IEICE TRANS. INF. & SYST., VOL.E93–D, NO.12 DECEMBER 2010

PAPER Special Section on Parallel and Distributed Computing and Networking

Parallel DFA Architecture for Ultra High Throughput DFA-Based Pattern Matching*

Yi TANG[†], ***Student Member***, **Junchen JIANG**[†], **Xiaofei WANG**[††], **Chengchen HU**[†], **Bin LIU**[†a)], ***Nonmembers***, ***and*** **Zhijia CHEN**[†], ***Student Member***

SUMMARY Multi-pattern matching is a key technique for implementing network security applications such as Network Intrusion Detection/Protection Systems (NIDS/NIPSes) where every packet is inspected against tens of thousands of predefined attack signatures written in regular expressions (regexes). To this end, Deterministic Finite Automaton (DFA) is widely used for multi-regex matching, but existing DFA-based researches have claimed high throughput at an expense of extremely high memory cost, so fail to be employed in devices such as high-speed routers and embedded systems where the available memory is quite limited. In this paper, we propose a parallel architecture of DFA called Parallel DFA (PDFA) taking advantage of the large amount of concurrent flows to increase the throughput with nearly no extra memory cost. The basic idea is to selectively store the underlying DFA in memory modules that can be accessed in parallel. To explore its potential parallelism we intensively study DFA-split schemes from both state and transition points in this paper. The performance of our approach in both the average cases and the worst cases is analyzed, optimized and evaluated by numerical results. The evaluation shows that we obtain an average speedup of 100 times compared with traditional DFA-based matching approach.

key words: *Deterministic Finite Automata (DFA), Deep Packet Inspection (DPI), regular expression, parallel matching, speedup*

1. Introduction

Nowadays, Network Intrusion Detection/Protection Systems (NIDS/NIPSes) are deployed to safe-guard the security of network operations. By inspecting the header and payload of a packet against attack signatures, NIDS/NIPSes are capable of discovering the invasion. As most of the known attacks can be represented by a set of regular expressions (regexes), multi-regex matching becomes one of the key components in NIDS/NIPSes design. Meanwhile, more and more security detections or application identifications are performed at embedded systems and network processors where the available memory resource is quite limited. Generally speaking, the two major metrics of matching engines are *throughput* and *memory cost*. Hence, designing a multi-regex matching scheme with increased throughput and quite low memory cost leads to a great challenge both in algorithm design and in hardware implementation.

Traditionally, Deterministic Finite Automaton (DFA) is widely used for multi-regex matching in NIDS/NIPSes for its constant matching speed even in the worst case. However, for byte-at-a-cycle processing, the throughput is strictly proportional to the memory access frequency. For example, an NIDS dealing with 10 Gb/s requires a memory access frequency of at least 1.25 GHz (for 8-bit data bus), which is impractical with the current technologies. To raise throughput, two accepted approaches are pipelining and parallelism. Considering the large amount of backtracking transitions in DFA, pipeline architecture is greatly restrained. Alternatively, exploiting parallelism, including intra-flow and inter-flow parallelism, which checks multiple characters per clock cycle is a promising solution. Current academic researches mostly focus on intra-flow parallelism [1]–[3], they still suffer from the memory explosion or other restraints on increasing throughput.

Approaches utilizing intra-flow parallelism consume multiple characters in one single flow per cycle. However, to ensure that a pattern starts or ends at any position of the flow leads to a huge extra memory consumption (severe transition explosion) and overmuch hardware logic, which limits their scalability. Actually, most of the existing inter-flow approaches can only achieve less than 5 times [1] speedup with reasonable extra memory expense.

Alternatively, exploiting inter-flow parallelism which processes characters of concurrent flows is seemly a more practical approach for accelerating matching procedure. Currently, the magnitude of concurrent flows in network is large [4], [5]. In [4], the experiments on traces from OC-48 links where the statistical time scale is configured as 10 microsecond, shows that the number of concurrent flows is at a level of several hundreds. Meanwhile, with the limitation of hardware chip's pins and available logic resource, the parallel processing units in FPGA or Network processor with external SRAM or DRAM chips will not too much (< 100). Hence, compared with available hardware parallel resource, the flow number is larger, suggesting a potential improvement on accelerating matching through utilizing inter-flow parallelism.

Currently, a commonly agreed upon standard for evaluating inter-flow matching architecture is still missing, leading to unfair comparisons and designs lacking in generality or scalability. For a better discussion, we address the three major metrics of performance as follows:

1. *Low Memory Duplication* indicates that contents are

Manuscript received February 5, 2010.
Manuscript revised June 14, 2010.

[†]The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing, China.
[††]The author is with School of Electronic Engineering, Dublin City University, Ireland.

*This research is partially published in ICC 2010.

a) E-mail: liub@tsinghua.edu.cn

DOI: 10.1587/transinf.E93.D.3232

selectively stored in parallel memories, rather than simply copied in memories, making the memory consumption sublinear to speedup.
2. *High throughput* should be achieved with full utilization on the parallel recourses.
3. *Short Latency* guarantees balanced processing speed among flows being matched in parallel.

Recently developed inter-flow approaches mainly focus on compressing single non-accelerated matching engine and exploiting inter-flow parallelism by quite simple duplication with little compression ratio. Therefore given the memory size, their speedup is strictly bounded by the compression ratio failing to deeply exploit inter-flow parallelism.

Aiming at making the best use of inter-flow parallelism regarding the above criteria and problems, we propose a parallel storage architecture of DFA called *parallel DFA* (PDFA) in this paper which selectively stores the underlying DFA in memory devices which can be accessed concurrently. Different from conventional DFA taking transition as the basic storage unit while transitions of the same state are stored as a whole in hardware, we investigate how to divide DFA using a hierarchical notion: DFA is initially divided into different groups of states and further into groups of transitions. Correspondingly, we give two schemes for states assignment (*SA*) and transitions assignment (*TA*) respectively. For state-level assignment, different states would be assigned into various groups which would be accessed independently during processing. And for transition-level DFA assignment, transitions with different character classes are stored in different memory modules to exploit parallelism. Sharing fundamentally the same idea of storing transition in multiple memories the two schemes can be applied at the same time.

PDFA makes little change to the conventional DFA structure and can be thereby combined with existing compression techniques (see Sect. 2). Multiple flows can be processed simultaneously once their memory accesses have no conflict. Consequently, a high speedup is achievable when matching large amount of concurrent flows, meanwhile its space complexity is no more than one compressed instance. In practice, proper construction of PDFA leads to less memory-access conflict among concurrently processed flows and thereby achieves high overall throughput. In other words, *Low memory duplication and increased throughput shall be both satisfied under PDFA architecture*. We split the DFA into transitions and at one extreme, we can store each of them in each memory module so that any two flows can be concurrently processed without conflict for a higher speedup, while at another extreme, we should store transitions in as less memory modules as possible (in fact, we store most transitions in one memory module) so that the total space consumption is sublinear to the memory number and the management is simpler. Therefore essentially, the devise of PDFA shall balance the two extreme cases. To this end, we bound the speedup (Sect. 4) which would later be shown determined by the distribution of current states, we analyze the probability distribution of current states by treating DFA as a Markov chain. Furthermore, fairness between flows will be taken into account to reduce the latency, since impolitic flow scheduling might cause problems like flow starvation.

Specifically, our contributions are listed as follows:

1. We propose a parallel DFA representation to exploit the inter-flow parallelism with little memory duplication. The speedup in both the average cases and the worst cases is theoretically discussed.
2. Based on a series of theoretical analysis, we propose a fast and effective DFA construction algorithm to fully utilize the inter-flow parallelism. Experimental evaluation shows that more than one hundred times speedup can be achievable with less than one-tenth the memory size compared with conventional approaches.
3. For a good flow scheduling, we devise a preliminary chip design for PDFA where a delicately designed multi-flow scheduling guarantees a balanced inter-flow processing.

The remainder of this paper is organized as follows: Section 2 goes over the previous works. Section 3 revisits the traditional DFA-based approaches, presents our approach by motivating examples and gives the problem statements on specific algorithm design. Section 4 addresses the speedup and Sect. 5 further devises an effective construction algorithm. Section 6 provides the chip design with scheduling policy. Section 7 presents the experimental results and Sect. 8 concludes the paper.

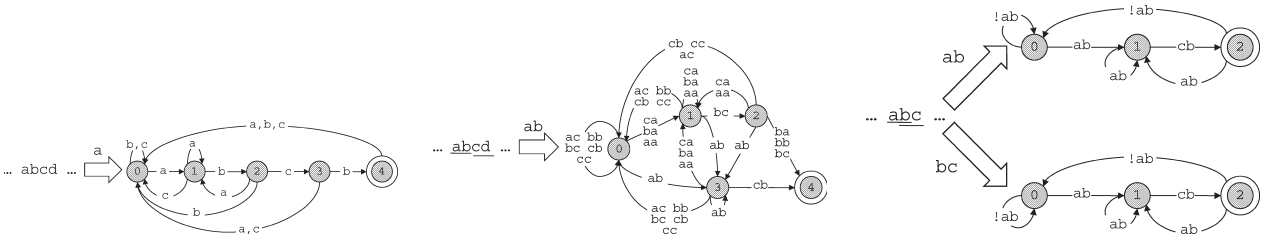
2. Related Work and Background

Many works on accelerating DFA-based pattern (regular expression) matching are proposed in recent years. Generally, they are classified into inter-flow and intra-flow parallelism from the view point of parallel speedup.

Intra-flow parallelism consumes multiple characters of a single flow in each clock cycle. These methods don't take the advantage of huge number of flows. However, there are also some problems:

1. As they don't utilize the characteristics of multiple flows, the extra price for speedup obtained within one flow is expensive, it limits the intra-flow speedup (usually less than 5).
2. Sensible to the length of pattern. Their use of bloom-filter or TCAM as match engine is hard to extend to regex, because the length of matched strings of certain regex rules is implicit. TCAM or bloom-filter would not take ambiguous streams as input.

Two kinds of memory explosions would occur in multi-character based DFA matching: The first is the transition explosion, which is introduced to consume multiple characters (as a block) in each transition. Such algorithms lead to an exponential number ($\Theta(\Sigma^l)$, l is the block length) of



(a) Single-character string matching.

(b) Transition explosion in block-oriented multi-character DFA matching for intra-flow parallel speedup.

(c) Duplication and transition explosion in block-oriented multi-character DFA matching for intra-flow parallel speedup.

Fig. 1 Different string matching schemes for pattern "abcb" on alphabet set {a,b,c}.

transitions in each state. Figure 1 (b) and 1 (c) illustrate the cases where each state has 3^2 transitions. The second is the DFA replication (see Fig. 1 (c), which duplicates DFA into multiple copies to ensure the so-called byte alignment requirement, i.e., every character of the input stream should have the chance to be examined as the first character of the pattern. Since the first character of the pattern can occur in any location of the input stream, the byte alignment problem is inherent in existing block-oriented algorithms.

Furthermore, most proposed intra-flow approaches mainly target on problems of string matching [1]–[3], whose expressiveness is incompetent to NIDS or DPI ruleset. The obstacle of adapting them to regular expression matching involves that (*i*) regular expression does not have the features of string pattern such as explicit length and content (mainly due to wildcards and sparks), so parallel facilities for string matching such as bloom-filter or TCAM become useless; and that (*ii*) the extra memory price for speedup is expensive which limits the scalability of intra-flow speedup approaches.

In the state-of-the-art [6], the authors raise the throughput of regular expression matching by expanding the alphabet set. This produces an exponential increase in memory usage (since the cardinality of the alphabet is now squared). Though they propose several heuristics to mitigate the memory blow-up, the speedup as well as the throughput is very limited.

Inter-flow parallelism accelerates pattern matching through exploiting the parallel recourse. Recently proposed approaches claim a high throughput through simply duplicating and compressing single matching instance. Traditional inter-flow speedup approaches just duplicate the DFA with multiple copies and store in different memory modules for multiple flows matching. In their approaches, the memory storage is overlapping with each other and they ignore the features of different flows.

Existing inter-flow approaches still rest on the duplication of DFA, they focus on compressing single non-accelerated DFA matching engine and utilize inter-flow parallelism through simple duplication. Transition compression approaches achieve a high compression ratio by greatly reducing per state transitions. D^{2}FA [7] acknowledged as the original work in this approach, is proposed to compress DFA by applying default transition, but at a cost of accessing DFA multiple times per input character. Based on it, later works including [8], [9] improve the worst case performance.

State compression technique is first utilized in [10], where pattern grouping is introduced to deflating state explosion. [11] performs a partial NFA-to-DFA conversion to prevent state explosion. The state-of-the-art work named XFA [12], [13] uses auxiliary memory to reduce the DFA state explosion and achieves a great reduction ratio. However, it is not suitable for real-time network applications for its significant startup overhead. These above mentioned DFA-based works can be easily employed in our approach with the similar compression ratios since their fundamental structures can be completely preserved. Our proposed method does not conflict with above works. The method of PDFA also can be exploited on the compressed DFA with above approaches. As the average speedup of the proposed parallel architecture is mainly influenced by the distribution of current states. So most DFA compression approaches will not affect the throughput of our proposed method.

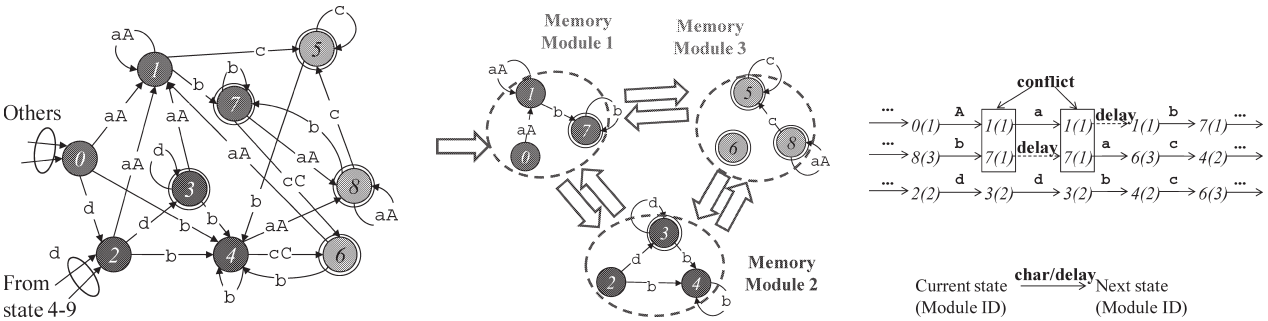
3. Exploiting DFA Parallelism

In this section, we introduce two schemes of exploiting parallelism, state assignment scheme (*SA scheme*) and transition assignment scheme (*TA scheme*). We begin with two motivation examples, and further give the problem statement of DFA matching speedup.

3.1 State Assignment (SA) Scheme

We depict a standard DFA recognizing pattern set {[aA]+b+,[aA]+c+,b+[aA]+,b+[cC],dd+} over input character set {a,A,b,B,c,C,d,D} in Fig. 2 (a). Traditional DFA-based approach is implemented by storing the depicted DFA in a single memory module. During the processing, one current state is maintained within the memory module for each current flow, and one byte can be processed per cycle by accessing the memory module once to update the corresponding current state. However, when matching multiple input flows, conventional storage architecture of DFA has to process them sequentially, since the updating of each flow's current state needs a memory access.

Alternatively, Fig. 2 (b) shows an instance of our stor-

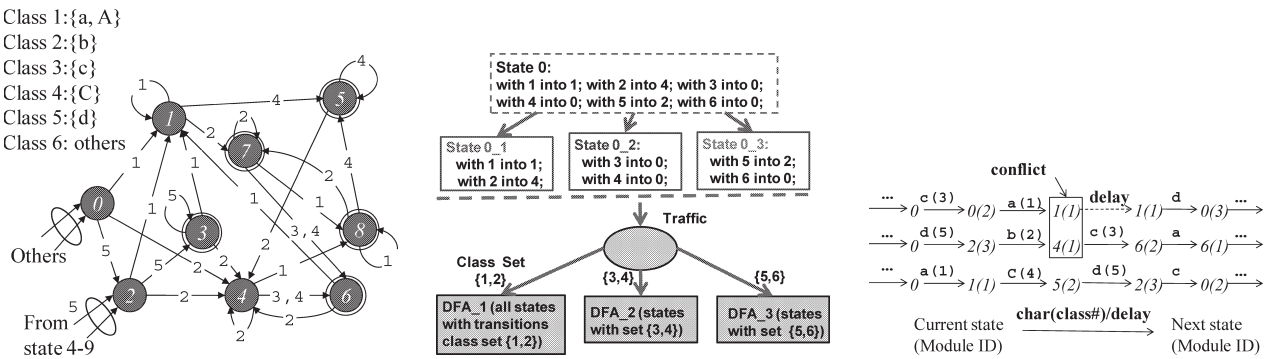


(a) A complete DFA of pattern set {[aA]+b+, [aA]+c+, b+[aA]+, b+[cC], dd+} over input character set {a,A,b,B,c,C,d,D} (transitions to state 0 is ignored for simplicity).

(b) Storing all states in $k = 3$ modules together with their transitions.

(c) The process of matching three flow segments $STR_1 = \ldots$ Aab $\ldots$, $STR_2 = \ldots$ bac $\ldots$, $STR_3 = \ldots$ ddbc $\ldots$.

Fig. 2 The State Assignment's motivating example and its implementation.



(a) A complete DFA of pattern set {[aA]+b+, [aA]+c+, b+[aA]+, b+[cC], dd+} over input character set {a,A,b,B,c,C,d,D} (transitions to state 0 is ignored for simplicity).

(b) Storing and visiting each state on different memory modules according to transition classes, here give the splitting sample on state 0.

(c) The process of matching three flow segments $STR_1 = \ldots$ cad $\ldots$, $STR_2 = \ldots$ dbca $\ldots$, $STR_3 = \ldots$ aCdc $\ldots$.

Fig. 3 The Transition Assignment's motivating example and its implementation.

age scheme, which stores states in independent memory modules with no duplication so that the total memory cost keeps unchanged. When processing k concurrent flows, it maintains k current states, and each of the states is updated by processing one character of the corresponding flow. The memory modules containing at least one current state are called *active module*s. In each cycle, an active module is able to process one current state, and the processed flow is called *active flow*. So the number of active flows is equal to the number of active modules. As an example, Fig. 2 (b) demonstrates one way to store states in three memory modules. In Fig. 2 (c), we illustrate the matching process on three flow segment STR_1, STR_2, STR_3. Initially, three current states are 0, 8 and 2, located in different modules, so a full speedup is obtained in the first cycle. In the second cycle, access conflict occurs when updating current states of STR_1 and STR_2 (state 1 and 7 are both in module one), so one of them is chosen to be processed and therefore the second cycle achieves a speedup of two. During the four cycles shown in Fig. 2 (c), our approach processes ten bytes, compared with the four bytes processed by the traditional method.

3.2 Transition Assignment (TA) Scheme

In this section, we will exploit the parallelism between transitions by assigning different transitions to given memory modules, called TA scheme, for processing enhancement. As the Fig. 3 shows, the above mentioned sample pattern set {[aA]+b+,[aA]+c+,b+[aA]+,b+[cC],dd+} over input character set {a,A,b,B,c,C,d,D} is given here. Our TA scheme is based on a classical technique called alphabet set classification whose original form is introduced as follows. Typical DFA (e.g., Fig. 2 (a)) has 256 next states for each state, since there are 256 characters in alphabet for ASCII. In practice, DFA generator unquestionably does not hold 256 entries for each state; instead, for most generators like JFlex [14], they perform classifications for transitions so that transitions with same character would be classified into one *class* shown in Fig. 3 (a). For instance, 'A' and 'a' have the same next hop for any states in DFA, so they are classified into one class encoded by 1.

Based on alphabet classification, we divide the transition set of each state into multiple *class-sets*, of which the one encoded i consists of transitions belonging to alphabet

class i. Figure 3 (b) depicts the result of TA scheme on the same standard DFA of Fig. 2 (a). Notice that state 1 has six transitions being preserved in 3 memory modules according to the labeled character on each transition. For any class-set i, TA scheme stores all transitions belonging to it into the same memory module, forming one DFA called sub-DFA where each state has one single transition labeled class i. In this match system, the contents of any two memory modules do not overlap implying that no extra memory cost is introduced for parallel speedup. When receiving one character, the matching system dispatches it into the corresponding sub-DFA labeled with the class containing the character. Sequentially, for multiple flows, they can be matched simultaneously once the incoming characters are not conflicted with each other (conflict when characters belong to the same class-set). Figure 3 (c) illustrates the matching process on three flow segments STR_1, STR_2, STR_3 under TA scheme. Initially, three current states are 0's, the first three input characters are `c,d,a`, they do not conflict with each other causing a full speedup in the first cycle. In the second cycle, `a,b` are in the same class-set, so one of them is chosen to be processed and therefore the second cycle achieves a speedup of two.

In conclusion, we refer to a DFA stored in TA scheme or SA scheme described previously as a *Parallel DFA (PDFA)*. As the motivating example shows, no extra information is stored in PDFA compared with the conventional one. Furthermore, PDFA makes no change to the typical structure of DFA as well as the processing on each flow. For easy implementation, PDFA requires that all transitions of class-set i are stored in one memory module. In other words, if two flows' current characters are of the identical class and then they would be sent into the same sub-DFA stored in one memory module. Also the definition of an active module is formally defined as any memory module that receives at least one character to process.

3.3 Problem Statement

In **PDFA**, the speedup equals to the number of active modules determined by characters in input flows as well as by the allocation of states in different modules. Since we have no control over input flows, our first target is to allocate states, as well as transitions, of the original DFA so that a substantially large amount of active modules would exist in each cycle. Moreover, how to handle the access conflicts between current states within one active module and how to prevent flow starvation make it indispensable to design a flow scheduling for PDFA. We regard the above mentioned two problems to two aspects respectively, where the transition assignment problem as well as the state allocation is decided in *pre-computing phase* while scheduling directly involves the selection among conflicting current states during *running phase*. We resolve the two problems in Sect. 4, 5 and Sect. 6 respectively.

4. Analyzing Speedup

Speedup of the PDFA is mainly influenced by the distribution of current states in different modules and further the distribution of current characters in various classes. Consequently, it is necessary to address the possibility of each state being the *current state* under one single flow (Here *current state* denotes the state being visited after reading a character). Then we will study the expected number of classes containing at least one current character.

4.1 A Brief Markovian Analysis of DFA Under Single Flow

The detailed analysis on the characteristic of traffic flow is not the main focus of this work, so we only provide an brief overview. Note that the current state in DFA is deterministic given the input sequence of characters. For modeling the character sequences in various real-world flows, we chose to consider "language-independent" statistical modeling of data flows best exemplified by the well known n-gram analysis. Generally speaking, given a sequence $x_i, (i = 0, 1, \ldots)$, n-gram predicts x_i based on $x_{i-1}, \ldots, x_{i-n}$. The method is well understood and effective. 1-gram is the most simplified version, and suffices to model the character distribution in most flow environments. In other words, a statistically stationary distribution of characters in any place of flow is guaranteed. For detailed treatment of this subject, the reader can refer to [15].

Given a DFA $(Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, \ldots, q_N\}$, Σ, δ, q_0 and F refer to the state set, the input character set, the transition function, the initial state, and the accept state set respectively. Of a certain input flow $T = t_0, t_1, \ldots$, we denote the current state sequence with $X = x_0, x_1, \ldots$, where $x_0 = q_0$ and $x_{n+1} = \delta(x_n, t_n), n = 0, 1, \ldots$. So the possibility of current state transferring from q_i to q_j in the nth step is $P_{ij}^{(n)} = \Pr(x_{n+1} = q_j | x_n = q_i) = \sum_{c:q_j=\delta(q_i,c)} \Pr(t_n = c)$.

Based on the flow modeling, we can assume the stationary distribution $\Pr(t_n = c), n \in \mathbb{Z}^+$ and denote it as $D(c)$. Then $P_{ij}^{(n)} = \sum_{q_j=\delta(q_i,c)} D(c) = P_{ij}$. Consequently, if we consider the DFA as a Markov chain over a finite state machine, the transition possibility $P_{ij}^{(n)}$ between q_i and q_j is independent with n which means that the DFA model here is a time-homogeneous Markov chain. Actually, large DFA can be seen as a nondecomposable Markov chain, whose states are all recurrent states, so a unique limiting probability distribution vector (stationary distribution) $\pi = (P(q_1), \ldots, P(q_N))$ exists, where $P(q_i)$ refers to the probability of state q_i being the current state. π satisfies the equation $\pi = \pi M$ where matrix $M = (P_{ij})$ is called transition matrix. For minute discussion on periodicity and decomposability of Markov chain in a large finite state machine, the reader can refer to [16].

4.2 Average Speedup with SA Scheme

The analysis of speedup under multiple flows involves a syn-

thesis upon the probability distribution of current state under single flow. In Sect. 4.1, we address the existence of a unique limiting probability distribution of current state under single flow which is statistically reliable when processing large data flows. As an example in Fig. 2 (a), the limiting distribution of the DFA in the motivating example of Fig. 2 (a) is computed as below (we assume $D(c) = 1/8$ for each input characters c without loss of generality):

$$\pi = \pi M$$

$$= \pi \begin{pmatrix} 4/8 & 2/8 & 1/8 & 0 & 1/8 & 0 & 0 & 0 & 0 \\ 3/8 & 2/8 & 1/8 & 0 & 0 & 1/8 & 0 & 1/8 & 0 \\ 4/8 & 2/8 & 0 & 1/8 & 1/8 & 0 & 0 & 0 & 0 \\ 4/8 & 2/8 & 0 & 1/8 & 1/8 & 0 & 0 & 0 & 0 \\ 2/8 & 0 & 1/8 & 0 & 1/8 & 0 & 2/8 & 0 & 2/8 \\ 3/8 & 2/8 & 1/8 & 0 & 1/8 & 1/8 & 0 & 0 & 0 \\ 4/8 & 2/8 & 1/8 & 0 & 1/8 & 0 & 0 & 0 & 0 \\ 2/8 & 0 & 1/8 & 0 & 0 & 0 & 2/8 & 1/8 & 2/8 \\ 3/8 & 0 & 1/8 & 0 & 0 & 1/8 & 0 & 1/8 & 2/8 \end{pmatrix}$$

$$\pi = (0.433, 0.208, 0.109, 0.015, 0.089, 0.035, 0.031, 0.035, 0.041)$$

where M is the transition matrix. The value in M is calculated as the probability from one state to another state. For example, there are 2 transitions from state 0 to state 1, and the total outgoing transitions from state 0 is 8. Hence, the probability from state 0 to state 1 is 2/8. M stores all states in k distinct memory modules $Q_1, \ldots, Q_k$, then we have

Theorem 1: When matching one single flow, the limiting probability of memory module Q_j containing the current state is $P(Q_j) = \sum_{q_i \in Q_j} P(q_i)$, where $P(q_i)$ is the stationary possibility of state q_i.

It can be inferred that $\sum_{j=1}^{k} P(Q_j) = 1$. Given r flows (r current states), we denote the probability that t current states being state q_i ($i = 1, \ldots, N$) in the nth cycle as $P_t^{(n)}(q_i)$. Then the expectation of the speedup S is

$$E(S) = \lim_{n\to\infty} \sum_{j=1}^{k} \prod_{q_i \in Q_j} (1 - P_0^{(n)}(q_i)) \tag{1}$$

where $1 - P_0^{(n)}(q_i)$ is the expectation of q_i being at least one current state in the nth cycle. In each cycle, any active module can process one current state, and new current states might transmit to it from the other modules (including itself). We consider the number (denoted as n_i) of new coming current states to q_i, then the probability

$$\Pr(n_i = m) = \sum_{Q' \subseteq Q, |Q'|=m} A(Q')B(Q') \tag{2}$$

where $A(Q') = \prod_{q_j \in Q'} (1 - P_0^{(n)}(q_j))P_{ji}$ is the possibility of all states in Q' transmitting their current states to q_i, and $B(Q') = \prod_{q_j \notin Q'} (P_0^{(n)}(q_j) + (1 - P_0^{(n)}(q_j))(1 - P_{ji}))$ is the possibility of all states not in Q' transmit no current state to q_i.

$$P_t^{(n+1)}(q_i) = P_0^{(n)}(q_i)\Pr(n_i = t) + \sum_{s=0}^{t} P_{s+1}^{(n)}(q_i)\Pr(n_i = t - s - 1) \tag{3}$$

According to Eq. (1), (2) and (3), we have the following theorem: $F(E(S)) = E(S) - \sum_{i=1}^{k} \left(1 - \frac{1}{\alpha_i^{E(S)}}\right) = 0$ where $\alpha_i = \left(\frac{1}{(1-P(Q_i))^r} + r(\frac{1}{P(Q_i)} - 1)\right) / (1 - P(Q_i))$. Instead of solving the equation, we deduce the factors to raise $E(S)$. Note that, when $E(S) = 0$, the left side is larger than zero. So the solution of $E(S)$ is bigger when the differential coefficient over $E(S)$ of the left side is smaller. By analyzing the differential coefficient we have the following theorem:

Theorem 2:

$$F'(E(S)) = \sum_{i=1}^{k} \left(P(Q_i) - \frac{1}{\alpha_i^{E(S)}} \ln \alpha_i \right) = \sum_{i=1}^{k} H(P(Q_i)) \tag{4}$$

and H is a convex function on $P(Q_i)$.

Given that $\sum_{j=1}^{k} P(Q_j) = 1$ (see Theorem 1), the more balanced the distribution of $P(Q_i)$ is, the larger the expectation of speedup can be achieved.

4.3 Average Speedup with TA Scheme

Average speedup with TA scheme refers to the number of modules accepting at least one character. The issue here is very similar with that of SA scheme, yet it is much more simple. As mentioned in Sect. 4.1, we assume a distribution over the set of alphabet classes $\{C_1, \ldots, C_t\}$, and denote the probability of C_i with $P(C_i)$. Now given the number of concerned flows n, the theorem below provides the expectation $E(S)$ of speedup here.

Theorem 3:

$$E(S) = t\left(1 - \sum_{i=1}^{t} \frac{P(C_i)^{t+n-2}((P(C_i) - \frac{1}{t})}{\prod_{j=1, i\neq j}^{t} (P(C_i) - P(C_j))}\right) \tag{5}$$

Here, S denotes the speedup value and $E(S)$ is the expectation of the speedup. $P(C_i)$ is the probability of class-set C_i containing the current input character and t is the number of class set.

Proof 1: Let function $sign(x)$ be 1 if $x > 0$ and 0 if $x = 0$, and $E(S) = E(\sum_{i=1}^{t} sign(x_i)) = \sum_{i=1}^{t} E(sign(x_i))$ under the condition $\sum_{i=1}^{t} x_i = n$ where x_i is the number of current input characters in class C_i. Note that $E(sign(x_i)) = 1 - Pr(x_i = 0)$. If $x_i = x$, the probability of a specific solution $(x_1, \ldots, x_t)$ of $\sum_{j=1, j\neq i}^{t} x_j = n - x$ is $\prod_{j=1, j\neq i}^{t} P(C_j)^{x_j}$. Sum them up, we get the probability of $x_i = x$, that is

$$Pr(x_i = x) = P(C_i)^x \sum_{x_1 + \ldots + x_t = n} \prod_{j=1, j\neq i}^{t} P(C_j)^{x_j} \tag{6}$$

which is the coefficient $\alpha_{n-x}^{(i)}$ of u^{n-x} in the polynomial

$\prod_{j=1,j\neq i}^{t} \frac{1}{1-P(C_j)u} = \sum_{i=0}^{\infty} \alpha_i^{(i)} u^i$. So

$$E(S) = \sum_{i=1}^{t} E(sign(x_i)) = \sum_{i=1}^{t}(1 - Pr(x_i = 0)) = t - \sum_{i=1}^{t} \alpha_n^{(i)} = t - \alpha_n \quad (7)$$

where α_n is the coefficient of u^n in polynomial

$$\sum_{i=1}^{t} \prod_{j=1,j\neq i}^{t} \frac{1}{1-P(C_j)u} = \frac{t - \sum_{i=1}^{t} P(C_i)u}{\prod_{i=1}^{t}(1-P(C_i)u)} \quad (8)$$

We give a lemma to compute α_n:

Lemma 4:

$$\prod_{i=1}^{c} \frac{1}{1-a_i x} = \sum_{i=1}^{c} \frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c}(a_i - a_j)} \cdot \frac{1}{1-a_i x} \quad (9)$$

Proof 2: See Appendix.

From the lemma,

$$\begin{aligned}\alpha_n &= t\sum_{i=1}^{t} \frac{P(C_i)^{t-1}}{\prod_{j=1,i\neq j}^{t}(P(C_i)-P(C_j))} P(C_i)^n \\ &\quad - \sum_{i=1}^{t} P(C_i) \sum_{i=1}^{t} \frac{P(C_i)^{t-1}}{\prod_{j=1,i\neq j}^{t}(P(C_i)-P(C_j))} P(C_i)^{n-1} \\ &= \sum_{i=1}^{t} \frac{P(C_i)^{t+n-2}((tP(C_i)-1)}{\prod_{j=1,i\neq j}^{t}(P(C_i)-P(C_j))} \end{aligned} \quad (10)$$

which indicates the theorem.

Theorem 5: For two possibility distributions on t classes: $P(C_i)$ and $P'(C_i)$ for $i = 1, \ldots, t$ only different in two classes, i.e., $P(C_s) < P'(C_s) < P'(C_t) < P(C_t)$ and $P(C_i) = P'(C_i), i \neq s, t$, and let $E(S)$ and $E'(S)$ be the expectation of speedup under two distributions. Then $E(S) < E'(S)$

Proof 3:

$$\begin{aligned}\frac{1}{t}(E' - E) &= \sum_{i=1}^{t} \frac{P(C_i)^{t+n-2}((P(C_i)-\frac{1}{t})}{\prod_{j=1,i\neq j}^{t}(P(C_i)-P(C_j))} \\ &\quad - \sum_{i=1}^{t} \frac{P'(C_i)^{t+n-2}((P'(C_i)-\frac{1}{t})}{\prod_{j=1,i\neq j}^{t}(P'(C_i)-P'(C_j))} \\ &= W_0 + W_x + W_y \end{aligned} \quad (11)$$

where $W_0 = \sum_{i=1,i\neq x,y}^{t} \frac{P(C_i)^{t+n-2}((P(C_i)-\frac{1}{t})}{\prod_{j=1,i\neq j}^{t}(P(C_i)-P(C_j))} - \sum_{i=1,i\neq x,y}^{t} \frac{P'(C_i)^{t+n-2}((P'(C_i)-\frac{1}{t})}{\prod_{j=1,i\neq j}^{t}(P'(C_i)-P'(C_j))}$ *and* $W_i = \frac{P(C_i)^{t+n-2}((P(C_i)-\frac{1}{t})}{\prod_{j=1,j\neq i}^{t}(P(C_i)-P(C_j))} - \frac{P'(C_i)^{t+n-2}((P'(C_i)-\frac{1}{t})}{\prod_{j=1,j\neq i}^{t}(P'(C_i)-P'(C_j))}, i = x, y$

Considering that $P(C_x) - P(C_y) > P'(C_x) - P'(C_y) > 0$, it is trivial to have $W_0 + W_x + W_y > 0$.

4.4 Bounding Speedup in Worst Cases

We bound the speedup theoretically in this subsection. The speedup in our approach as discussed previously, is the number of active modules. There are two cases that would lead to the system containing very few active modules. One is that there are few concurrent flows in the system and the other is that a large number of concurrent states are coexisting in a few modules. Since in most time, the network contains flows in abundance, we only consider the latter. If a module Q_i whose probability of being an active module is $P(Q_i)$, the probability of T current states co-existing in Q_i is $P(Q_i)^T$. In the next section, we will propose a state allocation algorithm, which guarantees that $P(Q_i) < \frac{2}{k} \ll 1$ where k is the number of modules. Given $P(Q_i) \ll 1$, $P(Q_i)^T$ will sharply decrease when T increases. So large amount of states blocking in one module is relatively a small probability event in our approach.

To sum up, we have seen that to reach a higher speedup ratio, it is always beneficial to balance the possibilities of multiple memory modules being the active module as balanced as possible. This requires an elaborate design on how the states and transitions are assigned. In the next section, we would show the detailed SA and TA scheme.

5. Refined SA and TA Scheme

The state assignment and transition assignment are decided during the pre-computation. The limiting distribution $\pi = (P(q_1), \ldots, P(q_N))$ must be computed beforehand. Unfortunately, traditional method (such as Cramer method) to compute π by $\pi = \pi M$ costs a time complexity of $O(N \cdot N!)$, while for SA scheme N is the number of states which is very large and for TA scheme N is the number of character classes. Alternatively, we introduce another approximating method which significantly reduces the time complexity to $O(N^2)$. Initially, $\pi^{(0)} = (1, \ldots, 1)$, and $\pi^{(n+1)} = \pi^{(n)} M$. Exponentially ergodic property of states shows that $|p_i^{(n)} - P(q_i)| \leq (1 - N\delta)^n$ where $\delta = \min(P_{ij} : 1 \leq i, j \leq N)$. We define the "distance" between $\pi^{(n)}$ and π as $d^{(n)} = |\pi^{(n)} - \pi| = \left(\sum_{i=1}^{N}(p_i^{(n)} - P(q_i))^2\right)^{\frac{1}{2}} < N^{\frac{1}{2}}(1 - N\delta)^n$ where $1 - N\delta \ll 1$. So in finite steps, the result is sufficiently approximating to π.

Now, we start to design the SA and TA scheme. According to Theorem 2, our aim is to make $P(Q_i) = \sum_{q\in Q_i} P(q), i = 1, \ldots, k$ as balanced as possible. In more detail, we assume a "deviation" ε and restrict that $|P(Q_i) - \frac{1}{k}| < \varepsilon$. The Algorithm 1 is designed to meet this purpose. The key step is to store state q_i whose $P(q_i) > \varepsilon$ for t times. When storing PDFA in memory modules, if the next state (denoted as q) of certain transition is stored in t memory modules, we randomly choose one of them to store and the probability of each state is thereby $\frac{P(q_i)}{t} < \varepsilon$ (in line 4). To bound the extra memory of state duplication, we denote the number of state q_i with $P(q_i)$ satisfying $j\varepsilon \geq P(q_i) > (j-1)\varepsilon$

Algorithm 1 Allocating N states in k modules so that $|P(Q_i) - \frac{1}{k}| < \varepsilon$

```
1: procedure ALLOCATION(q_1, ..., q_N, k, ε)
2:     N' ← 0
3:     for i = 1, ..., N do
4:         t ← ⌈P(q_i)/ε⌉                ▷ the probability of each state P(q_i)/t < ε
5:         q'_{N'+1} ← q_i, ..., q'_{N'+t} ← q_i
6:         N' ← N' + t
7:     end for
8:     for j = 1, ..., k do
9:         Q_j ← ∅, P(Q_j) ← 0
10:        while P(Q_j) < 1/k, i ≤ N' do
11:            Q_j ← Q_j ⋃ {q_i}
12:            P(Q_j) ← P(Q_j) + P(q_i)
13:            i ← i + 1
14:        end while
15:    end for
16:    return Q_1, ..., Q_k
17: end procedure
```

as a_j, then the total number of states is $N' = \sum_{j=1}^{N} ja_j = N + \sum_{j=1}^{N}(j-1)a_j < N + \frac{1}{\varepsilon}\sum_{i=1}^{N} P(q_i) = N + \frac{1}{\varepsilon}$. Generally, it would suffice to set $\frac{1}{k} = 10\varepsilon$, and then the upper bound of new adding states $N' - N$ is $10k(\sim 10^3)$ compared with $N(\sim 10^6)$. Hence, the extra memory is negligible even in the worst case.

6. Chip Design and Flow Scheduling of PDFA

We present a preliminary chip design of our matching system in Fig. 4. The overall structure follows the formal DFA matching engine manner. The input packets of flows are parsed into header (sent to Session Table) and content (sent to Packet Buffer). Then Session Table maps the packets to flows by parsing their headers and keeps the packets within each flow in their arrival order. Input packets sent to Processing Module are first buffered in a certain Char FIFO in Active Flow Buffer (AFB). Processing Module matches the contents in AFB (depicted in detail later), reports the matching results and feeds back to Session Table when a FIFO in AFB is empty. Then Session Table informs Packet Buffer whether Processing Module can receive data (empty FIFO exists in AFB).

The PDFA is stored in k memory modules which process contents of different packets buffered in r char FIFOs in parallel. We deploy r registers for each memory module. The ith register always receives the (char, state) pair from ith FIFO (buffering the content of the ith flow). In each cycle, every active module chooses one current state using round robin manner from its r registers to process. The memory modules producing no next state send default signs back and all FIFOs without feedback are set "hold". The round robin processing guarantees that all current states would be processed within at most r rounds. The upper bound of time that one flow is delayed in a certain module is the number of current states being just in this module. As analyzed in subsection 4.4, such delay will be very small when $P(Q_i)$ is small for each module Q_i.

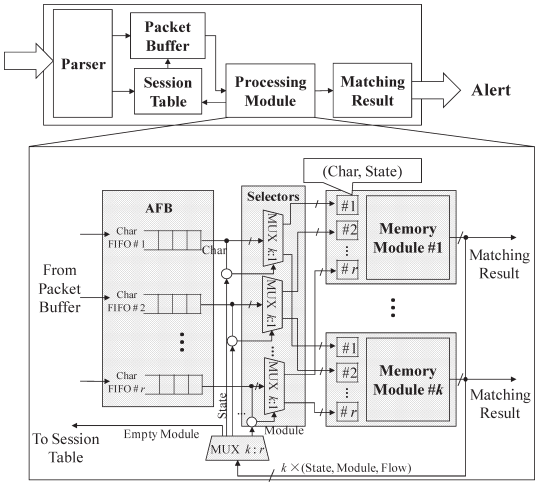


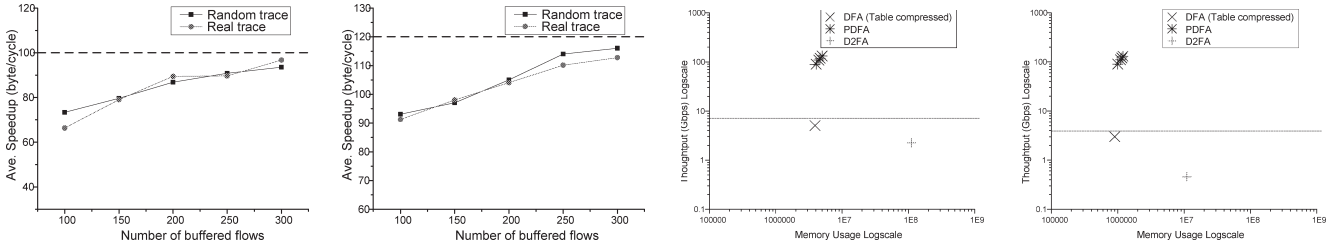
Fig. 4 The on-chip design for fast pattern matching and flow scheduling using PDFA.

7. Evaluation of Performance

We evaluate the performance of our approach on the latest ruleset from Snort [17] and L7-filter [18] where all patterns are written in regular expressions. The patterns are first compiled into standard DFAs using the set splitting techniques proposed by Yu et al. in [10]. It splits the pattern set into multiple subsets so that each subset creates a DFA which can be fit into the memory size. For DFA of each subset, we apply state allocation algorithm to assign states into k memory modules. We plot the result under various number of modules k and $\varepsilon = \frac{1}{10k}$ in Table 1. The table indicates that when k increases from 80 to 140, the total number of states stored of PDFA increases a little and even the theoretical upper bound of increments is negligible ($< 3\%$). When storing the PDFA, we employ one of the most acknowledged DFA compression techniques, D^{2}FA, to compress the memory size of PDFA.

The runtime test is conducted on 10 GB live traffic traces from campus gateway of Tsinghua University and a randomly generated trace. We create the random trace according to the 1-gram characteristic of flows mentioned in Sect. 4.1. The increases of speedup under given number of modules and various flows are plotted in Fig. 5 (a) and Fig. 5 (b). It is evident from the plots that as we increase the number of concurrent flows, the overall speedup scales up. Moreover, when the number of concurrent flows m increases to about $m \approx 3k$, the inter-flow parallelism is almost fully exploited (the speedup almost reaches the upper bound represented by dashed lines).

For the purpose of comparison, we display in Fig. 5 (c) and Fig. 5 (d) the time and space performance of PDFA together with D^{2}FA [7] and table-compressed DFA [9] on Snort ruleset [17] and L7-filter ruleset [18] under the real-world trace. In the two plots, the x-axis (memory usage) and y-axis (throughput) increase on a log scale. The dashed line represents the throughput of the largest instance of orig-



(a) average speedup k = 100 (b) average speedup k = 120 (c) time-space performance under Snort ruleset (k: from 100 to 150) (d) time-space performance under L7-filter ruleset (k: from 100 to 150)

Fig. 5 Performance evaluation: (a), (b) depict the average speedup with given number of modules k under various number of buffered flows; (c), (d) illustrate the time-space trade-offs of PDFA in real traces evaluation compared with D^{2}FA and traditional DFA compression techniques (table compression).

Table 1 The number of states in PDFA using DFA allocation algorithm on rulesets of L7-filter and Snort.

L7-filter (modules)	80	100	120	140
Original DFA	51274	51274	51274	51274
PDFA	51450	51485	51517	51532
Upper bound	51535	51814	52108	52501
Snort (modules)	80	100	120	140
Original DFA	720398	720398	720398	720398
PDFA	720450	720624	720720	721540
Upper bound	720511	720939	721210	721546

inal DFA if it can be fit into memory modules. With single matching instance, a high parallel speedup is obtainable with PDFA while there is little memory duplication (see Table 1) and other compression techniques can be employed. The points representing D^{2}FA and Table-compressed DFA illustrate trade-offs between space benefits and expenses on time. Meanwhile, the points labeling PDFA under various module number k (from 100 to 150) delineate a nearly uncompromising large speedup with little memory cost. It can be indicated that Table-compressed DFA is worse than PDFA in throughput and memory size. The plots also reveal that PDFA achieves high throughput compared with D^{2}FA while their memory sizes are very similar.

8. Conclusion

In this paper, our proposed parallelism architecture of DFA called PDFA takes the advantage of hardware parallelism as well as the large amount of flows in today's network environment to achieve a high parallel speedup. Such speedup is obtained with little extra cost compared with current intra-flow parallel methods. Theoretical analysis and experimental evaluation both show a good performance of PDFA on average situation. Even under worst case traffic, as analyzed in Sect. 4.4 and Sect. 6, the increment of memory size or the reduction of throughput PDFA is negligible.

Acknowledgement

This work is supported by NSFC (60625201, 60873250, 60903182, 61073171), 973 project (2007CB310702), Tsinghua University Initiative Scientific Research Program and open project of State Key Laboratory of Networking and Switching Technology (SKLNST-2008-1-05).

References

[1] N. Hua, H. Song, and T. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," Proc. INFOCOM, 2009.
[2] S. Dharmapurikar and J.W. Lockwood, "Fast and scalable pattern matching for network intrusion detection engines," IEEE J. Sel. Areas Commun., vol.24, no.10, pp.1781–1792, 2006.
[3] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, "A memory-efficient parallel string matching architecture for high-speed intrusion detection," IEEE J. Sel. Areas Commun., vol.24, no.10, pp.1793–1804, 2006.
[4] C. Hu, Y. Tang, X. Chen, and B. Liu, "Per-flow queueing by dynamic queue sharing," Proc. INFOCOM, 2007.
[5] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts, "Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing," Proc. ACM SIGMETRICS, 2005.
[6] B.C. Brodie, D.E. Taylor, and R.K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," Proc. ISCA, 2006.
[7] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," Proc. ACM SIGCOMM, 2007.
[8] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," Proc. ANCS, 2006.
[9] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," Proc. ANCS, 2007.
[10] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, and R.H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," Proc. ANCS, 2006.
[11] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," Proc. CoNEXT, 2007.
[12] R. Smith, C. Estan, and S. Jha, "Xfas: Faster signature matching with extended automata," IEEE Symposium on Security and Privacy (Oakland), 2008.
[13] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," Proc. ACM SIGCOMM, 2008.
[14] "Jflex—The fast scanner generator for java," http://jflex.de/
[15] K. Wang and S.J. Stolfo, "Anomalous payload-based network intrusion detection," Proc. 7th International Symposium on Recent Advances in Intrusion Detection (RAID), 2004.
[16] G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Markovian analysis of large finite state machines," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.15, no.12, pp.1479–1493, 1996.
[17] Snort. http://www.snort.org/

[18] "Application layer packet classifier for linux," http://l7-filter.sourceforge.net/

Appendix

Lemma1:

$$\prod_{i=1}^{c}\frac{1}{1-a_i x}=\sum_{i=1}^{c}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c}(a_i-a_j)}\cdot\frac{1}{1-a_i x} \quad (A\cdot 1)$$

Proof: The lemma would be proved by inducting c. Initially, when $c = 2$, it is trivial.

If the equation holds with certain c, we compute the left sides of (4) with $c + 1$.

$$\begin{aligned}\prod_{i=1}^{c+1}\frac{1}{1-a_i x}&=\left(\sum_{i=1}^{c}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c}(a_i-a_j)}\cdot\frac{1}{1-a_i x}\right)\cdot\frac{1}{1-a_{c+1}x}\\&=\sum_{i=1}^{c}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c}(a_i-a_j)}\cdot\frac{1}{(1-a_i x)(1-a_{c+1}x)}\\&=\sum_{i=1}^{c}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c}(a_i-a_j)}\\&\quad\cdot\left(\frac{a_i}{a_i-a_{c+1}}\cdot\frac{1}{1-a_i x}+\frac{a_{c+1}}{a_{c+1}-a_i}\cdot\frac{1}{1-a_i x}\right)\\&=\sum_{i=1}^{c}\frac{{a_i}^{c}}{\prod_{j=1,i\neq j}^{c+1}(a_i-a_j)}\cdot\frac{1}{1-a_i x}\\&\quad+\sum_{i=1}^{c}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c}(a_i-a_j)}\frac{a_{c+1}}{a_{c+1}-a_i}\cdot\frac{1}{1-a_{c+1}x}\end{aligned} \quad (A\cdot 2)$$

To induct the (5) to (4) under $c + 1$, we must prove that

$$\sum_{i=1}^{c}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c}(a_i-a_j)}\frac{a_{c+1}}{a_i-a_{c+1}}+\frac{{a_{c+1}}^{c}}{\prod_{j=1}^{c}(a_i-a_{c+1})}=0 \quad (A\cdot 3)$$

It is equivalent with

$$\sum_{i=1}^{c+1}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{c+1}(a_i-a_j)}=0 \quad (A\cdot 4)$$

We use induction again to prove (9). Actually, we seek to prove a stronger equation: $\forall t > c$, there is

$$\sum_{i=1}^{t}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{t}(a_i-a_j)}=0 \quad (A\cdot 5)$$

We induct with c. When $c = 1$, it is trivial. Suppose that (9) holds under a certain $c, c-1, \ldots, 1$, we define that

$$g(x_1,\ldots,x_t)\triangleq\sum_{i=1}^{t}\frac{{x_i}^{c-1}}{\prod_{j=1,i\neq j}^{t}(x_i-x_j)}=0, t>c \quad (A\cdot 6)$$

So we have,

$$g(a_1,\ldots,a_t)=0 \quad (A\cdot 7)$$

$$g(a_1,\ldots,a_{t-1},a_{t+1})=0 \quad (A\cdot 8)$$

$$\ldots \quad (A\cdot 9)$$

$$g(a_2,\ldots,a_{t+1})=0 \quad (A\cdot 10)$$

$$(A\cdot 11)$$

Adding them up, we get:

$$\begin{aligned}0&=\sum_{i=1}^{t+1}\frac{{a_i}^{c-1}\sum_{j=1,i\neq j}^{t+1}(a_i-a_j)}{\prod_{j=1,i\neq j}^{t+1}(x_i-x_j)}\\&=\sum_{i=1}^{t+1}\frac{{a_i}^{c-1}\sum_{j=1}^{t+1}(a_i-a_j)}{\prod_{j=1,i\neq j}^{t+1}(x_i-x_j)}\\&=\sum_{i=1}^{t+1}\frac{{a_i}^{c}}{\prod_{j=1,i\neq j}^{t+1}(x_i-x_j)}\\&\quad-\left(\sum_{j=1}^{t+1}a_j\right)\sum_{i=1}^{t+1}\frac{{a_i}^{c-1}}{\prod_{j=1,i\neq j}^{t+1}(x_i-x_j)}\\&=\sum_{i=1}^{t+1}\frac{{a_i}^{c}}{\prod_{j=1,i\neq j}^{t+1}(x_i-x_j)}\end{aligned} \quad (A\cdot 12)$$

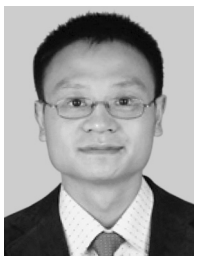
Yi Tang was born in 1983. He received his B.S. degree from Department of Computer Science, Northwestern Polytechnical University, Xi'an, China, in 2005; He is a Ph.D. Candidate at Department of Computer Science and Technology, Tsinghua University, China. His research area is in Deep Packet Classification, Web Security, Network Intrusion Detection.

Junchen Jiang was born in 1988. He is a undergraduate student at Department of Computer Science and Technology, Tsinghua University. His research area is in Deep Packet Classification, Traffic Management.

Xiaofei Wang was born in 1982. He is a Ph.D. Candidate at School of Electronic Engineering of Dublin City University, Ireland. His research area is in Deep Packet Classification, P2P detection, Traffic Management.

Chengchen Hu was born in 1981. He received his Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2008. He is currently an assistant research professor in the department of computer science and technology of Tsinghua University. His research interests include high performance routers, traffic management and network measurement.

Bin Liu was born in 1964. He received his Ph.D. degree in the Department of Computer Science and Engineering from Northwestern Polytechnical University, Xiän, China, in 1993. Currently, He is a Full Professor in the Department of Computer Science and Technology, Tsinghua University. His current research areas include network processors, traffic measurement and management, high performance switches/routers, and high speed network security. Bin Liu has received numerous awards from China including the Distinguished Young Scholar of China.



Zhijia Chen was born in 1982. He is a Ph.D. student at Computer Science department of Tsinghua University, China. He is a visiting student at school of engineering of Stanford University in Spring 2007 and visiting scholar in CS Department of Hong Kong UST in 2008. His research area is in content distribution networks and P2P networks.