Hybrid Regular Expression Matching for Deep Packet Inspection on Multi-Core Architecture

Yan Sun, Haiqin Liu, Victor C. Valgenti, and Min Sik Kim
School of Electrical and Computer Engineering
Washington State University
Pullman, Washington, U.S.A.
Email: {ysun,hliu,vvalgent,msk}@eecs.wsu.edu

***Abstract*—Many network security applications in today's networks are based on deep packet inspection, checking not only the header portion but also the payload portion of a packet. For example, traffic monitoring, layer-7 filtering, and network intrusion detection all require an accurate analysis of packet content in search for predefined patterns to identify specific classes of applications, viruses, attack signatures, etc. Regular expressions are often used to represent such patterns. They are implemented using finite automata, which take the payload of a packet as an input string. However, existing approaches, both non-deterministic finite automata (NFA) and deterministic finite automata (DFA), have limitations; NFAs have excessive time complexity while DFAs have excessive space complexity. In this paper, we propose an efficient algorithm for regular expression matching to implement deep packet inspection on multi-core architecture. A regular expression is split into NFA-friendly components and DFA-friendly components, which are then assigned to different cores. This hybrid method combines the merits of NFA and DFA implementations, and efficiently takes advantage of multi-core architecture. We evaluate our algorithm using rule sets provided by Snort, a popular open-source intrusion detection system. The simulation results show that our approach outperforms existing NFA/DFA and hybrid approaches. Furthermore, our algorithm performs well on the important issues on multi-core architecture design, such as load balancing, data locality and communication between cores.**

I. Introduction

Many network security applications in today's networks are based on deep packet inspection, checking not only the header portion but also the payload portion of a packet. Traffic monitoring, layer-7 filtering, and network intrusion detection all require an accurate analysis of packet content in search for predefined patterns to identify specific classes of applications, viruses, etc. Those patterns were traditionally a number of strings representing signatures to be compared against packet contents using exact matching algorithms. However, exact matching is not expressive enough to detect malicious patterns with the evolution of the network threats and evasion techniques. Thus, more expressive regular expressions are used to describe a wide variety of payload signatures. For example, Snort [1], an open-source network intrusion detection system (NIDS), has evolved from no regular expressions in its rule set in April 2003 to 3,737 unique Perl Compatible Regular Expressions (PCRE) as of November 2009.

The most popular method to implement regular expression matching is to use finite automata [2]–[5]. In this method, a finite automaton is built based on given regular expressions, and is run with packet payload as input. The finite automaton is either deterministic or non-deterministic, depending on underlying technologies and available resources. A non-deterministic finite automaton (NFA) requires as many state transitions per character in the payload as the number of states in the worst case. However, it is very efficient in terms of space usage compared to a deterministic counterpart. These properties make NFAs more suitable for ASIC (application-specific integrated circuit) or FPGA (field-programmable gate array) implementations, which can provide wide bandwidth but small amount of on-chip memory. On the other hand, a deterministic finite automaton (DFA) requires only one transition per character, while it needs a much larger amount of memory. Therefore, DFAs are more suitable for general-purpose processors and network processors.

Nowadays, most processor vendors are increasing the number of cores in a single chip [6], [7]. The multi-core trend is observed not only in general-purpose processors but also in embedded processors [8], [9], such as network processors, digital signal processors, and cores embedded in FPGAs and GPUs. Furthermore, the number of cores in a single processor continues to increase. The performance gain achieved by the use of a multi-core processor depends on the algorithms used and their implementations in software. Hence, how to design efficient algorithms and implement them to take advantage of available parallelism in multi-core processors receives more attention. Since deep packet inspection is often a bottleneck in packet processing, exploiting parallelism in multi-core architecture is a key to improving overall performance. Existing approaches are focusing on Finite Automata compression or simply divide rule sets into several groups and build corresponding DFAs, and all these methods can not efficiently take advantage of multi-core architecture such as load balancing, data locality and communication between cores.

In this paper, our contributions include: we first analyze the problems we are facing in regular expression matching based on our experiments. Second, we propose an efficient hybrid algorithm for regular expression matching to implement deep packet inspection on multi-core architecture. Last, we analyze our algorithm based on the important factors of multi-core architecture.

The remainder of the paper is organized as follows. In

978-1-4244-7116-4/10/$26.00 ©2010 IEEE

Section II, related work in regular expression matching is presented. Section III provides a discussion on using finite automata for regular expression matching, and Section IV describes our algorithm and its implementation. Then, the proposed algorithm is evaluated in Section V. Finally, we conclude in Section VI.

II. Related Work

Nowadays, regular expressions are the language of choice in NIDS from commercial products such as 3Com TippingPoint X505 [10] and Cisco IPS [11]. Regular expression matchers are typically implemented using finite automata, either NFA or DFA. We divided the approaches to regular expression matching into the following three categories:

a) ASIC-based: Several commercial network equipment vendors, including 3Com [10] and Cisco [11] have supplied their own NIDS, and a number of smaller players have introduced pattern matching ASICs which go inside these NIDS. Developing ASICs for NIDS, however, has several disadvantages; it requires a large investment and a long development cycle, and it is hard to upgrade.

b) FPGA-based: There is a body of literature advocating FPGA-based pattern matching [12]–[15]. It can provide not only fast matching cycle but also parallel matching operations. NFAs are well-suited for FPGA-based matching because of its wide bandwidth requirement and low memory consumption. This type of approaches, however, is not flexible enough for general-purpose regular expression matching.

c) Software-based: The software-based approaches are also called general-purpose approaches, and they are based on general-purpose processors or network processors [3], [16]–[19]. Our work falls into this category. DFAs are more popular in software-based approaches because they only need one state transition per input character, which causes at most one memory access for each character input. Therefore, they are often desirable at high network link rates. However, As we mentioned earlier, the practical use of DFAs is limited because of their excessive memory usage. In order to mitigate this issue, many methods have been proposed [16]–[20]. They develop several compression techniques for DFAs, focusing on reducing the number of transitions between states, and in some cases, 99% transitions can be eliminated. Although this can reduce the memory consumption significantly, unfortunately it is hard to reduce the number of states in DFAs with complex regular expressions. Becchi et al. [3] explored to use hybrid DFA/NFA, which consists of a head DFA and multiple tail NFAs/DFAs, and the tails will take multiple cycles to process one character, this approach achieves better performance than pure NFAs or DFAs. All these algorithms are not focus on how to efficiently utilize multi-core architecture to improve throughput.

III. Regular Expression Matching and Finite Automata

A network intrusion detection system (NIDS) classifies packets using a predefined rule set to determine whether packets are malicious or not by searching packet payloads for any signature in the rule set. Because of the increasing amount of network traffic and threats, intrusion detection systems become very resource-intensive. For instance, open-source NIDSs such as Bro [21] and Snort [1] expend all the resources, both CPU time and memory, and halt immediately when they are deployed under high-speed network environment [22]. Therefore, achieving high-throughput in pattern matching and reducing memory access frequency are crucial for overall intrusion detection performance. In this section, we examine NFA and DFA implementations for regular expression matching, which become basis of the hybrid approach we propose in Section IV. Note that in this paper we discuss pure NFA or DFA implementations that handle traditional regular expressions. Additional features such as back-references should be handled by extending states as suggested in [23].

A. Types of Regular Expression Components

In order to increase parallelism in regular expression matching, we first need to study types and characteristics of regular expression components. Although the regular language itself is a well-defined and well-understood language, there are many variants adopting additional notations to make the language more human-friendly. In this section, we consider some of the main types of regular expression components frequently found in Snort rule sets. In the following, we present them in the increasing order of their complexity.

1) Exact-match strings are the simplest kind of regular expressions. They are fixed-size patterns, and thus the number of states in a finite automaton (DFA or NFA) can be kept less than the number of characters in the regular expression (string). The Aho-Corasick algorithm [24] or the Boyer-Moore algorithm [25] can be used without modification, and hashing can be used for optimial performance. However, this type of regular expressions is not expressive enough, and cannot detect malicious packets if an attacker inserts padding in them. So, the percentage of this kind is dropping fast.
2) Character sets and single wildcards such as "`[`c_i`-`$c_j c_k$`]`" and "`.`". For this type of regular expressions, the exact-match algorithms such as the Aho-Corasick and Boyer-Moore algorithms or hashing schemes cannot be used directly. Instead, exhaustive enumeration of exact-match strings should be used. These regular expressions are more expressive than exact-match strings, but require larger finite automata with more transitions.
3) Simple character repetitions such as "c`?`", "c`*`", and "c`+`". For this type of regular expressions, exhaustive enumeration of exact-match strings are inapplicable because the length of a matched string may be infinite. However, it can be efficiently implemented as a loop transition in a finite automaton.
4) Bounded repetitions such as "c`{n,m}`", "`[^`c_i`-`c_j`]{`n`,}`". For this kind of regular expressions, the number of states of a finite automata grows fast as the counting constraints increase. However, we

can introduce counters as an augmentation to finite automata to alleviate this problem.

5) Character sets and wildcards repetitions such as "`[^`c_i`-`c_j`]*`", "`.*`", "`[^\r\n]*`". If multiple such regular expressions are implemented as a single DFA, the size of the DFA can grow exponentially. We demonstrate it through experiments in Section III-B.

In practice, most regular expressions in NIDS have more than one kind of regular expression patterns mentioned above in a single regular expression.

B. Size of Finite Automata with Complex Regular Expressions

With the evolution of network threats and evation techniques, the length and complexity of regular expressions in rule sets are increasing fast. Building a DFA of a set of complex regular expressions potentially results in an exponential number of DFA states and exponential build time. To investigate the impact of the complexity of regular expressions on the size of finite automata, we perform experiments with hundreds of real attack signatures.

In our experiments, we implement the NFA-based algorithm and DFA-based algorithm proposed by Ficara et al. [16]. Regular expressions are selected from Snort rule sets. Since the goal of these experiments is to understand the effect of complex regular expressions on finite automata, we use the last type of rules defined in Section III-A, which can cause exponential growth of the size of build time of finite automata. So, the selected regular expressions are long and complex, containing many wildcard repetitions. Note that this type of regular expressions exhibits an increasing trend in terms of the number of rules in NIDS. The other types are not included because they can be implemented efficiently either by augmenting finite automata or by other algorithms without finite automata. All the experimental results reported in this paper were obtained on an Intel 3.0 GHz Dual-Core machine with 4 GB main memory.

We first measure the size of NFA for different rule sets to estimate space complexity. We choose three sets of regular expressions from Snort rule sets, calling them R_1, R_2, and R_3. They contain 98, 185, and 298 regular expressions, respectively. For comparison with simple regular expressions, we also create three rule sets, E_1, E_2, and E_3, each of which has the same number and average length of exact-match regular expressions as the corresponding R_i.

Table I shows the number of states and the number of transitions for each configuration. For completeness, each column contains the number before applying the compression algorithm by Ficara et al., as well as the number after compression. For visual comparison between the simple regular expression sets, E_i, and the complex sets, R_i, we plot the numbers in Figures 1 and 2.

We can see from those Figure 1 that, in NFA, the number of states grows proportionally to the number of regular expressions, and there is only a slight difference between the simple regular expressions and the complex regular expressions. On the other hand, the difference between the numbers of transitions is much larger, as shown in Figure 2. This result indicates that the copmlexity of regular expressions mainly affect the number of transitions, not the number of states.

In Table II, we compare the build time of NFA and DFA. The build time of NFA is expected to exhibit the linear tendency as we observed in Figures 1 and 2, because

TABLE I
NUMBER OF NFA STATES AND TRANSITIONS

Rule set	# of states before/after compression	# of transitions before/after compression
E_1	2746/1387	3596/1892
R_1	2348/1206	15006/7220
E_2	6044/3024	7124/3549
R_2	5136/2631	28756/13398
E_3	9554/4683	11264/5797
R_3	8048/4176	41155/20014

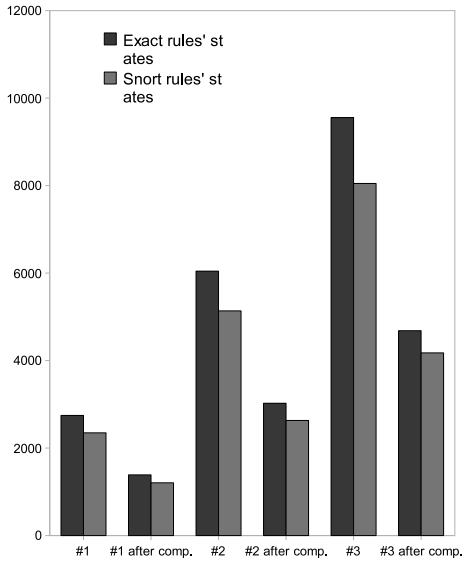

Fig. 1. Number of NFA states

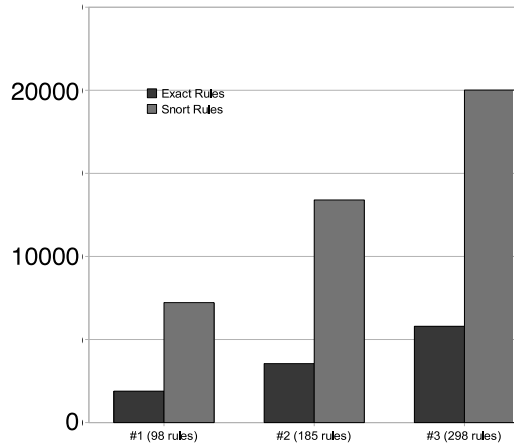

Fig. 2. Number of NFA transitions

TABLE II
BUILD TIME OF NFA AND DFA

Rule set	Build time of NFA (sec)	Build time of DFA (sec)
E_1	0.36	1.98
R_1	0.81	103.98
E_2	1.24	4.57
R_2	1.93	432.56
E_3	2.08	9.05
R_3	3.93	2928.92

TABLE III
STATES COMPARISON BETWEEN NFA AND DFA

Rule set	# of states in NFA	# of states in DFA
E_1	1381	1370
R_1	1207	16673
E_2	3024	3012
R_2	2630	64297
E_3	4781	4779
R_3	4176	$>$ 200000
R_3 with 3 DFAs	–	47235

creating states and transitions is the main task in building finite automata. The first column of Table II confirms this. However, the second column shows a very different trend. For DFAs, while the build time of the simple regular expressions grows linearly with the increase of the number of expressions, the build time of the complex regular expressions grows exponentially with the increase of the number of regular expressions. Therefore, it would be desirable to avoid DFA-only implementations for today's complex rule sets.

We also compare the number of states between NFA and DFA. The results are shown in Table III, and the corresponding chart is presented in Figure 3.

Table III and Figure 3 show the exponential growth of DFA more clearly. While NFA still demonstrates linear growth, regardless of the complexity of regular expressions, the number

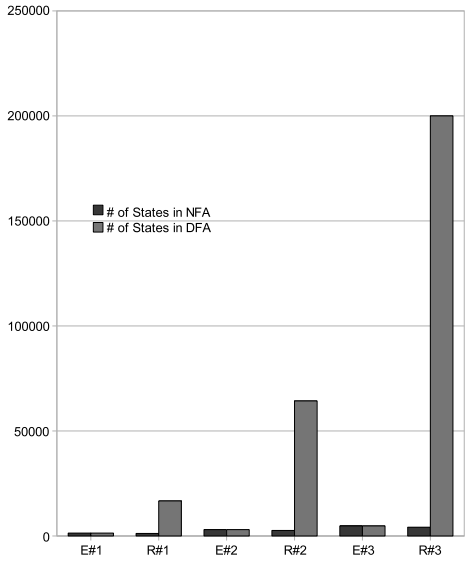

Fig. 3. States comparison between NFA and DFA

of states of a DFA exceeds 200,000 for R_3, which has fewer than 300 rules.

Note that the last row in Table III. It corresponds to a finite automaton implementing R_3 with three DFAs. It is obtained by dividing R_3 into three components, building a DFA for each component, and then combining them using a NFA. The numbers of states of DFAs are 9304, 15782, and 22149, respectively. This means that we can use several small DFAs to replace a single huge DFA to reduce the number of states. Of course, such an approach will increase the memory bandwidth requirement, resulting in more memory accesses when there is not enough bandwidth.

Furthermore, the number of complex regular expression components, such as Kleene Stars, which lead state explosion are increasing fast in the rule sets and each rule contains multiple such components in the latest Snort rule set. For example, we divide the Snort rule set in December 2009 into several groups based on the header information and the top three groups are shown in Table IV. And we can see that about 6 Kleene Stars in each rule in average.

To summarize, we observe in the experiments that DFA is more efficient for the regular expressions with less complexity, such as exact-match, character sets, and simple character repetitions. For more complex regular expressions, however, NFA is more efficient and the cost of DFA implementation is prohibitive. Furthermore, the regular expressions are becoming more and more complex, and each rule may contains multiple points which can lead to state explosion in DFAs. These findings lead us to a hybrid approach based on multi-core architecture, which is presented in Section IV.

IV. HYBRID APPROACH

A. DFA-Friendly Regular Expressions and NFA-Friendly Regular Expressions

For better utilization of multi-core architecture, we need to split regular expressions in an efficient manner. As we observed in Section III, a regular expression can be implemented using multiple DFAs combined with a NFA. To apply this idea, we need to be able to tell which part of a regular expression is better implemented as a DFA and which as a NFA. These are what we call DFA-friendly regular expressions and NFA-friendly expressions, explained in the following.

1) DFA-friendly regular expressions: Regular expressions suitable to be implemented as a DFA. Such regular expressions must keep the number of states and build time of DFAs reasonably small. Implementing them as a NFA would not bring any gain in terms of memory bandwidth.
2) NFA-friendly regular expressions: Regular expressions suitable to be implemented as a NFA. The DFA implementation of such regular expressions will have an exponentially large number of states and build time. By implementing them as NFAs, memory consumption can be reduced significantly.

Actually, there is no clear boundary between these two groups of regular expressions. When we categorize regular

TABLE IV
TOP THREE RULE GROUPS IN SNORT

Protocol	Source IP	Dest IP	Source Port	Dest Port	Rule Number	Kleene Star Number
tcp	EXTERNAL_NET	HOME_NET	HTTP_PORTS	other	675	4584
tcp	EXTERNAL_NET	HOME_NET	other	other	551	3098
tcp	HOME_NET	EXTERNAL_NET	other	HTTP_PORTS	447	480

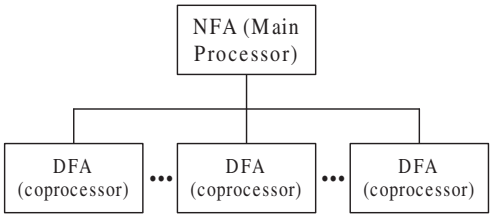

Fig. 4. Hybrid architecture of deep packet inspection on multi-core platform

TABLE V
AN EXAMPLE OF THE HYBRID APPROACH

Regular Expression	DFA-friendly	NFA-friendly
`ab.*cde.*fghi`	⟨1⟩`ab`	⟨1⟩`.*`⟨2⟩`.*`⟨3⟩
`jkl.*mn`	⟨2⟩`cde`	⟨4⟩`.*`⟨5⟩
`opq.*rst`	⟨3⟩`fghi`	⟨6⟩`.*`⟨7⟩
`uvwxyz`	⟨4⟩`jkl`	
	⟨5⟩`mn`	
	⟨6⟩`opq`	
	⟨7⟩`rst`	
	⟨8⟩`uvwxyz`	

expressions in practice, we also need to consider many other factors such as the actual regular expression set and payload balance between cores.

B. Proposed Multi-Core-Based Algorithm

The hybrid approach we propose in this paper is based on the following observations in our experiments:

1) For simple regular expressions, DFA is more efficient (DFA-friendly). Such regular expressions include exact match, character sets, and simple character repetitions.
2) For complex regular expressions, NFA is more efficient (NFA-friendly). Such regular expressions include wildcards repetitions.
3) Majority of packets match only short prefixes of regular expressions.
4) In typical complex Regular expressions, a DFA-friendly expression and a NFA-friendly expression appear in an alternating manner as in "`abc.*defg.*hijklmn.*opq`".
5) Both the number of Kleene Stars in a single rule and the total number of Kleene Stars in the rule sets are increasing fast, and this trend forces us to explore new algorithms to deal with these points because it will degrade existing approaches' performance. For example, more Kleene Stars will be included in the tails in [3], then the tails will becomes much bigger when converted into DFAs, and this will decrease the performance significantly.

The first step in our approach is to identify all DFA-friendly components (substrings), $d_1, d_2, \ldots, d_m$, from the regular expressions included in a given rule set. Each DFA-friendly substring d_i is converted into a DFA D_i, which works as a coprocessor in our implementation.

Every instance of d_i in the original rule set is replaced with a reference to D_i. Then this modified rule set is converted into a single NFA, which becomes the main processor, as shown in Figure 4. Since all DFA-friendly substrings were removed, we expect this NFA is of reasonable size.

For better utilization of multi-core architecture, we identify the followings as key issues in designing the algorithm.

1) Memory access patterns of cores are critical in multi-core architecture. Different cores may use the same area in the main memory, and the frequency and locality of memory accesses will affect the overall performance significantly. Thus, each core should minimize the number of accesses to shared contents in the main memory, which may cause bandwidth contention and consistency problems.
2) Even with a single core, increasing locality is an important factor in reducing memory access latency. In multi-core architecture, this becomes more crucial because higher cache miss rates of multiple cores will aggravate bandwidth contention, resulting in poor performance.
3) To minimize idle time, the load should be evenly distributed among cores.

The architecture in Figure 4 addresses all of these issues. First, each processor uses its own data (states and transitions). Second, because of the reduced memory consumption demonstrated in Table III, the overall cache hit ratio should be higher than DFA-only or NFA-only approaches. Third, although the main processor (NFA) needs more memory bandwidth, which may decrease the overall performance, for most of the time a branch of the NFA is waiting for coprocessor's responds (DFA matching), or doing repetition on the same state, or being inactive, none of which consume memory bandwidth. Furthermore, we find that the size of the NFA in the main processor is small enough for most rule sets that the states and transitions can reside in its own cache. All of these contribute to reducing memory accesses by the main processor, alleviating memory bandwidth contention. Finally, for even distribution of loads, we may classify character sets and wildcards repetitions as DFA-friendly regular expressions.

A simple example for this approach is illustrated in Table V. Four original regular expressions are used in this example, and eight DFA-friendly substrings are identified and converted

into DFAs (left-hand side). The remaining parts of regular expressions are converted into a NFA (right-hand side).

V. Evaluation

To evaluate the proposed hybrid approach, we build a single NFA as the main processor and build a single DFA as the coprocessor using the 298 rules in the R_3. We compare a single-DFA approach, a single-NFA approach, and our hybrid approach in terms of memory consumption (state number) and processing speed (clock cycles/character).

We make the following assumptions for evaluation.

- Each core can only fetch a single state or transition per memory access.
- Each core's cache can store 2000 states or transitions.
- A cache hit and a cache miss need 2 and 30 clock cycles, respectively,
- Five percent of branches in the NFA are active all the time and these active branches may access memory at the same clock cycle.
- The average length of DFA-friendly substrings is about 10 times of the average length of NFA-friendly ones.
- The average number of repetitions is 30 for each NFA-friendly substring.

Note that the last two items conform to what we observe in Snort rule sets.

Based on the assumptions above, we choose the boundary between the DFA-friendly and NFA-friendly regular expressions for a rule set as follows. We first define the value V, the weighted number of states in DFA and NFA as in the following equation:

$$V = D + \alpha N \tag{1}$$

where D and N are the numbers of states in DFA and NFA, respectively, and α is a parameter that depends on the memory bandwidth or additional memory accesses needed by NFA. In our experiments, we set $\alpha = 5$. We classify regular expressions into 5 types as we presented in Section IV. We use Types "1–2" to represent the boundary between the first two categories, "2–3" between the second and the third, and so on. The results are shown in Figure 5. Since the values of V for 0–1 and 5–0 are much larger than others, so we chose 4–5 as the boundary between the DFA-friendly and NFA-friendly in our simulation.

The numbers of states are shown in Figure 6. The single-DFA has more than 200,000 states, the single-NFA has 4176, and the hybrid approach has the fewest, 4093, which consists of 3110 DFA states and 983 NFA states.

The clock cycles per character is plotted in Figure 7. The single-DFA (running on a single core) achieves 29.72 cycles/character with cache hit rate 1%, the single-NFA (running on a single core) 80.3 with hit rate 47.89%, and the hybrid approach achieves 23.98 ($= 11.99 \times 2$) cycles/character with hit rate 64.31% for DFA, and almost 100% for NFA. For fair comparison, we double the clock cycles used for each matching character.

From the evaluation results we can see that for the complex rule set, a DFA consumes too much memory while a NFA is slow due to multiple active states accessing memory in parallel. Our algorithm outperforms these approaches not only in memory consumption but also in processing speed.

We simulate our algorithm on four-core chip architecture using the previous three rule sets. We use core 0 works as the master and deals with the NFA and core 1, core 2 and core 3 work as the slaves and deal with DFAs. We allocate the new DFA-friendly rules to the three cores through Round-robin scheduling. Because this can come up with three benefits: first, there is no shared data between DFA cores because each core has its own rule set, which ensures better data locality higher cache hit rate. Second, there is no communication between DFA cores. Last, multiple new DFA-friendly rules extracted from the same original rule will be allocated into different cores, which can provide good load balancing on different cores and the DFA cores will communicate with the NFA core at different time during a complete original rule match to alleviate internal bus contend between cores. Based on different rule sets, the numbers of states and numbers of transitions in the three DFA cores are are shown in Table VI.

From the simulation results we can see our algorithm outperforms the algorithm proposed in [3] in terms of total number of states and number of transitions, which indicates that our approach consumes less memory.

VI. Conclusion and Future Work

In this paper, we studied techniques used in deep packet inspection, in particular regular expression matching with growing complexity. We built efficient NFA/DFA generators and investigated the impact of the complexity of regular expressions using rule sets from actual NIDS. We demonstrated that a singe NFA or a singe DFA performs poorly given a large set of complex regular expressions. Because multi-core processors are expected to dominate, there will be great demand on fast deep packet inspection algorithms that can utilize multi-core architecture for complex regular expressions. Thus, we proposed a hybrid algorithm that combines both NFA and DFA. It divides a complex regular expression and configures them into multiple cores to take advantage of available parallelism provided by multi-core processors. The evaluation results show that the hybrid approach outperforms existing DFA/NFA and hybrid approaches. With the advent of processors with more embedded cores, achieving maximum parallelism will be crucial. We plan to compare our approach with other types of parallelization, e.g., building multiple DFAs and running them on separate cores [26], or running NFA on SIMD processors [27]. We also plan to expand this work to general traffic monitoring systems with pipelining architecture.

References

[1] *Snort User Manual 2.8.6*, The Snort Project, Apr. 2010, http://www.snort.org/assets/140/snort_manual_2_8_6.pdf.

[2] M. Paolieri, I. Bonesana, and M. D. Santambrogio, "ReCPU: a parallel and pipelined architecture for regular expression matching," in *Proceedings of 15th Annual IFIP International Conference on Very Large Scale Integration*, Oct. 2007, pp. 19–24.

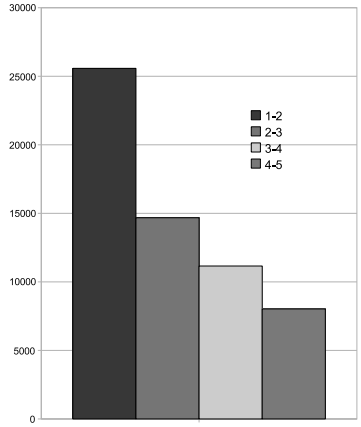

Fig. 5. V for differet boundaries

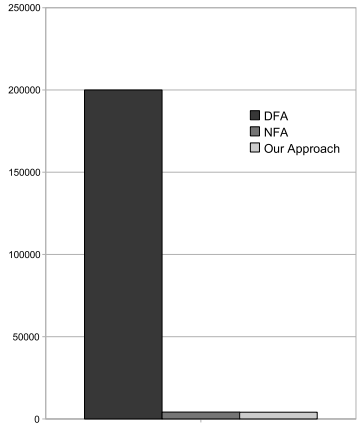

Fig. 6. Memory consumpton

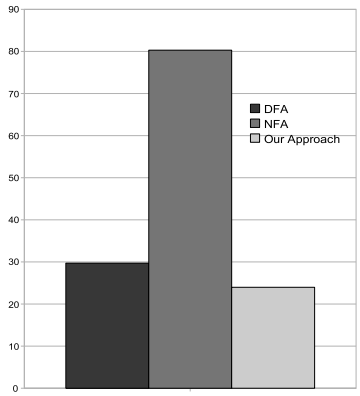

Fig. 7. Evaluation on processing speed

TABLE VI
NUMBER OF STATES AND TRANSITIONS IN DIFFERENT CORES

Rule set	# of State/Trans. in core 0 (NFA)	# of State/Trans. in core 1 (DFA)	# of State/Trans. in core 2 (DFA)	# of State/Trans. in core 3 (DFA)	# of State/Trans. in head in [3]	# of State/Trans. in tails in [3]	# of Tails in [3]
R_1	27/66	241/688	297/881	124/431	2337/8673	48/86	2
R_2	119/154	382/1130	463/1158	229/563	8250/31942	102/245	4
R_3	206/268	667/2058	704/2246	562/1638	15435/69372	247/508	7

[3] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of ACM CoNEXT*, Dec. 2007.

[4] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proceedings of the 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, Dec. 2007, pp. 155–164.

[5] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, Dec. 2006, pp. 339–350.

[6] "Intel multi-core technology," http://www.intel.com/multi-core/.

[7] "Multi-core processors: The next evolution in computing," http://multicore.amd.com/Resources/33211A_Multi-Core_WP_en.pdf.

[8] "Coherent Processing System (CPS): Multi-threaded multiprocessor IP cores," http://www.mips.com/.

[9] "The ARM11 MPCore is synthesizable," http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[10] "TippingPoint x505," http://www.tippingpoint.com/products_ips.html.

[11] "Cisco IOS IPS signature deployment guide," http://www.cisco.com/.

[12] I. Bonesana, M. Paolieri, and M. Santambrogio, "An adaptable FPGA-based system for regular expression matching," in *Proceedings of the conference on Design, Automation and Test in Europe*, Mar. 2008, pp. 1262–1267.

[13] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *International Conference on Field Programmable Logic and Applications*, Sep. 2008, pp. 131–136.

[14] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," in *Conference on field-programmable logic and applications*, Sep. 2003, pp. 956–959.

[15] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, 2007, pp. 127–136.

[16] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved DFA for fast regular expression matching," *Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.

[17] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, 2007, pp. 145–154.

[18] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, 2006, pp. 81–92.

[19] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proceedings of the 26th IEEE International Conference on Computer Communications*, May 2007, pp. 29–40.

[20] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the 2009 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Oct. 2009.

[21] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23–24, pp. 2435–2463, Dec. 1999.

[22] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Operational experiences with high-volume network intrusion detection," in *Proceedings of the 11th ACM Conference on Computer and Comm. Security*, Oct. 2004.

[23] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proceedings of ACM CoNEXT*, Dec. 2008.

[24] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.

[25] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[26] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Dec. 2006, pp. 93–102.

[27] F. Kulishov, "DFA-based and SIMD NFA-based regular expression matching on Cell BE for fast network traffic filtering," in *Proceedings of International Conference on Security of Information and Networks*, Oct. 2009.