- **A hybrid pattern matching algorithm**: By thorough analysis of existing algorithms, we propose a hybrid algorithm leveraging the advantages of both DFA based algorithms and hash based algorithms. We also introduce the hardware/software co-design principles to optimize the hybrid algorithm for commercial multi-core network processing platforms.
- **System-level evaluation on real platform**: To verify the effectiveness of our design, the hybrid algorithm is implemented and tested on an advanced network processing platform with Cavium OCTEON3860 16-MIPS-core processor. Experimental results show that, when compared with Aho-Corasick algorithm, our solution saves 60~95% memory space while guarantees stable performance on large pattern sets and against adverse test traffic.

The following part of this paper is organized as follows. In Section II, we introduce the hybrid algorithm for pattern matching; in Section III, we show the experimental results and performance evaluation on the multi-core platform. As a summary, we state our conclusion in Section IV.

II. A HYBRID PATTERN MATCHING ALGORITHM

A. Background

Pattern matching represents the operation of finding one or all positions in the given text where certain patterns appear. In typical network security applications, patterns are often strings which represent special semantics of network traffic.

One of the most famous pattern matching algorithms is the Aho-Corasick (**AC**) [4]. This algorithm is adopted by a lot of network applications, such as the widely used open source network intrusion detection system SNORT [1]. The basic idea of AC algorithm is as follows: preprocess all patterns into deterministic finite automata (DFA), and input the text one character by another. Every character results in a transition from current status to a new one. The outputs are generated in certain states when pattern matches happens.

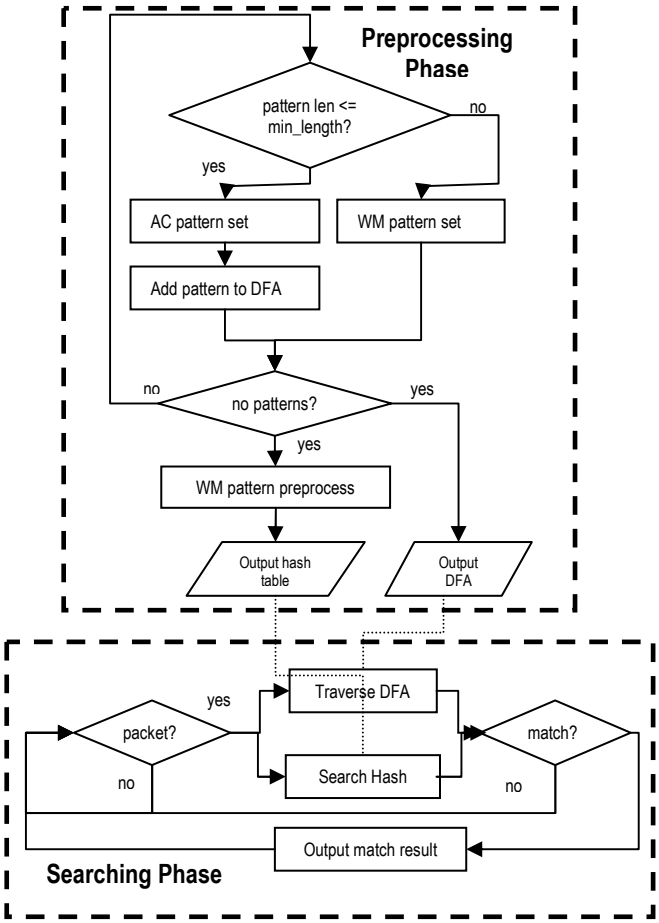

Figure 1. Hybrid pattern matching scheme

The advantages of DFA-based AC algorithm are: 1) its time complexity is linear to the length of input text, so that its performance is stable and not sensitive to input pattern sets [4]; the data structure of any AC automata states is identical when using same character set, such as ASCII; 3) traversing the DFA is simple and can be easily implemented by hardware or a co-processor to achieve high searching performance.

Therefore, the DFA-based AC algorithm seems to be well suited to multi-core platforms. However, this algorithm consumes exponentially large memory space as the number of patterns increases. Because large DFA cannot be stored in size-limited L2 cache, the overall performance will significantly fall down due to the random accesses to a large memory space. To solve the memory explosion problem, a variety of hash-based algorithms are proposed, including the well-known Wu-Manber (**WM**) algorithm [5]. The WM algorithm preprocesses first *m* characters of every pattern to build a series of hash tables (*m* is the length of the shortest pattern in the pattern set) and stores the possible "shift" value in the table. By checking the shift value of the input characters, the WM algorithm ether provides the next-step characters to load or reports a possible match.

Because the size of the hash table is fixed, the WM algorithm requires only $O(N)$ memory space, where N is the number of patterns. At the same time, because random input usually generates large shift values, the WM algorithm achieves good average performance. Therefore, the WM algorithm is considered efficient in terms of both search speed and memory usage. However, further analyses reveal that, there are two serious limits in the WM algorithm. One is the *pattern-set dependency*: according to the original WM algorithm, all the shift values must be smaller than the minimum pattern length (referred as ***min_length*** in the following discussion) in a given pattern set, so the shorter the *min_length*, the poorer the WM performance. The other limit is the unstable *performance*: when hash collision happens (the shift value is zero), the WM algorithm will load all the collided patterns to check for possible matches. Because loading multiple patterns often resorts to unpredictable memory access (caused by different number of collided patterns) with extra-high latency (caused by long patterns), the performance of WM is unstable.

Therefore, although DFA-based algorithms meet the basic byte operation requirement of pattern matching, the memory blowup issue limits their applicability. On the other hand, while hash-based algorithms successfully reduce the memory usage, their performance is unstable when *min_length* is small and/or collision rate is high.

B. Motivation

Considering all these issues, our motivation is to leverage the advantages of these two types of algorithms and propose a hybrid pattern algorithm optimized for multi-core network processing platforms. At first, this algorithm is based on the following three facts: 1) when DFA is small, the AC algorithm can achieve high performance. This is because network processors often have a small amount of low-latency memory or shared cache, such as the RLDRAM in Cavium OCTEON [6] and the SRAM in Intel IXP [7]. Small DFA could be put into these kinds of special memory or be pre-fetched into shared cache for extra-high speed access; 2) when *min_length* is large, the WM algorithm can achieve high performance. This is because larger *min_length* generates larger shift value, and thus accelerates the overall lookup speed of the WM algorithm; 3) in real-life application, the size of DFA generated by short patterns is considerably small. This is because the complexity of shorter patterns is significantly less then longer patterns, i.e. longer patterns generate more DFA states than shorter patterns.

According to the facts above, basic ideas of our algorithm are: 1) divide the pattern set into two sub-sets according to pattern length (*min_length*); 2) apply DFA-based algorithms to short patterns and hash-based algorithms to long patterns; 3) the algorithm runs in parallel or pipeline mode on multi-core network processor. As shown in Figure 1, the hybrid algorithm contains two phases: preprocessing phase and searching phase. In the preprocessing phase, the hybrid algorithm examines the pattern length one by one. If the pattern length is smaller than or equal to *min_length*, we add this pattern into DFA according to the DFA-based algorithm; otherwise, the pattern is put into the pattern set of hash-based algorithm. After examining all patterns, we then preprocess the pattern set of hash-based algorithm. During the searching phase, we run DFA-based algorithm and hash-based algorithm in parallel

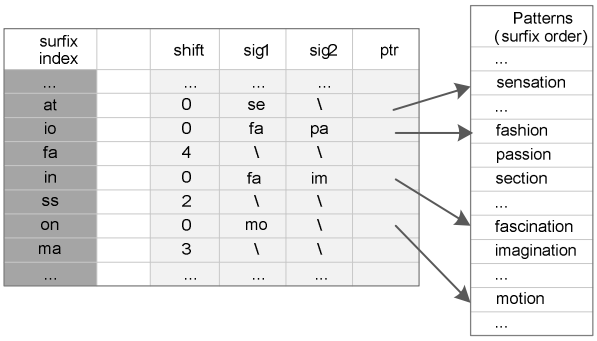

Figure 2. Signature-based WM (SigWM) Algorithm

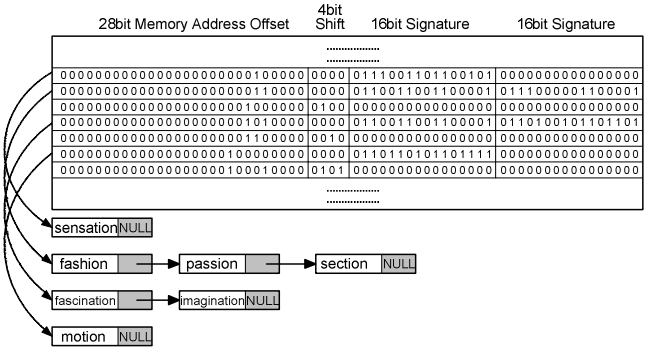

Figure 3. Data-structure of the SigWM Algorithm

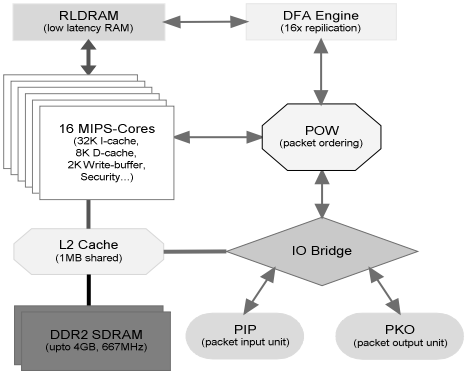

Figure 4. Cavium OCTEON3860 Architecture

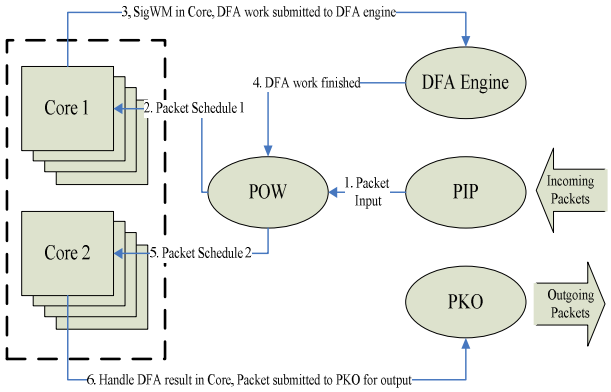

Figure 5. Implementation of the hybrid algorithm

mode to search the preprocessed sets for the pattern matching.

C. *Optimization*

First of all, the algorithm must have stable performance. DFA-based algorithms have stable time performance due to its byte-level state-transition. However, the performance of hash-based algorithm is unstable when collisions constantly happen. In our design, we use "signature" comparison other than pattern comparison to solve hash collision (to a certain extend). The signature is a hash value generated by patterns in collision (with the same suffix). If the signature of the input text is not identical to the signatures stored in the hash entry, we can avoid the unnecessary comparison with the corresponding patterns. Figure 2 explains the basic idea of signature-based WM algorithm. For convenience, we just depict the first-stage hash (suffix hash) in Figure 2, and the signature is simply defined as the prefix of the patterns in collision.

Secondly, the algorithm must have efficient data structure for efficient memory access on a network processor. According to previous analysis, DFA-based algorithms already have "neat" storage structure. So we only need to well arrange the signature-based WM algorithm to meet the 64-bit alignment of memory word. Figure 3 shows the optimized data structure of Signature-based WM algorithm: there are 4 bits for jump info, 28 bits for next-state pointer, and 32 bits for collision-resolve.

Finally, according to the architecture of Cavium OCTEON3860 multi-MIPS-core processor shown in Figure 4 [6], we can employ the hardware/software co-design principle to achieve optimized performance. Figure 5 shows the optimized implementation of the hybrid algorithm: In the preprocessing phase, we preprocess all patterns to generate DFA and SigWM data structure. DFA is then stored in the RLDRAM of the DFA engine, and SigWM data structure is stored in the DDR2 SDRAM (would be loaded into L1/L2 cache during searching phase). Incoming packets are scheduled to one of the 16 MIPS cores for SigWM searching, and also forwarded to the DFA engine. The DFA engine will run DFA-based algorithm over the packet payload. After that, the packets together with the search result will be scheduled to another core, and further operations will be done by this core according to the search results. In addition, given the pattern set and the size of available memory for DFA, the optimum partition of the pattern set are determined by choosing the maximum *min_length* under the following conditions: DFA_SIZE (*min_length*) < MEM_LIMIT.

III. PERFORMANCE EVALUATION

In this section, the proposed hybrid pattern matching algorithms are evaluated on Cavium OCTEON3860 network processing platform with real-life pattern sets. We will first describe the test environment and then show the test results under different pattern set and test setup.

A. *Data-set and test-bed*

Our study focuses on real-life patterns because we believe experimental results using real-life data sets are more convincing than those obtained on synthetic patterns. In our test, pattern matching rules are obtained from the open source

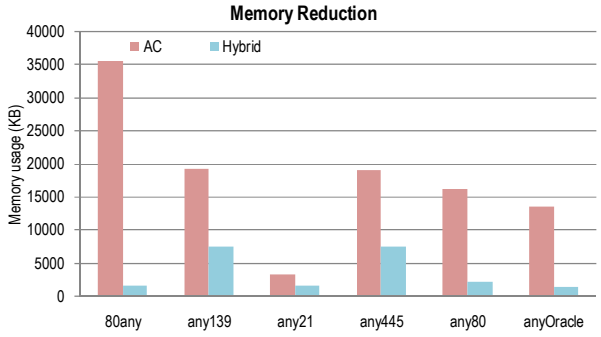

Figure 6. Memory usage stability

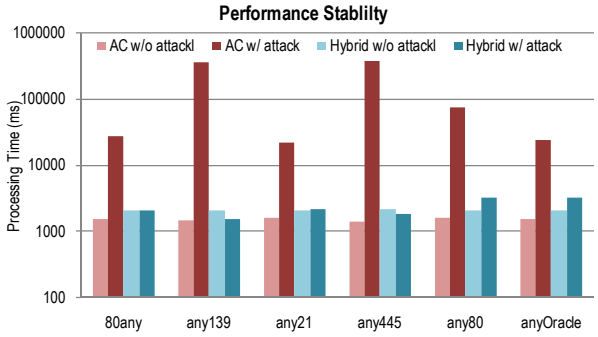

Figure 7. Processing time stability

intrusion detection system SNORT v2.8.There are totally 5831 patterns attracted from the SNORT database. The length of pattern ranges from 1 byte to more than 100 bytes. All the rules are publicly available from SNORT official site [1].

Hardware platform is a Cavium OCTEON3860 multi-core network processor. The OCTEON3860 has 16 MIPS cores running at 500MHz. Network interfaces are eight 1Gbps RGMII port. Memory hierarchy includes 1MB shared L2 cache, 2GB DDR2 SDRAM and 8x16MB RLDRAM. The PIP unit receives packets from network, then POW unit schedules packets (as a work) to different cores for packet processing, and finally packets are sent out from PKO unit. More detail about Cavium OCTEON3860 can be found at [6].

There are two basic programming choices with Cavium OCTEON Software Developer Kit (Cavium SDK version 1.5): programming in Linux mode (with Linux OS) and Simple executive mode (no OS). Because we are evaluating fast-path algorithms, all the algorithm are implement in simple executive mode (without any performance cost due to running an OS). Although Cavium provides a cycle-accurate simulator for software performance evaluation, all the test results in this paper are obtained from hardware test (OCTEON3860 board with SmartBits packet generator).

B. Performance stability evaluation

To make our experimental results more convincing, we use the same method as SNORT to group and build the DFAs. We merged the rule files and group the rules by destination and source ports. For each source or destination port that has relevant rules, we build the data structure for AC and the hybrid algorithm. For a particular rule, if it has only destination or source port specified, it will be compiled in the data structure of the corresponding port; if it has both destination and source port specified, it will be compiled into two data structures corresponding to both destination and source ports. In Figure 6 and Figure 7, the structures are denoted by port number. For example, *80any* means this structure contains all the rules with a source port equals 80, and *any139* means this structure contains all the rules with destination port equals 139.

The first performance stability evaluation is to measure the memory usage of each algorithm for on different pattern sets. The result shows that on average, the hybrid algorithm has 60~95% (70% on average) memory reduction. Figure 6 shows the memory usage comparison of 6 largest SNORT pattern sets. From this figure, we see that the memory usage of the hybrid algorithm is significantly smaller than that of the AC algorithm. As the number of patterns increases, the memory usage of AC algorithm exponentially grows up. While on the opposite, the hybrid algorithm only has sub-linear grow in its memory usage.

The other performance stability evaluation is the matching speed. We use two kinds of input packets to simulate normal and attack traffics: one is real-life traffic dumped from campus network and the other is synthetic traffic in which 90% packet content will trigger a match. Figure 7 shows that with an approximately the same performance under normal network traffic, while the hybrid algorithm has a much stable performance than AC algorithms under attack traffic. Such performance stability can be achieved because: when dealing with normal traffic, the AC algorithm has a good cache hit rate because only a few states (5% on average) are traversed so that most of the memory accesses may take place in system cache. However, the attack traffic will significantly increase the number of states to traverse, i.e. nearly most of the states need to access, so the cache hit rate drops and the overall time for processing increases. In comparison, since the memory usage of the hybrid algorithm is quite small, i.e. almost all data structure can be cached, the performance of the hybrid algorithm is nearly independent of the pattern matching rate.

C. Multi-core platform evaluation

Figure 8 and Figure 9 show the performance of AC and WM algorithm with different partition of the policy set. Patterns with less than *min_length* bytes are processed by AC algorithm. Patterns have longer length are handled by WM algorithm. From Figure 8 we can see that, with RLDRAM support, AC algorithm can achieve 2.4 Gbps throughput on OCTEON3860 when *min_length* is less than 10. However, if the DFA is stored in main memory (DDR2 DRAM), the throughput is less than 1.2 Gbps and decreases as the *min_length* increases (as the number of rules goes up, the L2 cache miss rate also increases). Thus, to reach multi-Gbps pattern matching, the DFA must be stored in RLDRAM with limited *min_length*. In comparison, WM performance with different *min_length* is shown in Figure 9. We can see from this figure that as the *min_length* increases, the performance of WM algorithm linearly grows up. When the *min_length* is greater than 4, the WM algorithm achieves more than 3 Gbps throughput. So according to Figure 8 and Figure 9, the

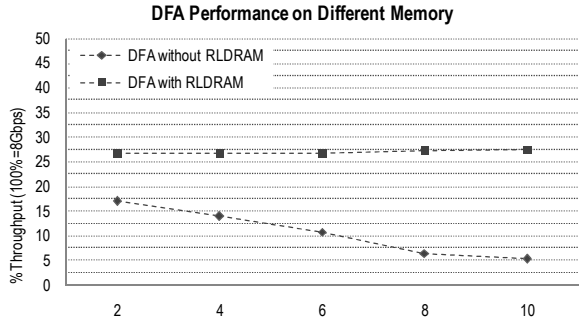

Figure 8. DFA Perf with Different min_length

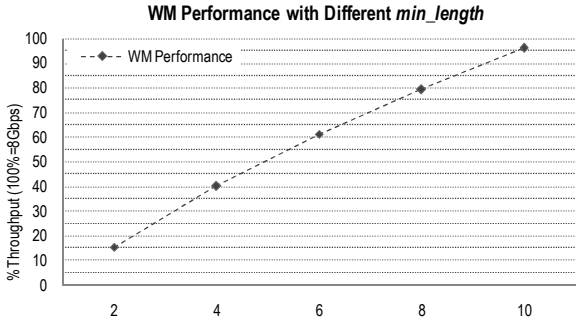

Figure 9. WM Perf with Different min_length

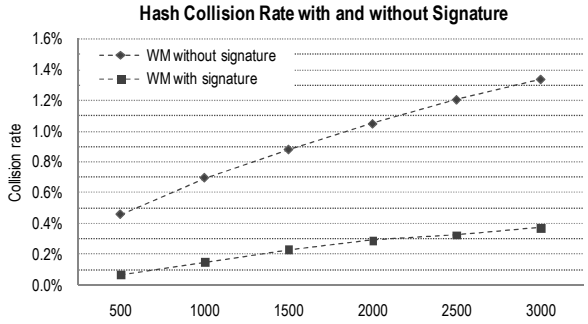

Figure10. WM and SigWM Collision Rate

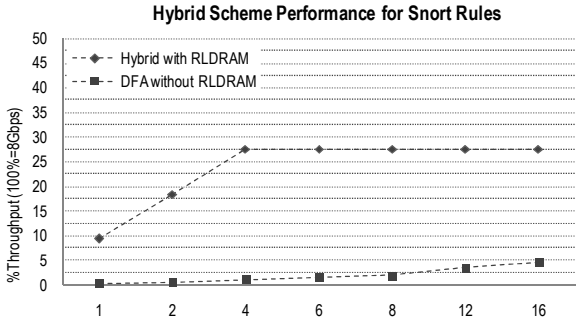

Figure 11. Hybrid Scheme Performance Speedup

min_length can be chosen from 4 to 10.

Figure 10 compares the collision rate of the original WM algorithm and the redesigned signature based scheme. We can see from this figure, as the number of rules increases, the collision rate of signature based scheme grows more slowly than the original WM. Even with 3,000 real-life patterns, the collision rate of our scheme is less than 0.4%, which is 70% less than that of the original WM algorithm. Although hard to verify using large traffic trace, we still believe that extremely low collision rate will bring about much better performance.

Figure 11 shows the performance of the hybrid scheme. With the RLDRAM support, the overall performance is stable at 2.15Gbps (limited by the DFA performance shown in Figure 8). In comparison, pure software solution without RLDRAM support only achieves less than 50Mbps performance. The low performance reflects that: random access to large data structure DFA through the L2/DRAM memory hierarchy is severely inefficient.

D. Summary

From all these tests we can conclude that:

- DFA generated by real-life rules cannot be stored in low-latency memory due to the huge DFA size.
- Without low-latency memory support, the performance of DFA is very poor.
- WM algorithm only works well with larger *min_length*.

Therefore, only with the proposed hybrid scheme, multi-Gbps pattern matching performance can be archived on the OCTEON3860 network processor.

IV. CONCLUSION

In this paper, we propose a hybrid pattern matching algorithm optimized for multi-core network processing platforms. As a system-level solution, our approach focuses on: performance stability and hardware/software co-design. To verify the effectiveness of our solution, the proposed algorithms are implemented on a 16-MIPS-core network processing platform and evaluated with real-life data sets. Experimental results show that, when compared with Aho-Corasick algorithm, our hybrid solution saves 60~95% memory space while guarantees stable performance on large pattern sets. Our future work is to optimize more network algorithms for multi-core platforms, including range matching [8] and regular expression matching [9]. We hope all these algorithms can effectively work together to achieve high-performance integrated network applications.

ACKNOWLEDGMENT

This work was supported by National High-Tech R&D 863 Program of China under grant No. 2007AA01Z468.

REFERENCES

[1] http://www.snort.org/
[2] http://www.clamav.net/
[3] http://l7-filter.sourceforge.net/
[4] A. V. Aho, M. J. Corasick, Efficient string matching: An aid to bibliographic search, Communications of the ACM, 1975.
[5] S. Wu and U. Manber, A fast algorithm for multi-pattern searching, Technical Report TR-94-17, University of Arizona, 1994.
[6] http://www.cavium.com/OCTEON_MIPS64.html
[7] http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm
[8] Y. Qi, L. Xu, B. Yang, Y. Xue, J. Li, Packet classification algorithms: from theory to practice, Proc. of the 28th IEEE INFOCOM, 2009
[9] M. Becchi, P. Crowley, Efficient regular expression evaluation: theory to practice, Proc. Of the ACM/IEEE ANCS'08, 2008.