A Regular Expression Matching Engine with Hybrid Memories

Shuhui Chen[†] and Rongxing Lu[‡]
[†]College of Computer Science, National University of Defense Technology, Changsha, China
[‡]Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada
Email: shchen@nudt.edu.cn; rxlu@bbcr.uwaterloo.ca

***Abstract*—A key technique of network security inspection is by using the regular expression matching to locate the specific fingerprints of networking applications or attacks in the packet flows, and accordingly identify the underlying applications or attacks. However, due to the surge of various networking applications and attacks in the past years, much more fingerprints needs to be investigated in this process, which demands a large memory space for regular expression matching. In addition, with the network links nowadays frequently upgraded, the network flow rate increases dramatically. This demands the fast operation of regular expression matching accordingly with the enhanced throughput for network inspection. The requirements on fast operations and large memory space are, however, conflicting due to the limited space of the fast memory. On addressing this issue, in this paper, we propose to use hybrid memory. In specific, by investigating on the transition table state access probability through the Markov theory, it can be observed that there exist a number of states which is much more frequently accessed than others. Therefore, we devise a matching engine which is suitable for FPGA implementation with two-level memories, where the first-level memory uses the on-chip memory of FPGA to cache the frequently accessed state transitions, and the second-level memory, composed of slow and cheap DRAM, stores the whole state transitions. Lastly, the L7-filter's regular expression patterns have been applied to obtain the state access probability, and different quantities of memory assignment approaches have been investigated to evaluate the throughput.**

***Index Terms*—Deep Packet Inspection; Regular Expression; Matching Engine; Markov Chain**

I. INTRODUCTION

Deep Packet Inspection (DPI) is a key technique for Network Intrusion Detection System (NIDS) and Network Forensic System (NFS). In these network security devices, the payload of packets is matched against certain pre-defined patterns, namely fingerprints, to identify specific classes of applications, viruses, attacks and criminal evidences, etc.

One approach in DPI is using the string matching. However, it is not so flexible and powerful to describe complicated fingerprints. As a remedy to that, regular expressions become a main approach rapidly instead of explicit string patterns as the pattern matching language in packet scanning applications. Its widespread use is due to its expressive power and flexibility for describing useful patterns. For example, in the Linux Application Protocol Classifier (L7-filter) [1], all protocol identifiers are expressed as regular expressions. Similarly, the Snort [2] intrusion detection system has evolved from no regular expressions in its rule set in April 2003 to several thousand rules using regular expressions now. Another intrusion detection system, Bro [3], also uses regular expressions as its pattern language.

To conduct the match, regular expressions are compiled into Finite State Machine (FSM), and then the packet payload is scanned with the FSM. In the case of a light-weight network, with low flow rate and a small number of patterns, pattern separated approach (in which every regular expression is compiled into several FSMs, and the packet payload is matched one FSM by another FSM) can satisfy the throughput requirement. However, with the dramatic increase of bandwidth, multi-Gbps links are nowadays widely applied in campus networks, and the scale of the patterns of the typical DPI system is over one hundred, the traditional pattern separated approach no longer meets the critical performance requirement. If a pattern integrated FSM is utilized, the inflation of the states makes it impossible to be filled into high speed memory like Static Random Access Memory (SRAM) or on-chip memory of Field Programmable Gate Array (FPGA).

In regular expression matching, the storage capacity and the memory speed become an irreconcilable conflict. In order to decrease the storage requirement, the previous researches (a detailed survey of the previous literatures will be provided later) exert to reduce the transition table memory. In this paper, we do not consider how to deduce the memory of the state transition table, but take hold of how to conjointly use high speed memory with small space and slow speed memory with large space, to make matching engine at about the throughput of the high speed memory.

Every byte of packet payload needs at least one memory access in DPI systems; as a result, the throughput constraint of the network security devices is memory access time. Generally, the immanent fluidness of network packets results in a very low hit ratio of the cache. Since the number of the memory access is pre-defined and cannot be changed subsequently, the potential to enhance system performance is to improve the efficiency of each memory access.

Traditional network security devices like NIDS and NFS adopt high performance CPU platforms such as x86, MIPS, to improve the system performance. These typical CPUs are developed for the improvements on the calculation performance, so they are made as perfect or effective as possible on cache coherence, branch prediction, out-of-order execution, multi-core parallelism, etc. The improvements on memory access concentrate on how to increase the hit rate of the cache, but the fluidness of network packets cannot exploit these advantages; therefore, customizing hardware based approach such as FPGA is introduced as its better pipelining and parallelism. In addition, FPGA has specially designed on-chip memory for high bandwidth linkage with logic units, which could accomplish high speed communication between matching engine and memory. Nevertheless, the space requirement of the state transition table compiled from real-world patterns is far more than the capacity of the FPGA's on-chip memory.

To address the contradiction between the performance of the matching engine and the capacity of the memory, in this paper, we present a new hybrid memory matching engine, which enables the high throughput and a large scale of state transition table storage by utilizing two levels of memories to accommodate the state transition table. Specifically, the contributions of this paper are three-fold.

- First, based on the transition table, a state transition possibility matrix has been constructed, and the state accessing probability using Markov chain with stable state vector is investigated. To the best of our knowledge, it is the first work researching on formed states of regular expressions.
- Second, a two-level-memory hierarchy storage mechanism has been introduced, in which the Markov theory is used to obtain the frequently accessed states and their transition table entries are stored in the first-level memory while the second-level memory is used to hold the whole transition table entries.
- Finally, real-world regular expression patterns are used to produce the state transition table and state probability table, and a simulation analysis is employed to show that our hybrid-memory architecture can obtain the throughput almost as the first-level memory while the memory cost is nearly the same as the second memory.

The remainder of this paper is organized as follows. The related work is provided in section II, and the motivations on introduction of Hybrid Regular Matching Engine are presented in section III. Then, section IV depicts the system architecture and framework. The process of obtaining the stable vector is introduced in section V. The experiments and result analysis are in section VI. Finally, we conclude our work in section VII.

II. Background and Related Work

Finite State Automata (FSM) is a natural formalism for regular expressions. Although Deterministic Finite Automaton (DFA) and Nondeterministic Finite Automaton (NFA) are two kinds of FSMs that can be used to conduct DPI, the more preferred approach is DFA as it supports non-backtracking search. In DFA-based systems, a number of researches based on DFA systems compile m regular expressions into a composite DFA, which provide guaranteed performance benefit over running m individual DFAs. Specifically, a composite DFA reduces processing cost from O(m) (O(1) for each automaton) to O(1), i.e., a single lookup obtains the next state for any given character. However, the number of states in the composite automaton grows to $O(\Sigma^{nm})$ in the theoretical worst case.

TABLE I
Worst case comparisions of Different FSM.

	One regular expression of length n		m regular expressions compiled together	
	Processing complexity	Storage cost	Processing complexity	Storage cost
NFA	$O(n^2)$	$O(n)$	$O(n^2m)$	$O(nm)$
DFA	$O(1)$	$O(\Sigma^n)$	$O(1)$	$O(\Sigma^{nm})$

Even through the approach compiling m regular expressions into a single FSM is relatively fast, its $O(1)$ scanning complexity still cannot meet the current links' bandwidth requirement, as the aggregated Internet traffic has been experiencing an annual bandwidth growth of 40%-50% in recently years [4]. To break the performance bottleneck of regular expression matching engine, a number of researches have been studied to improve the overall throughput by achieving efficient content-matching. Previous reported researches mainly focus on improving the throughput of the rule matching algorithms, and/or employing FPGA [5]–[7], GPU [8]–[11] or TCAM [12] for efficient content-matching.

Another kind of researches exert to reduce the memory requirements, which cluster into two classes:

(1) The first FPGA-based solutions implement NFA [13]–[15]. Although NFAs are always smaller than DFAs, they need more memory bandwidth as an NFA may be in several states simultaneously whereas a DFA is always only in one state. Thus each byte that is processed might need to access the transition table for $|Q|$ times. Prior works have looked for different ways to find good NFA representations of transition table that limits the number of states need to be processed simultaneously. These approaches combine the matching engine with the state transition table, which may lead to inconvenient pattern update.

(2) The second class of the researches that exert to reduce the memory is based on DFA. Although the approach with m regular expressions compiled together into a DFA can gain O(1) computing complexity, its storage cost is extremely higher than that of the separated DFA. The number of states in a DFA scales poorly with the size and number of wildcards in the regular expressions. For a naive DFA with k states, its memory requirement is $|\Sigma|\times k\times\lceil\ln k\rceil$, which is unsustainable as the k increases so fast with the increasing of the regular expression number m and n (average length of the expressions). In 2007, Becchi, etc [16] proposed a redundance eliminating method. Its basic idea is merging "nonequivalent" states in a DFA by introducing labels on their input and output transitions. Its evaluation shows that it can drastically reduce the DFA memory requirement, but its performance is influenced as it needs to access several sub tables.

In [17], the authors analyzed the size of DFAs for typical payload scanning patterns, and developed a grouping scheme which can strategically compile a set of regular expressions into several engines, resulting in a remarkable improvement on regular expression matching speed without much increase in memory usage.

Default transition and D^2FA (Delayed Input DFA) were introduced and employed in [18], [19]. D^2FA is a special FSM based on DFA but with default transitions where each state can have at most one default transition to one other state. The directed graph named as deferment forest consisting of only default transitions in a D^2FA must be acyclic. D^2FA representation reduces transitions by more than 95% but with a speed penalty for long default transition paths in the D^2FA matching.

In [12], the authors introduced a hardware-based RE matching approach that uses Ternary Content Addressable Memory (TCAM) and its associated Static Random Addressable Memory (SRAM) to hold state transition table. They proposed three novel techniques: transition sharing, table consolidation, and variable striding, to reduce TCAM space and improve regular expression matching speed. The experiments based on 8 real-world regular expression sets show that small TCAMs can be used to store large DFAs and achieve potentially high regular expression matching throughput.

We find the above-mentioned DFA based space compressing methods except [12] sacrifice performance to some extent to occupy less memory.

Our research is orthogonal to these researches as the second-level memory is not so sensitive to the access performance, which signifies that using these research achievements can reduce the second-level memory space in our proposed approach.

III. Process in Regular Expression matching

The proposed matching architecture is based on the Deterministic Finite Automata (DFA), which is due to DFA is faster than NFA (Nondeterministic Finite Automata), as depicted in Table I. A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, A)$, where Q is a set of states, Σ is an alphabet (containing the 2^8 symbols from the extended ASCII code), $\delta : \Sigma \times Q \rightarrow Q$ is the transition function, q_0 is the initial state, and $A \subseteq Q$ is a set of accepting states.

Any set of regular expressions can be converted into an equivalent DFA with the minimum number of states by transforming the regular expressions into NFA, converting NFA to DFA using subset construction algorithm, and state minimization algorithm [20]. The fundamental issue in regular expression matching is the throughput and large amount of memory required to store transition table δ, as we have to store $\delta(q, a) = p$ for each a of every q.

To make the matching progress more clear, a simple example is the regular expression "ab.c". The DFA after compilation is depicted in Fig. 1. If a string (packet body) "ababac" needs to be scanned, matching engine begins from initial state 0; byte 'a' makes it go to state 1; byte 'b' makes it go to state 2. Afterwards, matching engine goes to state 4, state 2, state 4, and state 5 finally. As state 5 is an accepting state, the input is accepted.

From the aforementioned example, it is observed that every byte of packet payload needs one memory access, and every two successive memory accesses are interrelated. Pipeline

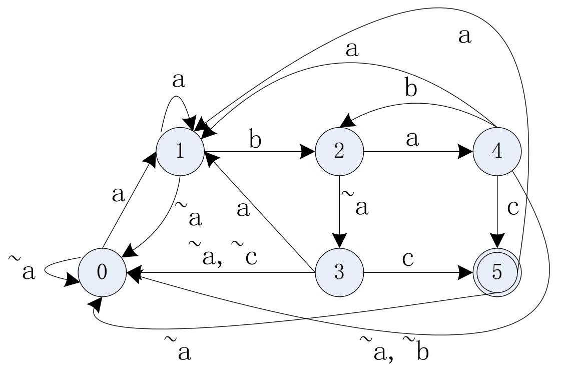

Fig. 1. An example of FSM.

for memory access cannot be achieved due to this type of interrelation. Therefore, the matching throughput performance is almost totally dependent on the state number that needs to be accessed and the state access delay. As the state number is not easy to be decreased because it is the same as the length for every packet payload, we consider how to reduce the state access delay. It seems the best way is using faster memory to reduce the delay.

Different memory has different properties. For example, SRAM has a relatively high access speed (166MHz when this paper is being written), but its price is high and its size could not be very large (about several Mega bytes according to the current technology); while the Dynamic Random Addressable Memory (DRAM) has a relatively large size and high throughput for its burst, but one memory access may take a number of cycles to obtain a result as DRAM has a long startup time. In the regular expression matching, the successive accesses are associative as the following access is dependent upon the result of the previous access, so pipeline cannot take effect and the high throughput of the DRAM cannot be fully utilized. FPGA always has its own embedded (on-chip) memory with high throughput and fast response time (about 1 to 2 ns), thus it is very suitable to this continuing dependant access, but its size is often below one mega bytes, which is too small to accommodate the whole transition table in practical security systems.

Although embedded memory in FPGA is small, its maximum frequency can reach more than 500 Mega Hz. Even through the frequency will be reduced to about 400Mhz (depending on the complexity of logic and the model of the FPGA) after synthesized, it is still much faster than DRAM. As an example, if Altera Stratix II EP2S180 (with 9 M-RAM blocks and all the blocks can be configured as real dual-port mode) is utilized and the time frequency after synthesized is 400 Mhz, matching engine with single scanning module can reach a throughput of 400Mhz*2*8=6.4 Gbps. In fact, it is very easy to parallelize the memory block access in one matching engines to reach more than 20Gbps throughput. The biggest challenge is to solve the memory space problem of the transition table.

In brief, there is not a single memory can satisfy both the size and the throughput requirements. As a popular technology in computer architecture, Cache is used by the CPU of a computer to reduce the average time to access the main memory. A Cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached, the average latency of memory accesses will be closer to the Cache latency than to the latency of main memory.

The challenges of most multi-memory computer system are that:

(1) How to make the first-level memory fast enough.

(2) How to make the last-level memory large enough to accommodate the frequently accessed program instructions or data.

(3) How to make the data being accessed with higher hit ratio at the first-level memory.

We borrow the idea from computer architecture. In our design, we find the fastest memory in optional is on-chip memory, which has a feature that almost every mainstream FPGA has several banks of on-chip memory that can be concurrently accessed. If the same entries are stored in different banks of memory, access performance can be increased several-fold for the access operations can be distributed to different bank, but the constraint is that its size gives an upper bound of the DFA states stored on-chip.

The whole state number of a security system is often bigger than a hundred thousand, which makes the whole memory requirement more than a couple of Giga Bytes. Although there are some kinds of compression methods can decrease the storage space, a conservative estimate is that more than one dozen Mega Byte space is still required to implement a practical security detection system [16]–[19]. DRAM is chosen to be our second-level memory as its capacity could be very large.

Now, the arisen problem is that how to select the most frequently accessed states with the limited space of the on-chip memory in FPGA. From Fig. 1, the intuitive feeling is that state 0 is the most frequently accessed state, as there are 1528 transitions go to this state. It is observed that state 3 has 254 incoming transitions while state 1 has only 5 incoming transitions. Does it mean that state 3 is more frequently accessed than state 1? In fact, it is not as state 3 can only be transferred from state 2 with only two outgoing transitions, accessing frequency of state 3 may be less than that of state 2. Now that state 2 has only two incoming transitions, so the accessing frequency of state 3 may be less than that of state 1. It is a probability and statistics issue to determine which states are more frequently accessed than the others. The second issue is that the memory delay needs to be 8bits/10Gbps=0.8 ns in the mainstream 10G POS or Ethernet links, on-chip memory cannot meet the performance requirement, so concurrently accessing multi-bank may be a choice.

IV. System Architecture

Fig. 2 shows the proposed system architecture. The design mainly contains a Regex Compiler, a Storage Assigner, a Matching Engine and two levels of memories.

As Regex Compiler and Storage Assigner are components with very complex computation processes, they are implemented by software in embedded CPU connected to FPGA.

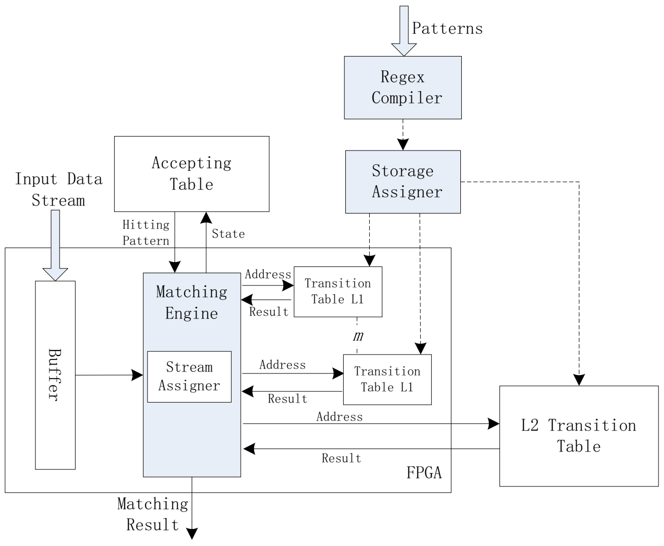

Fig. 2. Hybrid matching engine.

Regex Compiler's input is multiple regular expression patterns while its output is the state transition table. Storage Assigner obtains the state transition table and determines how to assign the entries to the two different memories. In the proposed architecture, Storage Assigner is very significant as the perfect state assignment can lead to the best performance/cost ratio, and the detailed analysis of it is in section V .

The whole state transition table is divided into two sub-tables: the first-level sub-table is a duplicated of some entries stored in different on-chip memory banks while the second-level sub-table is stored in the off-chip memory. The matching engine consists of several scanning modules each of which processes particular streams (Algorithm 1). As the packets from the same stream (the packets defined by a five tuple consisting source IP address, source port number, destination IP address, destination port number and protocol number, which presents the bidirectional communication comes from the same peer) must be scanned by the same modules, a Stream Assigner is employed to make sure the fairness of the load of every scanning module. There are a number of researches [21], [22] focus on solve the balance of the stream and packet, which are beyond the range of this paper. In the performance analysis of the section VI, it is assumed that all the scanning modules are fully utilized.

During the matching process, every scanning module utilizes the current state identification left shifting 8 bits with the addition of the current byte to obtain the entry that consists of the next state and the hit tag.

Every state has 256 entries each of witch consists of a special hit bit and the next state indexed by the corresponding character (as depicted in Fig. 3). The hit bit (tag) is introduced to avoid one more memory access, although most states do not hit any rule, one memory access is still necessary to check whether it is an accepting state. The introduced hit bit leads to one memory access gaining not only the next state but also whether the next state is hit, while the trade off is a little more memory. The hit bit is 1 if the next state is hit, else it is 0. Accepting table is a map of state identification and its correspondent hitting rule identification. As this table is not

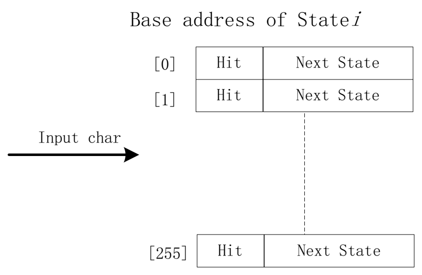

Fig. 3. Entries of every state.

so large and is only accessed when hit happens, SRAM or DRAM can be used according to the remaining resource of the chosen FPGA model as SRAM is easier to control than DRAM while DRAM is cheaper than SRAM.

The scan algorithm of the matching engine is depicted in Algorithm 1. From the scan algorithm, we find the matching engine is still very simple compared with the naive matching engine. The only difference is that our matching engine needs one operation to decide which memory to access.

Algorithm 1. Scan(pkt).

```
Input: $pkt$ : new arrived packet.
$FirstTab$ : State Transition table of the first level.
$SecondTab$ : State Transition table of the second level.
$AcceptingTab$ : Accepting table.
Output: $return$ : Hitting result.
1  $CurrentState \leftarrow 0$;
2  for ($i$=0;$i < pkt\_len$;$i$++) do
3      $ch \leftarrow pkt(i)$;
4      $Address \leftarrow CurrentState \ll 8 + ch$;
5      if ($CurrentState < FirstStateNum$) then
6          $Item \leftarrow FirstTab[Address]$;
7      end
8      else
9          $Item \leftarrow SecondTab[Address]$;
10     end
11     $CurrentState \leftarrow Item.NextState$;
12     if ($Item.Hit$) then
13         return $AcceptingTab[Item.NextState]$;
14     end
15 end
```

The above design is suitable for implementing high-speed regular expression matching engine, as almost every FPGA has several bands of on-chip memories that are connected with logic units closely, which makes the matching throughput higher than the usage of any kind of off-chip memory. The only requirement to achieve cost-effective matching of such a system is to ensure that the scan process accesses the on-chip memory in an overwhelming major occasions. The coming section will discuss state probability problem and the key issues of how to ensure such a locality.

V. State Probability

The storage arrangement of the state transition table consists of (1) obtaining the state transition probability matrix, (2) gaining the steady probability vector and (3) state renumbering. We depict them in the following sub-sections.

A. State transition probability matrix

Assume the matching engine is being in a state q_i, what about the probability that the next state will be q_j while $0 \leq j < |Q|$? It is obvious that if there is no a meets $\delta(q_i, a) = q_j$, then the probability from q_i to q_j named as $p_{i,j}$ is zero.

In addition, how about we have some a meet $\delta(q_i, a) = q_j$? Here, as DFA instead of NFA is used, so for an arbitrary input a, $\delta(q_i, a)$ is not a set consisting of several states but only one state, and it is postulated that the packet content is stochastic, thus the probability is proportional to the number of a that meets $\delta(q_i, a) = q_j$. As en example, if the next states of the one step reachable transitions from q_i are only q_j and q_k, and the set of inputs from q_i to q_j is A and the set of input from q_i to q_k is B, then the probability from q_i to q_j is $|A|/|A+B|$, while the probability from q_i to q_k is $|B|/|A+B|$.

The state transition table is a $|Q| \times |\Sigma|$ matrix with the values in the matrix are the next state numbers or zeros if the next states are unreachable directly. From the state transition table, the transition probability $|Q| \times |Q|$ Matrix P is constructed as follows:

$$p_{i,j} = |\{a|\delta(i,a) = j, a \in \Sigma, i, j \in Q\}|/|\Sigma| \tag{1}$$

Since we postulate the packet content is stochastic, P is a matrix whose entries are all between 0 and 1 and whose row-sums are all 1. P is a Markov chain, as if the chain is currently in state q_i, then it moves to state q_j at the next step with a probability denoted by $p_{i,j}$ and this probability does not depend upon which states the chain was in before the current state.

The probability $p_{i,j}$ is called transition probability from state i to state j. During the scanning process, the next state can be the same as the current state, and this occurs with the probability $p_{i,i}$. As the example in Fig. 1, when current state is 1, and the input character is ‘a’, the next state is still 1, which means $p_{1,1} = 1/256$ as there is only one character makes the transition from state 1 to itself.

To make the transition probability issue more clear, Fig. 1 is taken as an example again to construct the transition probability matrix as follows:

$$P = \begin{bmatrix} \frac{255}{256} & \frac{1}{256} & 0 & 0 & 0 & 0 \\ \frac{254}{256} & \frac{1}{256} & \frac{1}{256} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{255}{256} & \frac{1}{256} & 0 \\ \frac{254}{256} & \frac{1}{256} & 0 & 0 & 0 & \frac{1}{256} \\ \frac{253}{256} & \frac{1}{256} & \frac{1}{256} & 0 & 0 & \frac{1}{256} \\ \frac{255}{256} & \frac{1}{256} & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

As it can be seen from the probability matrix, state 0 has a probability of 255/256 to move to itself and a probability of 1/256 to move to state 1, which is due to that only one character ‘a’ leads the state 0 to transfer to state 1 while the other 255 characters lead the state 0 to transfer to itself.

B. Steady probability vector

As patterns in DPI systems are always very complicated, some of them are with starting anchor (^), the constructed DFA transition matrix is very huge and complex, and some states induced from the pattern with starting anchor are caused to be transient states. Take patterns "ab" and "^a.*c" as an example, the DFA diagram is depicted in Fig. 4.

Consider the transition diagram in Fig. 1. Starting at any state, we are able to reach any other state in the diagram directly in one step or indirectly through one or more intermediate states. Such a Markov chain is termed as irreducible Markov chain. On the other hand, starting from any state, we might not be able to reach other states in Fig. 4, directly or indirectly. For example, if we begin at state 2, we cannot reach state 0, 1, 3 and 8. Such a Markov chain is named as reducible Markov chain.

An irreducible Markov chain is defined as one in which each state is reachable from every other state either directly or indirectly; whereas a reducible Markov chain is one in which some states cannot reach other states. Furthermore, the states of a reducible Markov chain can be divided into two sets: closed states (C) and transient states (T). In Fig. 4, states 0 and 2 are transient states, whereas the two groups $\{1,3,8\}$ and $\{4,5,6,7\}$ are both closed states.

We study the steady-state behavior of reducible composite Markov chains in this sub-section. In our reducible Markov chain, it is composed of some sets of closed states. At the begin of the scan, it is always in initial state 0, which is a transient state (as in Fig. 4); therefore, it can move to either state of closed sets. In the view of the scanning, hitting states of unanchored patterns are always in closed sets, while those of anchored patterns may be in closed or transient sets. From the initial state, the matching engine can reach every closed sets to match different patterns.

The transition matrix is given by the canonic form:

$$P = \begin{bmatrix} T & A_1 & A_2 & ... & A_l \\ 0 & C_1 & 0 & ... & 0 \\ 0 & 0 & C_2 & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & C_l \end{bmatrix} \tag{3}$$

where
$C_i = square\ stochastic\ matrices. \quad (i = 1, ...l)$
$A_i = rectangular\ nonnegative\ matrices. \quad (i = 1, ...l)$
$T = square\ substochastic\ matrix.$

Therefore, the states of the original Markov chain are now separated into $l+1$ mutually exclusive subsets, where states in $C_i (i = 1, 2, ..., l)$ are closed states that states in every of them can communicate with each other within the same subset and they have no edge outgoing to other state out of the set, and states in T are transient. The probability transition probability matrix after n multiplications [23] is

$$P^{(n)} = \begin{bmatrix} T^{(n)} & Y_1^{(n)} & Y_2^{(n)} & ... & Y_l^{(n)} \\ 0 & C_1^{(n)} & 0 & ... & 0 \\ 0 & 0 & C_2^{(n)} & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & C_l^{(n)} \end{bmatrix} \tag{4}$$

where $T^{(n)} \to 0$ as $n \to \infty$ and

$$Y_i^{(n)} = \sum_{k=0}^{n-1} C_i^{n-k-1} \cdot A_i \cdot T^i = C_i^n \cdot A_i \cdot (I - T)^{-1} \tag{5}$$

.

As scan begins at initial state 0, we only need to gain the first row vector of $Y_k^{(n)}$ to obtain the distribution probabilities after n bytes (steps). We observe that each row of matrix $Y_i^{(n)}$ is a scaled copy of the rows of $\lim_{n\to\infty} C_i^{(n)}$ from equation (5). Also, the sum of each row of $Y_i^{(n)}$ is lesser than one, and the total sum of the row from every $Y_i^{(n)}$ is exactly one. From the equation (5), we know that the convergence speed of $Y_i^{(n)}$ is strictly dependent upon the convergence speed of the $C_i^{(n)}$.

For a reducible Markov chain, during the state transition process, there is a stationary distributions of $W = \lim_{n\to\infty} P^{(n)}$, but unlike irreducible Markov chain, rows in W are not the same for a reducible Markov chain. The steady probabilities of a reducible Markov chain are dependent upon the initial state.

There are two kinds of approaches to obtain the stationary distributions of a Markov chain, the first is iteration (likes Jacobi, GaussCSeidel and Successive Over-relaxation), and the second is direct computation. Traditional iteration approaches are not the preferred methods as we don't know how to define the threshold; in other words, we don't know when is the right time to stop the iterations. Thus, the direct computation may be our selection. Even though the example in Fig.1 has only 6 states, the states of a FSM consisting of more patterns may inflate to several millions. The computation complexity of gaining the stationary distributions must be acceptable as the update of the pattern occurs sometimes although not very frequently.

To gain the steady matrix W from $WP = W$ is likely to solve the m-linear equations. The first row of W is the steady vector beginning at initial state 0. For this reason, the issue is converted to obtaining w_1, w_2,... and w_m from

$$[w_1 w_2 ... w_m](P - E) = 0$$

and

$$w_1 + w_2 + ... + w_m = 1$$

As P is a large-scale square matrix, it is computation-intensive to use general methods of linear algebra on such an issue, because most of the algorithms need O(m^3) arithmetic operations devoted to solving the set of equations. Even through we can obtain a set of solutions of w from the equations, we still need some computation to obtain that which w is located at the first row of W. Special sparse matrixes, for example

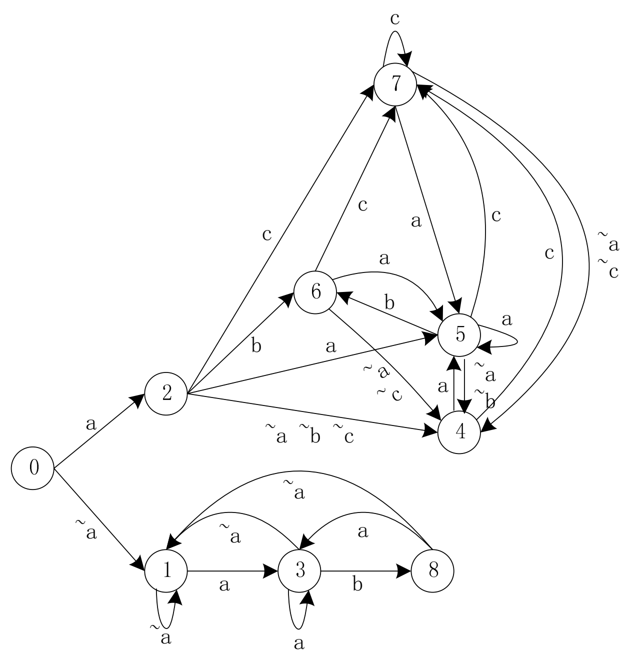

Fig. 4. Markov chain deduced from transition table.

trigiagonal, band diagonal with bandwidth M bolck diagonal, and etc. are all have some special approches to solve. However, our probability transition matrix does not have such features that can be used to accelerate the solution.

We turn back to the method of the iteration and find the convergence speed is seriously dependent upon the scale of the matrix. From equation (5), if the state number of every C_i is not so big, multiplication of C_i will converge rapidly, so every $Y_i^{(n)}$ will converge rapidly; as a result, the first row of the iterative multiplication of P will converge rapidly. Fortunately, our Markov chain has such a feature, even through the whole chain is in large scale, which will be discussed in Section VI. The repeated multiplication technique is prone to numerical roundoff errors and one has to use repeated trials until the first row of the stop changing for increasing values of n, which is not a problem as our approach does not need to obtain the exact result of the probability distribution.

C. State renumbering

After the order of the first row vector in $P^{(n)}$ is already steady, the state transition entries with the highest probabilities in the first vector of the matrix are selected to be stored in the first-level memory. In Algorithm 1, in order to accelerate the memory access speed, only the state identification number and the input character are utilized to obtain the access address of the next state; as a result, these states stored in the first-level memory (M_1) must have relatively lower identification number to avoid more complicated addressing mechanism.

Rearranging the state identification numbers is not a hard work as we only need the most frequently accessed state transitions to be stored in M_1, instead of sorting the states according to their probabilities. If the M_1 is arranged to hold m_{L1} states (every state has 256 entries), the original preceding m_{L1} states can be copied to another buffer, and then scan the whole transition table to change the state identification number and conduct movement, whose computation complexity is only $O(m \cdot m_{L1})$.

The second-level memory (M_2) holds the whole transition entries instead of the remaining ones. The first reason is that we do not prefer to use a complex addressing mechanism as if we store the whole entries in the second-level memory, simple state shifting with the addition of the current byte can gain the access address. The second reason is that the entries stored in the first-level memory is very small relative to the whole memory requirements, so even repeated storage is a cost that can be sustained relative to the performance it brings to.

The selection of how many states are stored is dependent on what throughput is required and the FPGA types, which will be discussed in the next section.

VI. Evaluation

Experiments are employed to evaluate our proposed two-level-memory approach. Real-world regular expression patterns of L7-Filter are utilized to generate the state transition table, and the probability transition matrix is obtained to evaluate the throughput performance.

It is assumed there are k banks of memories and all the k scanning modules are fully utilized, the average time from the launched reading operation by the matching engine to the return of the result from the first-level memory is t_1, and the average time of accessing one entry in the second-level memory is t_2. As one memory access can only process one byte of the packet payload, the throughput performance is deduced as follows:

$$T = \frac{8}{\sum\limits_{State_i \in M_1} w_i \cdot \dfrac{t_1}{k} + \sum\limits_{State_i \in M_2} w_i \cdot t_2} \quad (bps) \qquad (6)$$

.

DRAM is chosen as the second-level memory for its large storage capacity. In the current technical conditions, the clock frequency of the typical DRAM is 800Mhz, but it needs about 100 cycles to gain one entry. Therefore, the average access time of one entry is about $(1000 \times 100)/800$ =125 ns. As a reference to evaluate the whole throughput, Altera EP2S180 is assumed as the FPGA that will be used to hold the first-level state transition table. EP2S180 has 9 double-port M-RAM Blocks, each of which has a frequency of 420 MHz and a capacity of 512K, so the average access time of one entry is about 1.19 ns.

The following sub-sections firstly discuss the statistics result of the state probability transition table. Afterwards, based on the probability statistics and attributes of the chosen FPGA, throughput using a single bank and multi-bank memory is respectively discussed.

A. Basic statistics of state transition table

L7-Filter's 114 patterns are used as our source of the Regular Expressions. A modified Thompson algorithm [24] is employed to compile multiple Regular Expressions into the same NFA; and then the minimum subset construction algorithm [20] is used to transfer NFA into DFA; finally, minimizing the DFA is conducted. As a result, 407,281 states are obtained through the above-mentioned steps.

As every state needs 256 entries to rapidly obtain the next state, the state transition table entry space for every state needs $(\lceil \ln(407281) \rceil + 1) * 256 \approx 5Kbits$ (the one additional bit is used as the hit tag). Therefore, the whole storage space for the state transition table is about 254 Megabytes. As a common sense, SRAM cannot meet the space requirement, which is why we select DRAM as the second-level memory. On the other side, if the Altera EP2S180 is chosen in our FPGA implementation, every M-RAM block (its size is 512K) can accommodate 102 states.

The state transition probability matrix is obtained from the DFA based on the methods depicted in section III. We utilize $P^{2^n} = P^n * P^n$ and find after 8 times of matrix multiplications, the order of the probabilities in the first row becomes steady, and we gain 3044 states with nonzero access probability, and the probabilities of the other 404,237 (the ratio to the whole state number is 99.25%) states are zero, which means a majority of the states are transient. Therefore, after enough times of scan, the massive states can not be accessed if the a hit has not happen and the scan is still continuing (many DPI systems stop the scan of the corresponding stream after finding a match in a packet).

We sort all the states based on their probabilities after the order is steady, and the result shows that very little states occupy huge access probability, the first state has a probability of 94.578935% especially. Our statistics shows that about 0.0004% of the whole states(that is 18 states) occupy about 99.902342% of the access probability, which is depicted in Fig. 5. In this figure, only the probability distribution function of the first 100 states is shown as the probability sum of the remaining states only occupy about 0.000527% of the state accesses.

In summary, the memory access of state transient table indeed presents a perfect convergence feature. Thus, if the transition entries with larger probability are stored in the first-level memory and the ones with smaller probability or zero probability are stored, the approach utilizing two level of memories may perfectly exploit the performance of the first-level memory, which will be analyzed in the following two sub-sections.

B. Throughput on one-bank memory

In this sub-section, we don't consider how many parallel memory banks does a FPGA have. Since even for a FPGA has several memory banks, we can assemble them as a uniform storage space.

The throughput performance changing with the increase of the first memory space is evaluated in this sub-section. We know that more state transition entries are stored in the first-level memory, the higher the throughput will be. However, as the limit of the technique and the cost, we cannot make the on-chip memory too large.

In this experiment, the entries of the state transition table stored in M_1 are added state by state, so the increment of memory space at every step is 5K bits. If all the transition entries can be stored in the on-chip memory, the ultimate throughput is 8 bits/1.19 ns = 6772.7 Mbps. We found that if only the entries of one state are stored in on-chip memory according to equation (6), the throughput is 1012.4 Mbps. Moreover, when the entries of 62 states, which are only 0.0117% of the whole storage quantum, are stored in M_1, the throughput will be 6705 Mbps which is about 99% of the ultimate throughput. In other words, as the on-chip memory is very precious and cannot be expanded easily, it is not appropriate to store more entries than it is required, for example, it is not cost-efficient to store entries more than 62 states.

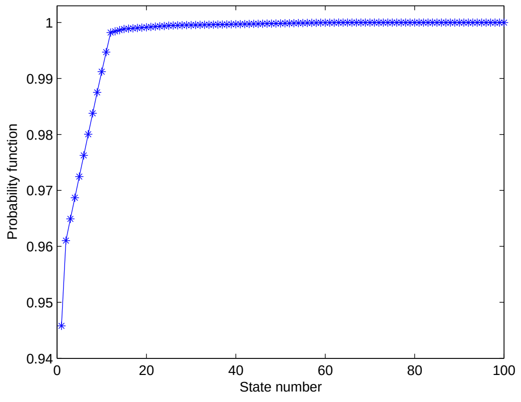

Fig. 5. Probability distribution function for the foremost 50 states.

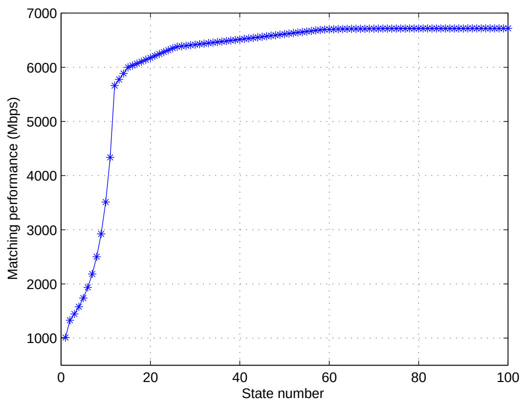

Fig. 6. Throughput with only one bank used in FPGA.

C. Throughput of multi-bank memory

Different FPGA has different configuration. For example, the number of logic units, pin number, I/O features and memory features are different for different model. For a system designer, the chosen chips with optimal performance/cost ratio are ofen expected. Although the number of M-RAM blocks within a FPGA doesn't absolutely determine the price, it is closely tied up with it. To gain a directive idea about the selection of a FPGA, we hereby examine the throughput with the fixed size of the first-level memory. For the sake of simplicity, it is supposed that the fixed sized memory can be separated as several parallel banks as we like.

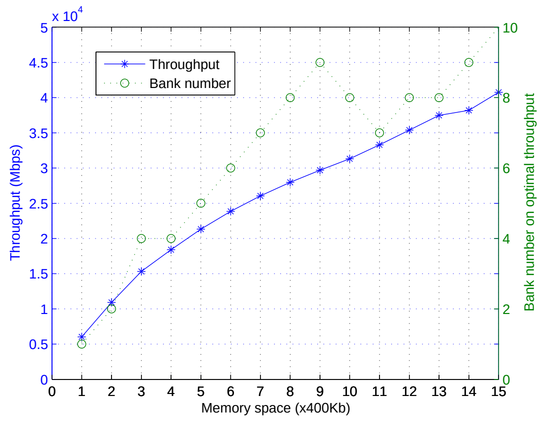

Fig. 7. Throughput with multibanks of memory in FPGA.

It is easy to understand that more banks make better performance, as different bank can be accessed concurrently. On the other hand, the more state transition entries can be stored in the first-level memory, the better performance is obtained as more transitions can be obtained in higher speed memory. Therefore, it must be a compromise between the bank number and the state number. If the whole memory that used to store state transition entries is fixed, it is expected to know how many banks are the best choice and what is the optimal throughput for a certain size of memory.

Fig. 7 shows the effect of memory size on optimal throughput performance, and the corresponding bank number in the case of the optimal throughput. Generally, when the whole memory space is relatively less, higher throughput is made by storing more entries; therefore, fewer banks can make the highest throughput. As the memory space is increasing, more banks may decrease the average access time of the first-level memory. If the whole memory is set to be 4.0 M bits, the optimal throughput is 27.9 Gbps when 8 banks are used. The result in Fig. 7 can guide the selection of the FPGA model. For example, if Altera Stratix II EP2S180 is select, we know that we can obtain the peak performance of 29.6 Gbps with 9 banks and total 4.5 M bits memory are fully used. This throughput is relatively high as the method utilizing expensive TCAM [12] can only gain the 19 Gbps level of performance, while our proposed approach doesn't need any complicated coding mechanism and expensive electronic component.

VII. Conclusion

Regular Expression Matching is a fundament of NIDS and NFS. However, it faces the challenges of the throughput performance and the memory inflation. Many researches exert to tackle one of these two issues to some extent. We present in this paper a DFA-based regular expression matching engine with all the patterns compiled into one FSM and some of transition entries chosen to be stored in on-chip memory of the FPGA.

First, through the analysis of the state probability distribution of a real-world regular expression patterns, we found different state has different access ratio, and then the concept of cache was borrowed from the computer architecture to accelerate the the access speed of the state transitions. Based on the aforementioned discovery, a hybrid matching engineer based on two-level memories has then been introduced: the first-level memory is utilized to hold the state transition entries of the most frequently accessed states and the second-level memory contain the whole state transition entries. Finally, L7-Filter is employed as the real-word patterns to identify that by employing two levels of memories, we can obtain the scanning performance of 29.6 Gbps, which is higher than the most advanced method, without any expensive electronic components and complicated architecture.

VIII. Acknowledgment

This work has been supported by the National High-Tech Research and Development Plan of China under Grant No.2011AA01A103.

References

[1] "Application layer packet classifier for linux," http://l7-filter.sourceforge.net/.
[2] "Snort," http://http://www.snort.org/.
[3] "The bro network security monitor," http://http://bro-ids.org/.
[4] "Minnesota internet traffic studies (mints)," http://www.dtc.umn.edu/mints/2002-2009/analysis-2002-2009.html/.
[5] N. Weaver, V. Paxson, and J. M. Gonzalez, "The shunt: an fpga-based accelerator for network intrusion prevention," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, New York, USA, 2007, pp. 199–206.
[6] M. Labrecque and J. G. Steffan, "The case for hardware transactional memory in software packet processing," in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, New York, USA, 2010, pp. 37:1–37:11.
[7] H. Wang, S. Pu, G. Knezek, and J. Liu, "Min-max: A counter-based algorithm for regular expression matching," 2012.
[8] V. Giorgos, A. Spiros, P. Michalis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 116–134.
[9] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 265–283.
[10] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Midea: a multi-parallel intrusion detection architecture," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 297–308.
[11] C. H. Lin, C. H. Liu, and S. C. Chang, "Accelerating regular expression matching using hierarchical parallel machines on gpu," in *IEEE Globecom 2011*, Houston, Texas, USA, December 2011, pp. 1–5.
[12] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *Proceedings of the 19th USENIX conference on Security*. USENIX Association, 2010, pp. 8–8.
[13] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the 2007 ACM CoNEXT conference*. ACM, 2007, p. 1.
[14] M. Becchi1 and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008, pp. 50–59.
[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*. ACM, 2007, pp. 127–136.
[16] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE, 2007, pp. 1064–1072.

[17] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz, “Fast and memory-efficient regular expression matching for deep packet inspection,” in *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*. IEEE, 2006, pp. 93–102.

[18] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 339–350, 2006.

[19] 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*. ACM, 2006, pp. 81–92.

[20] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to automata theory, languages, and computation*. Addison-wesley, 2007.

[21] T. Nelms and M. Ahamad, “Packet scheduling for deep packet inspection on multi-core architectures,” in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2010, p. 21.

[22] J. Kuang, L. Bhuyan, H. Xie, and D. Guo, “E-ahrw: An energy-efficient adaptive hash scheduler for stream processing on multi-core servers,” in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*. IEEE, 2011, pp. 45–56.

[23] F. Gebali, *Analysis of computer and communication networks*. Springer Verlag, 2008.

[24] K. Thompson, “Programming techniques: Regular expression search algorithm,” *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.