Computer Communications 70 (2015) 86–94

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

A compression approach to reducing power consumption of TCAMs in regular expression matching

Jinmin Yang[a,*], Jie Yang[a], Kun Huang[b], Huigui Rong[a], Kin Fun Li[c]

[a] College of Computer Science and Electronic Engineering, Hunan University, China
[b] Institute of Computing Technology Chinese Academy of Sciences, China
[c] Department of Electrical and Computer Engineering, University of Victoria, Canada

ARTICLE INFO

Article history:
Received 13 January 2015
Revised 11 June 2015
Accepted 2 August 2015
Available online 8 August 2015

Keywords:
Regular expression matching
Ternary content addressable memory
Power consumption
Compressibility identification
Source state differentiation

ABSTRACT

Ternary content addressable memory (TCAM) is a popular device for fast regular expression (Regex) matching in networking and security applications. The rapid growth of Regexes necessitates large TCAM memory consumption, which in turn impacts power consumption. Compressing the transition table can cut down TCAM memory consumption, thereby reducing its power consumption. This work identifies the compressibility of transition entries and then proposes a compression scheme. In our scheme, the compression ratio is improved by skillfully assigning identifiers to states in a deterministic finite automaton (DFA). Furthermore, our scheme utilizes the wildcard function and the priority matching mechanism provided in TCAM to exploit the minimum differentiation among a set of source states. A complete implementation of the identifier assignment and transition table compression is presented. Experimental results on real-world Regex sets show that our scheme is significantly more effective, reducing power consumption by 87.4% and memory space by 93.2%, and improving throughput up to 114.7% on average compared to prior work.

1. Introduction

Deep packet inspection (DPI) [1, 2] is a core component of networking and security devices such as network intrusion detection/prevention systems (NIDS/NIPS) [3,4], firewalls, and routers, etc. Its function is to detect malicious intrusions and identify specific applications. This is achieved by examining both the header and the payload of network data packets against a set of predefined string signatures. This process is called 'string matching'. However, it is difficult to use such a string matching approach to represent and match complex string signatures. Therefore, regular expression (Regex) matching [5–6] is adopted to displace string matching in many DPI systems such as bro [7], snort [8], and Cisco [9], as Regex is more flexible and expressive in describing complex string signatures. In practice, a given Regex can be precisely realized by equivalent deterministic finite automaton (DFA). DFA is a finite state machine in which a source state accepts an input character and then migrates to a unique destination state. In DPI, DFA built from Regex is typically implemented in hardware such as ternary content addressable memory (TCAM) [10].

TCAM is a type of memory that parallel search can be performed at a high speed. A TCAM consists of a set of memory entries. Each entry is a vector of cells, where every cell can store one bit. Therefore, a TCAM memory entry can be used to store a string. TCAM is expensive and its content space is limited, so TCAM is typically implemented with a low-cost memory device, like static random access memory (SRAM) [11]. As to a search, TCAM works as follows: given an input string, it compares this string against all entries in its memory in parallel, and returns the matching data in SRAM. When TCAM is used to perform DFA, every state transition in a DFA is encoded as one TCAM memory entry. Every entry consists of two parts: the TCAM part storing one source state of a DFA and one input character, and the associative SRAM storing the corresponding destination state. All TCAM memory entries built from transitions in a DFA constitute a global transition table.

TCAMs are off-the-shelf chips capable fast parallel lookups, and have been widely deployed in modern network devices. Though, the high power consumption of TCAM has become a critical issue with the proliferation of data centers which have thousands of TCAM-based network devices. Nowadays, more and more companies are concerned with power consumption in their devices. For example, a Cisco system provides an additional 600 Watt redundant power module to support its four 150 Watt external network devices [12]. In these devices, a typical 18 Mb TCAM, such as Cypress's NSE10K and Netlogic's NL3280, can consume up to 15 Watts of power when all TCAM entries are activated for a search [13]. The 15 Watts of each TCAM is significant power consumption in data centers with

* Corresponding author. Tel.: +86 13975896967.
E-mail address: rj_jmyang@hnu.edu.cn (J. Yang).

http://dx.doi.org/10.1016/j.comcom.2015.08.003
0140-3664/

thousands of TCAM-based network devices. This is especially critical in power-constrained situations where most high performance devices, such as routers and switches, can easily consume tens and hundreds of Watts if these searches are not optimized.

The power consumption of TCAM is directly proportional to the number of its memory entries. A huge DFA with a large number of Regex signatures means the need of many memory entries in the TCAM. Consequently, the high power consumption of TCAM occurs in lockstep with the rapid growth of the size of Regex signatures. Furthermore, DFA suffers from the state explosion problem, where the number of DFA states grows exponentially with the size of Regex signatures. With the growing size of Regex signatures, how to reduce TCAM power consumption has become a hot spot in large data centers with thousands of TCAM-based network devices.

When the memory entries of TCAM are chunked into a set of fixed-size blocks, it is feasible to reduce the power consumption of TCAM. The feasibility comes from the heuristics that a search is involved in only a small part of memory entries of the TCAM. When a search is executed, only the involved blocks need to be activated, while non-involved blocks can stay in the dormant state. Thus, the TCAM power consumption is decreased. The key issue of the strategy is how to identify in advance which blocks are involved in a search. The character-indexed scheme [14] was proposed to address this issue by separating input characters from the global transition table in a TCAM. This scheme constructs an extra front-end character index table for the blocks involved in a specific search. Nevertheless, such a scheme still leads to high power consumption, because every transition table has many redundant transitions. If every transition table could be compressed, the TCAM power consumption can be further reduced.

This work aims to reduce TCAM power consumption by transition table compression. We call our approach power-efficient DFA (PEDFA). The main idea is to compress all transition tables so as to minimize TCAM space usage, thereby achieving lower TCAM power consumption. PEDFA exploits two features of TCAM. First, TCAM has a ternary state in each cell. Each cell has three states: 0, 1 and * (wildcard). A "*" state matches both "0" and "1", hinting the possibility of memory compression. Second, TCAM returns the first matching entry when two or more entries match the input key, which is called TCAM priority matching. This mechanism can be used to further reduce the number of TCAM entries. It implies that those TCAM memory entries with the same prefix codes and destination state can be displaced by a single TCAM memory entry consisting of the prefix codes plus wildcard state "*". Suppose that there are two TCAM memory entries "000 → 010" and "001 → 010". They have the same prefix code "00" and destination state "010". Consequently, they can be compressed into one TCAM memory entry "00* → 010".

The proposed PEDFA is evaluated using the real Regex sets *snort*, *bro*, and *cisco*. A series of experiments are conducted to compare PEDFA to state-of-the-art character-indexed (CIDFA). The preliminary results demonstrate that PEDFA reduces 93.2% memory space as well as 87.4% power consumption, and improves throughput up to 114.7%, on average against CIDFA.

The structure of this paper is as follows. In Section 2, we discuss related work. Section 3 gives a detailed description of the proposed PEDFA. Section 4 presents our evaluation methodology and the experimental results on real Regex sets. Finally, conclusion is presented in Section 5.

2. Related work

For TCAM-based implementations of Regex matching, memory efficiency is a key issue. It has a direct influence on matching speed and throughput, and especially on power consumption. Many approaches have been proposed to improve memory efficiency.

Sharing common data [15] is an effective approach to memory efficiency. For transition entries stored in TCAM, there exists a lot of redundancy in terms of input character, source state, and destination state. The redundancy can be reduced or eliminated by extracting common parts and then sharing them. One way is to transform the storage structure from a table into a tree [15]. In this approach, many schemes were proposed, such as transition sharing, default transition, and failure transition in [15], and character indexing in [14]. Transition sharing merges multiple transitions of DFA into one TCAM entry by exploiting both character redundancy and state redundancy, while a table consolidation scheme [15] extracts the common part of multiple transition tables to form a shared table.

Memory efficiency can be also achieved by transformation [16–20]. For example, delay input DFA (D^2FA) [16] constructs a tree structure for every DFA according to its inherent logic relationship, respectively, and then analyzes local similarity in a tree or among trees, similar to default transition path [17] and variable striding [16]. After that, every similar part is extracted and transformed into a simple transition with a bit of additional information and/or treatment. For example, CD^2FA [18] uses a prefix and a hash function to determine the specific outlet of a default path in matching. The DFA deflation scheme [19] implements one simple TCAM lookup to match an input character by exploiting the structural connection between DFA and non-deterministic finite automaton (NFA). NFA is space-efficient, as the memory needed by NFA grows linearly with the size of Regex signatures. However, NFA is time-inefficient, as it maintains possibly multiple active states at the same time. CFA [20] focuses on the transformation of the complex terms of Regex patterns like the wildcard closure ".*" and the associated character "{}". Those complex terms have an important influence on DFA state size, so CFA can effectively reduce TCAM space consumption. However, CFA requires devices to have certain logic handling capability.

Compression [21,22] is another approach to memory efficiency. Although multiple transition entries have different values in one or more attributes, they can be compressed into one entry by exploiting TCAM characteristics. The wildcard provided by TCAM contributes to this target. In TCAM, a value with one or more wildcards can cover and express multiple values. In this compression approach, higher compression ratio is usually the primary research effort. The global scheme in [21] directly compresses the rows of the transition table, while the local scheme in [22] conducts compression after transforming the two-dimensional table into a hierarchical tree. Compact DFA [22] achieves a higher compression ratio by adding a bit of information about state properties in the state code.

Although memory efficiency contributes to lower power consumption of TCAM, special emphasis should be put on power efficiency. This is critical for large-scale deployment of devices in network environment such as data center networks. String matching probability was exploited to reduce power consumption [22]. Based on the characteristic that the mismatch has a very high probability, prefixes are extracted from strings to form two-level matching in separate modules. Thus the scale of parallel matching becomes smaller, leading to lower power consumption. Another low-power TCAM solution was called character-indexed DFA (CIDFA) [14]. CIDFA reduces the power consumption by indexing characters and structuring TCAM units. In CIDFA, TCAM memory consists of two kinds of blocks: character-index blocks and transition blocks. In the character-index blocks, all input characters are associated with the corresponding transition block IDs. In transition blocks, all transitions are clustered based on the input characters. In a Regex matching, the first thing is to scan character-index blocks to determine which transition blocks need to be activated.

CIDFA can avoid activating irrelevant transition blocks, resulting in lower power consumption. However, its memory consumption is still high because it does not compress its TCAM space. High memory consumption implies high power consumption in TCAMs.

Therefore, this paper proposes TCAM space compression over CIDFA to further reduce TCAM power consumption. In many existing compression schemes [15–22], the typical strategy is to extract common units and then share them by structure adjustment, or to conduct an equivalent transformation to logic units to reduce redundancy. In contrast, our scheme exploits the minimum differentiation among a set of source states, and then utilizes the wild card function and the priority matching mechanism provided in TCAM to implement compression. Moreover, our scheme improves the compression ratio by adequately assigning identifiers to the states of a DFA.

3. The scheme for power-efficient DFA (PEDFA) compression in TCAM

Section 3.1 addresses basic TCAM architecture for Regex matching and shows how to implement a DFA constructed from Regexes into a TCAM. The implementation consists of two steps: constructing an equivalent DFA from the given Regexes, and then encoding DFA into a TCAM. After that, we examine the encoded DFA in TCAM for some compressibility characteristics. In Section 3.2, we use two representative Regexes "{.*a.*bc}" and "{.*c.*de}" to illustrate the basic idea of our PEDFA approach and its implementation details. Section 3.3 gives the proof of the validity of PEDFA.

3.1. DFA-based TCAM architecture

A Regex matching can be represented by a DFA that can be used to validate an input string is a part to be retrieved. The automaton of DFA reads an input string provided in advance from left to right, one character at a time. The automaton begins at a start state and then proceeds by accepting the first character and following the state transition corresponding to the character. The computation continues to read the next characters and follows the state transitions until there are no more inputs. If the automaton reaches some accepting state, the matching succeeds.

Given a Regex set, an equivalent DFA needs to be constructed first, and then it is stored into a TCAM device. We take the Regex {.*a.* bc} as an example to demonstrate the construction details. Fig. 1(a) illustrates its corresponding DFA. In Fig. 1(a), a single circle represents an ordinary state, and the number inside is its identifier. A solid arrow represents a transition. For example, state 0 accepts input character 'a' and then moves to state 1. Theoretically, every state can accept any character and then migrate to a new state. The symbol "[^a]" of state 0 represents input characters other than 'a'. In this example, state 0 is the start state. A double circle represents an accepting state that indicates a successful Regex matching, such as state 3. Fig. 1(b) shows the corresponding TCAM. Every transition is stored into one TCAM memory entry. To concisely illustrate its structure, some transition entries are omitted in Fig. 1(b). Every entry consists of two parts: source state code and input character. For example, for the first entry in Fig. 1(b), its source state is "00", its input character is "01100001", and its destination state is stored into the associated SRAM entry. In a search, suppose the current state is "10" and the current input character is "01100011". The TCAM entry "10–01100011" matches the search and the destination state "11" is returned. Then the state "11" becomes the current state and it continues to read the next input character.

In CIDFA, the transition entries of every specific input character are organized together and stored into TCAM blocks as adjacent as possible. Thus CIDFA knows in advance which TCAM blocks need to be activated when it reads in a specific input character. That is, every possible input character has its own transition table in which transitions are stored into TCAM blocks according to the order of their source state identifiers. Additionally, there is a character-index table in which every row records a possible input character and its corresponding segment of TCAM blocks. The architecture of CIDFA is shown in Fig. 1(c). In a search, CIDFA first queries the character-index table to get those TCAM block IDs corresponding to the current input character. It then activates those TCAM blocks and searches the stored entries for the destination state corresponding to the current state. After that, the current state is shifted to the hit destination state and CIDFA continues to process the next input character.

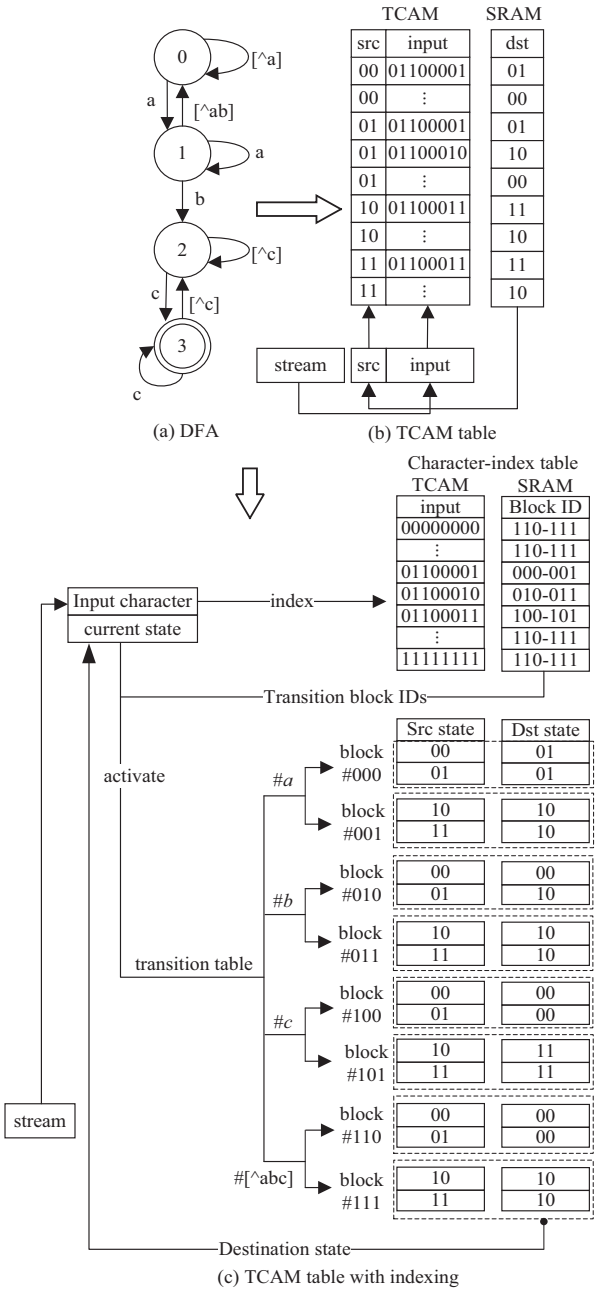

Fig. 1. DFA and TCAM table of {.*a.* bc}.

CIDFA achieves power consumption efficiency by identifying those TCAM blocks that need to be activated. Nevertheless, as the transition table of every character contains a lot of entries, the number of activated TCAM blocks is still large. If those entries can be compressed effectively, the number of activated TCAM blocks will be reduced. In the next subsection, we present such a compression scheme.

3.2. Transition table compression

Transition table is the main component for high power and memory consumption, especially when the DFA size is very large.

Therefore, the more a transition table is compressed, the more the power and memory consumption is reduced.

Our scheme conducts two levels of compression. The first level is a kind of recoverable compression, and the second level is a sort of feasible compression by exploiting TCAM priority matching. The basic idea of recoverable compression comes from the following heuristics. In a transition table, if two rows have the same destination state value, and their source state codes are identical except in only some bit positions, they can be compressed into one row. For example, the two rows, "000 → 000" and "001 → 000", can be compressed into one row, "00* → 000". In order to ensure the validity of this compression scheme, we present a compressible condition in Section 3.2.2. According to the compressible condition, reasonable assignment of identifiers to states in a DFA is crucial for a desirable compression result. Therefore, our transition table compression consists of three steps: pre-processing, recoverable compressing, and feasible compressing. Pre-processing tries to make as many rows meeting the compressible condition as possible, by reasonably assigning the identifiers to states in a DFA. Algorithm 1 shows the pseudo code of the transition table compression.

3.2.1. Pre-processing

The motivation of pre-processing comes from the heuristics that the assignment of identifiers to states in a DFA affects the compression ratio of recoverable compression. For example, suppose that there are five transitions in a transition table: "000 → 000", "001 → 000", "010 → 001", "011 → 000", and "100 → 000". According to the compressible condition, only the first two rows can be compressed into one row, reducing the transition table to four rows. If we reassign identifiers of states, we can obtain better compression ratio. Specifically, let the state "010" swaps its identifier with the state "100", and other states remain the same. So, the new transition table becomes "000 → 000", "001 → 000", "010 → 000", "011 → 000", and "100 → 001". Consequently, the first four rows can now be compressed into one row, resulting in a better compression ratio. This situation widely exists in all transition tables.

The idea of pre-processing is to grant the priority of the identifier assignment to those states that can bring a maximum compression ratio. Let's treat a DFA as a set of directed graphs. If some vertex has a maximum in-degree, then the assigned identifiers of those vertexes with an edge pointing to this vertex should meet the compressible condition. With this identifier assignment strategy, our preprocessing scheme consists of four steps: counting, reordering, identifier reassigning, and state encoding (see Algorithm 1, from 1st line to 14th line). We take two Regexes "{.*a.*bc}" and "{.*c.*de}" as an example to illustrate the details of pre-processing. We firstly assign a sequential number to every state in DFA in turn. Fig. 2(a) is the original global transition table built from DFA. Its horizontal axis indicates the input characters and its ordinate axis indicates the source states. For example, state "0" migrates to state "1" when it reads in an input character 'a'. The label '^' indicates input characters other than the ones listed in Fig 2(a).

The counting step collects statistics to destination states in the global transition table in the horizontal direction. Then the counts are sorted in descending order. The ordered results are shown in Fig. 2(a), where the number to the left of the brackets is referred to some destination state, while the number in brackets is its statistical count.

The reordering step moves the positions of rows according to the counting results in the above step. First all rows are ordered in descending pattern by the first column as in Fig. 2(a). After that, the rear rows are moved recursively forward to the position after the row with the same destination state. This processing transforms the table in Fig. 2(a) to the one shown in Fig. 2(b).

After reordering, the adjacency relationship of states becomes clear. We assign sequential identifiers to those states close to one another according to the compressible condition. One of the reassignment results is shown in Fig. 2(c). The first column of Fig. 2(c) shows the change of state identifiers. For instance, the element "3 → 2" in the third row indicates the state with three as its original identifier has two as its new identifier. After every state is reassigned with a new identifier, the global transition table is updated based on the new identifiers. Note that the identifier reassignment step simply changes the identifiers of DFA states, without destroying the original DFA logic.

The state encoding step encodes DFA states with a binary code. First, we calculate the width of the binary code, *bit_width*, according to the maximal value of state identifier. In this example, *bit_width* is equal to $\lceil \log_2 11 \rceil = 4$. The encoding is quite simple: the identifier of every state is converted into the corresponding binary code. For

Algorithm 1
Transition table compression.

```
Pre-processing
1:  for each row in transition table do
2:    statisticsTable.push_back(the number of states in rows);
3:  end for
4:  sort(each entry in descending order of statisticsTable);
5:  for each entry in transition table do
6:    for each item in entry do
7:      updating the state_ID of item according statisticsTable;
8:    end for
9:    updating the position of entry according statisticsTable;
10: end for
11: bit_width = log2(DFA size);
12: for each state in DFA do
13:   state.code = conver_binary_code(state_ID.bit_width);
14: end for
Recoverable compression
15: function recoverable_compression(column, index, end)
16: if(destination states in column from index to end are same) do
17:   m = ⌈log2(end-index)⌉;
18:   src_code = prefix_code+ m times '*';
19:   result.push_back(entry<src_code, dst_code>);
20: else
21:   new_end = (index+end)/2;
22:   recoverable_compression(column, index, new_end);
23:   recoverable_compression(column, new_end, end);
24: end if
25: end function
Feasible compression
26: cur_iterator = end_iterator = result.begin();
27: prefix_code = end_iterator->src_code.prefix_code;
28: while(end_iterator != result.end()) do
29:   dst_state = end_iterator->dst_state;
30:   while(dst_state == end_iterator->dst_state) do
31:     updating prefix_code and end_iterator;
32:   end while
33:   while(end_iterator->src_code.prefix_code == prefix_ code) do
34:     ++end_iterator;
35:   end while
36:   for(each iterator between cur_iterator and end_iterator) do
37:     map_iterator = count_map.find(iterator->dst_state);
38:     if(map_iterator != count_map.end()) do
39:       ++map_iterator->second;
40:     else
41:       count_map.insert(make_pair(iterator->state,1));
42:     end if
43:   end for
44:   remove_state = count_map.find(max_count)->first;
45:   for(each iterator between cur_iterator and end_iterator) do
46:     if(iterator->dst_state == remove_state) do
47:       result.remove(iterator);
48:     end if
49:   end for
50:   initialize every bit of com_transition with '*';
51:   strncpy(com_transition, prefix_code, strlen(prefix_ code));
52:   entry = make_pair(com_transition, remove_state);
53:   insert entry into result after those uncompressed transitions;
54:   updating cur_iterator and end_iterator;
55: end while
```

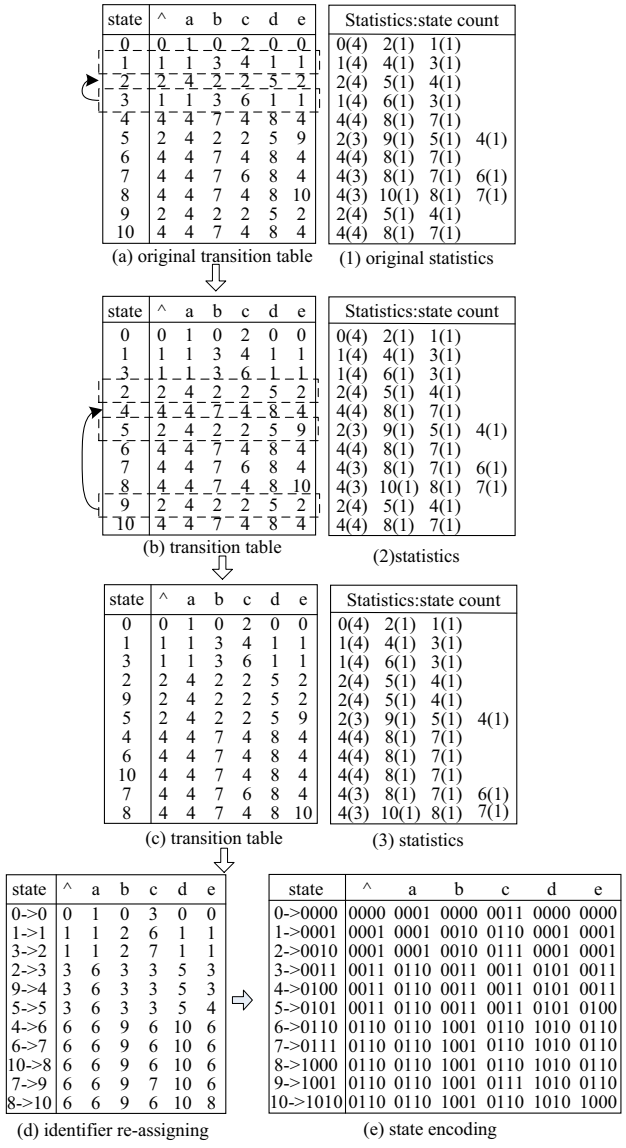

Fig. 2. Pre-processing of {.*a.*bc} and {.*c.*de}.

example, the binary codes of state 0 and 1 are "0000" and "0001", respectively. Fig. 2(e) shows the encoding results.

3.2.2. Recoverable compression

The motivation of recoverable compression is to exploit the wildcard "*" matching state of TCAM to recursively merge two rows into one in the transition table of an input character, thus reducing the number of rows. At the same time, the original semantics is preserved completely. That is to say, after being compressed, the original rows can be recovered completely, without any excess or loss. Furthermore, the correctness of matching is not impacted.

In our scheme, for any two rows in the transition table of an input character, the compressible condition is defined as follows:

(a) They have the same destination state.
(b) Their source state codes are the same except only one bit in some position, the unlike bit is '1' in a row, and '0' in the other.
(c) The unlike bit is the last bit or all bits to its right are '*'.

For any two rows meeting the compressible condition, they can be compressed into a row with the same destination state, and the unlike bit in the source state code is set to '*'.

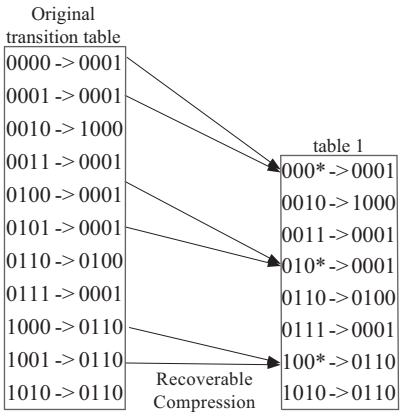

Fig. 3. Recoverable compression.

^		a		b	
src	dst	src	dst	src	dst
0000	0000	000*	0001	0000	0000
0001	0001	0010	0001	0001	0010
0010	0001	0011	0110	0010	0010
0011	0011	01**	0110	0011	0011
010*	0011	100*	0110	010*	0011
011*	0110	1010	0110	011*	1001
100*	0110			100*	1001
1010	0110			1010	1001
c		d		e	
src	dst	src	dst	src	dst
0000	0011	0000	0000	0000	0000
0001	0110	0001	0001	0001	0001
0010	0111	0010	0001	0010	0001
0011	0011	0011	0101	0011	0011
010*	0011	010*	0101	0100	0011
011*	0110	011*	1010	0101	0100
1000	0110	100*	1010	011*	0110
1001	0111	1010	1010	100*	0110
1010	0110			1010	1000

Fig. 4. Recoverable compression of {.*a.*bc} and {.*c.*de}.

From this definition of compressible condition, we can infer that, as to the source state of a row in the compressed result, if the number of '*' it contains is m, the row comes from the compression of 2^m rows. Furthermore, since the identifiers of the 2^m rows are sequential, we call this adjacent transition compression. Additionally, we should note that the third constraint is necessary for correct matching. Without this constraint, correct matching could be impacted. For example, assuming there are two rows "0*10 → 000" and "011* → 001" in a compressed transition table. When the current state is "0110", both rows match to it. Therefore, the correct matching is impaired. In fact, the two rows are interleaving. We will discuss this in detail in the next subsection.

The implementation of the recoverable compression is shown in lines 15 to 26 of Algorithm 1. Fig. 3 illustrates the course of recoverable compression on a transition table, and Fig. 4 shows the compressing results from Fig. 2(e). 66 rows are compressed into 48 rows. So the compression ratio of the recoverable compression is (66 – 48)/66 = 27.3% in this case.

From the definition of the compressible condition, it is clear that this kind of compression is directly recoverable.

3.2.3. Feasible compression

The above compressible condition is very strict. Although it can achieve recoverable compression and correct matching, it has a low compression ratio. In this subsection, we investigate a feasible compression by relaxing the compressible condition.

The relaxed compressible condition is defined as follows: for two rows in the transition table of an input character, if

(a) They have the same destination state.
(b) The bit values of their source state codes are unlike in at least one bit position.

The two rows can be compressed into one row. In the resulting row, unlike bits in the source state code are set to '*'. For example, two rows "0110* → 00001" and "00101 → 00001" can be compressed into the row "0*10* → 00001". This kind of relaxed compression inevitably leads to higher compression ratio, but with three problems, i.e., the special case problem, the interleaving problem and the identity problem. For instance, in the above compression example, the result row covers four source state instances. However, it comes from the compression of three rows only, since "0110* → 00001" implies two rows. If there is a row "00100 → 00011" in the transition table, its source state "00100" is a special case of "0*10*". When the current state is "01000", there exist two rows matching it. Consequently, correct matching is impaired.

We use the following example to demonstrate the interleaving problem. Suppose that there are still two rows "10100 → 00011" and "00100 → 00011" in the transition table. They can be compressed into one row "*0010* → 00011". When the current state is "00101", both "*010* → 00011" and "0*10* → 00001" match it. As a result, an incorrect matching could occur. The reason is that "*010* and "0*10*" are interleaved, as the '*' is in the first bit position in "*010*", while it appears in the second bit position in "0*10*". Note that the second '*' appears in the same bit position for two source state codes. Therefore, it does not impact correct matching.

To address the above three problems, we formulate the following definition. For two source state codes A and B, if they meet the following three conditions:

(a) They all contain at least one '*'.
(b) There is at least one bit position which value is '*' in A, but not in B.
(c) There is still at least another bit position which value is '*' in B, but not in A.

We say that A and B are interleaved. Furthermore, if they meet the following two conditions:

(a) The number of '*' in A is lower than that in B;
(b) For every '*' in A, the corresponding bit position in B is also '*'.

We say that A is a special case of B. In addition, if they meet the following two conditions:

(a) There is '*' in both A and B.
(b) $A = B$.

We say A is equal to B.

The special case problem can be resolved by exploiting the TCAM priority matching mechanism mentioned in Section 1. When two rows are compressed into one, other rows in the same transition table could become its special cases. The priority matching mechanism of TCAM implies that correct matching would not be impacted, provided that the compressed row is moved behind its special case rows. The reason is that TCAM will grant priority in matching to the special case rows located physically ahead of the compressed row.

In our scheme, the following principle is used to guarantee correct matching as well as high compression ratio. After two operations of compression, A and B, are conducted, if the source state code in the result row of A is equal to or interleave with that of B, compression operation with lower compression ratio should be canceled.

The implementation of the feasible compression is shown in lines 26 to 55 of Algorithm 1. Fig. 5 demonstrates the course of feasible compression in the transition table after recoverable compression is executed in Fig. 4, while Fig. 6 shows the results where 48 rows are compressed into 31 rows. In this case, the compression ratio of the feasible compression is $(48 - 31)/48 = 35.4\%$.

Table 1
Parametric statistics of Regex sets.

Regex set	Regex size	State size
snort24	24	8335
snort31	31	4864
snort34	34	9754
bro79	79	4156
bro217	217	6533
cisco110	110	11696

3.3. Proofing the validity of PEDFA

Our compression scheme consists of three components: pre-processing, recoverable compression and feasible compression. If all three components are valid, the whole compression scheme is valid.

The purpose of pre-processing is to perform adequate assignment of identifiers to states in a DFA to facilitate recoverable compression. It does not change the DFA's structure and logic. Therefore, it is irrelevant to the correctness of compression.

In recoverable compression, the compressible condition inhibits the appearance of the special case problem, the interleaving problem and the identity problem. So it is valid.

In feasible compression, although there exists the special case problem, the interleaving problem and the identity problem, it adequately identifies and deals with them. So it is also valid.

Consequently, the false matching problem does not exist in PEDFA.

4. Experimental evaluation

The experimental evaluation in this work consists of three parts: DFA generation, PEDFA implementation, and TCAM simulation. Based on regular expression processing software [23], we have implemented PEDFA and the code has been uploaded to github (see [24]). The TCAM simulator [25] is used to obtain the results of our algorithm. Three performance indicators, power, memory, and throughput consumption, are examined. The Regex sets used in the experiment come from *snort*, *bro*, and *cisco* libraries [7–9]. These Regex sets follow the standard regular expression specification. Table 1 shows information of the sets. The first column shows their names. The second column lists their sizes. The third column is the state size of DFAs output by the Regex processing software. These results are also compared to that of CIDFA.

4.1. Experimental methodology

The TCAM power consumption is an output of the TCAM simulator, and it is related to the TCAM memory consumption. So the TCAM power consumption depends on two important parameters: the block size and the number of active blocks in a search. This work uses eight different block sizes to simulate TCAM performance: 32, 64, 128, 256, 512, 1024, 2048, and 4096. Suppose that there are N TCAM memory entries needed to store the transition table of an input character and the TCAM block size is B. The needed number of TCAM blocks is $X = \lceil N/B \rceil$. The TCAM power consumption is referred to that part needed to activate simultaneously X TCAM blocks, rather than the product of X and the part needed to activate one block. Every character has its own transition table and the size of every compressed transition table may be different. It means that different input characters have the different number of active TCAM blocks, leading to different TCAM power consumption. Suppose that p_i indicates the TCAM power consumption of character i, and the number of characters is C. Thus the total TCAM power consumption is

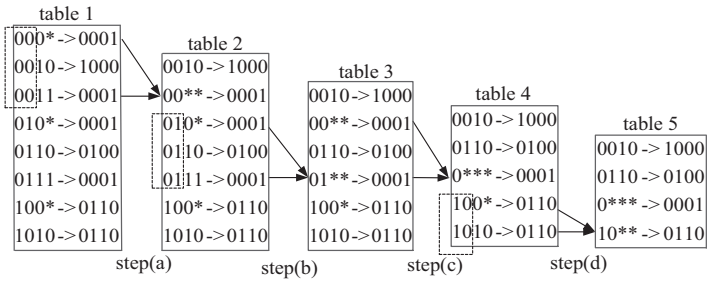

Fig. 5. Feasible compression.

^			a			b		
src		dst	src		dst	src		dst
0000	->	0000	0011	->	0110	0000	->	0000
0011	->	0011	00**	->	0001	0011	->	0011
00**	->	0001	****	->	0110	00**	->	0010
010*	->	0011				010*	->	0011
****	->	0110				****	->	1001
c			d			e		
src		dst	src		dst	src		dst
0001	->	0110	0000	->	0000	0000	->	0000
0010	->	0111	0011	->	0101	0011	->	0011
011*	->	0110	00**	->	0001	00**	->	0001
0***	->	0011	010*	->	0101	0100	->	0011
1001	->	0111	*****	->	1010	0101	->	0100
10**	->	0110				1010	->	1000
						****	->	0110

Fig. 6. Feasible compression of {.*a.*bc} and {.*c.*de}.

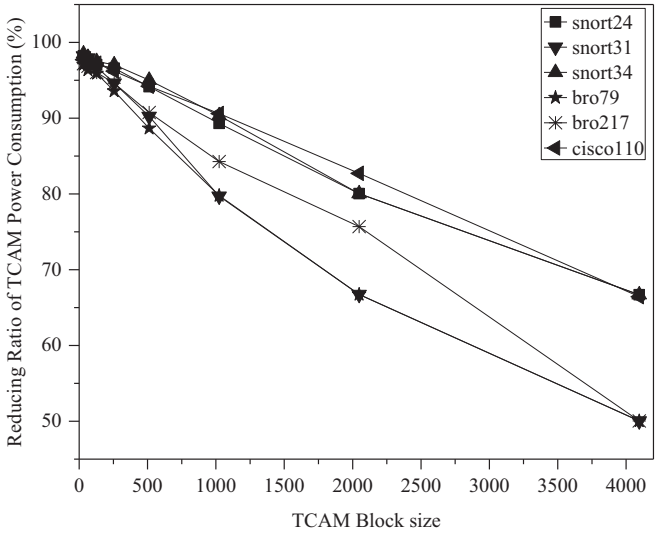

Fig. 7. Reducing ratio of TCAM power consumption.

$P_{total} = \sum_0^C P_i$. The average TCAM power consumption is $p_{avg} = p_{total}/C$. In PEDFA, a TCAM search for an input character involves two steps: searching in character-index blocks and then searching in transition blocks. Suppose that p_{index} is the power consumption of searching in character-index blocks, the total TCAM power consumption on average is $p = p_{avg}+p_{index}$.

TCAM memory consumption S is the product of the bit size of a TCAM entry *bit_size* and the number of TCAM entries N, namely $S = bit\_size*N$. Suppose that S_1 represents TCAM memory consumption of the character-index blocks and S_2 indicates that of the transition blocks. The total TCAM memory consumption is $S = S_1 + S_2$.

TCAM throughput T is used to evaluate its matching speed. Its value is equal to the quotient of the bit number *bit_num* read at a time and the matching delay D in each search, namely $T = bit\_num/D$. The total matching delay mainly consists of two parts: the matching delay D_{index} in searching the character index table and the matching delay D_{trans} in searching the transition table. Therefore, the throughput is $T = bit\_num/(D_{index} + D_{trans})$. Suppose that T_i is the throughput of character i. Thus, the average throughput is$T_{avg} = \sum_0^C T_i/C$.

4.2. Results on power consumption

There are eight different block sizes to simulate TCAM power consumption under CIDFA and PEDFA, respectively. Suppose the TCAM power consumption under CIDFA and under PEDFA are p_1 and p_2, respectively. The power reduction ratio is $R_{power} = (p_2 - p_1)/p_1*100\%$.

Fig. 7 shows the trends of power reduction ratio of the six Regex sets in different TCAM block sizes. The horizontal axis represents the TCAM block sizes and the vertical axis represents the TCAM power reduction ratios. Under CIDFA, the power reduction ratios range from 50.0% to 98.4% in different block sizes. The average reduction ratio is 87.4%. From Fig. 7, we observe that the power reduction ratio is minimal when block size is 4096 and the ratio is at its maximum when block size is 32. It implies that the power reduction ratio decreases with the increase of block sizes. Let's take the Regex set *snort24* to address this issue. In CIDFA, the transition table size of the Regex set is 558445. There are 8335 TCAM entries for every input character on average. There are $\lceil 8335/32 \rceil = 261$ TCAM blocks needed to be activated for an input character when the TCAM block size is 32. In the last block, only 8335 – 260*32 = 15 entries are utilized. The remaining 32 – 15 = 17 entries are unused for searching, but they are still a part of the last TCAM block as TCAM can only be divided into fixed-size blocks. After being compressed by PEDFA, the transition table size of the Regex set is only 28,937. The number of active blocks is different for different input character but the average number is eight blocks. With the increase of the TCAM block size, TCAM memory of the last block for every input character is more wasteful, which in turn leads to a smaller power reduction ratio.

Fig. 8 shows the comparison of the TCAM power consumption under CIDFA and PEDFA when the block sizes are 1024, 2048, and 4096, respectively. The power consumption under PEDFA is significantly lower than under CIDFA. The other two characteristics are the power consumption is the same for different Regex sets under PEDFA in terms of the block size, and decreases with decreased block size.

4.3. Results on memory consumption

The TCAM memory consumption comes from two aspects: the character index and the transition tables. CIDFA does not care about memory consumption, while PEDFA conducts compression to transition entries in the transition tables.

Table 2 shows the comparisons of TCAM memory consumption under CIDFA and PEDFA. From Table 2, we can observe that the

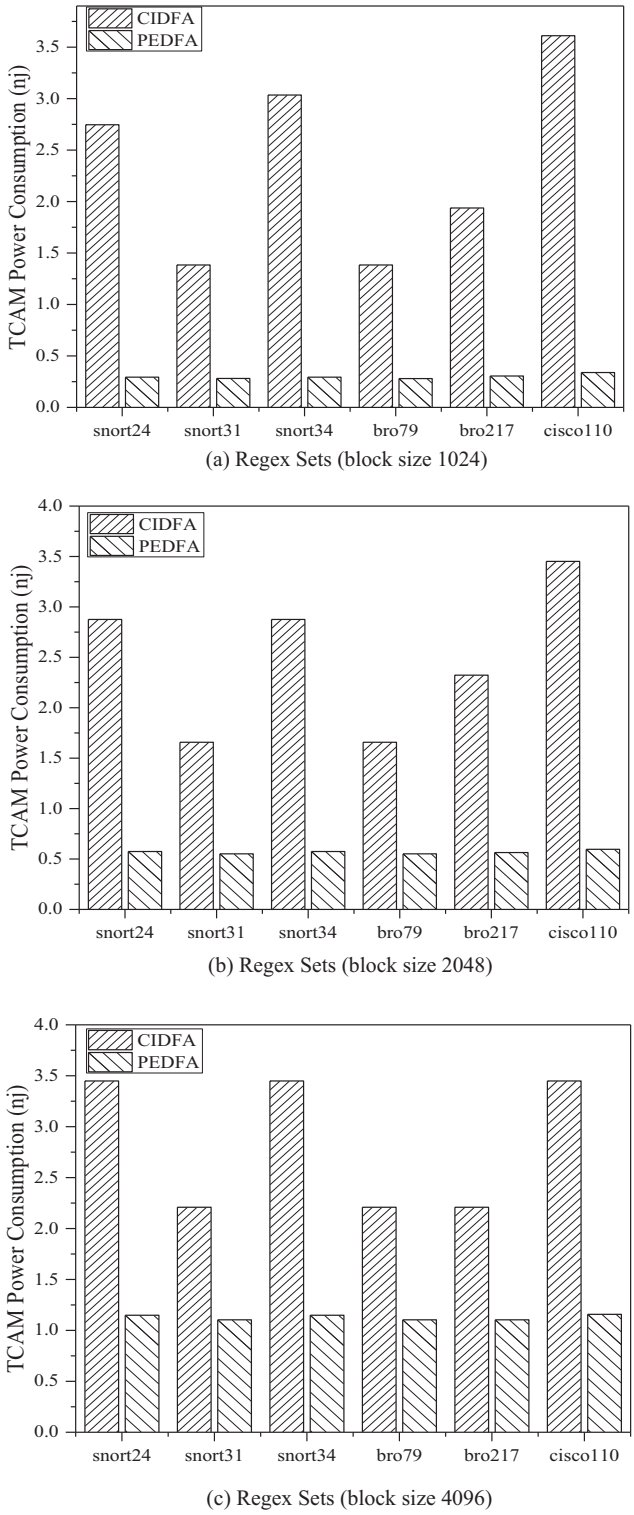

Fig. 8. Comparison of TCAM power consumptions.

Table 2
TCAM memory consumption.

Regex set	CIDFA/MB	PEDFA/MB	Ratio
snort24	7.458	0.387	94.8%
snort31	4.645	0.230	95.0%
snort34	9.509	0.434	95.4%
bro79	3.042	0.302	90.1%
bro217	8.992	0.652	92.8%
cisco110	11.558	1.016	91.2%

Table 3
TCAM throughput of Regex 24.

Block size	CIDFA/Gbps	PEDFA/Gbps	Ratio
32	3.384	13.187	289.7%
64	3.384	11.357	235.6%
128	3.384	8.887	162.6%
256	3.384	6.077	79.6%
512	3.384	3.564	5.3%
1024	1.784	1.784	0.0%
2048	0.757	0.757	0.0%
4096	0.274	0.274	0.0%

Table 4
TCAM throughput on average.

Regex set	CIDFA/Gbps	PEDFA/Gbps	Ratio
snort24	2.467	5.736	132.5%
snort31	2.831	5.859	107.0%
snort34	2.467	5.748	133.0%
bro79	2.831	5.818	105.5%
bro217	2.831	5.581	97.1%
cisco110	2.467	5.385	118.2%

average reduction ratio on TCAM memory consumption is 93.2% under PEDFA as compared to CIDFA. Under CIDFA, there are a large number of transition entries in the transition tables. PEDFA compresses them as much as possible. Therefore, the TCAM memory consumption under PEDFA is significantly lower than under CIDFA.

4.4. Results on throughput

Although TCAM performs a query in parallel pattern, its matching speed is actually related to its active memory size. The larger active memory size is, the slower TCAM matching speed is. Therefore, TCAM matching speed can be improved by a decrease in TCAM memory consumption. This work uses throughput to evaluate TCAM matching speed.

In the experiment, the character encoding used is ASCII. Therefore, the formula of throughput is $T = 8/(D_{index} + D_{trans})$. Its unit is Gbps. Table 3 compares the throughput of Regex set *snort24* under CIDFA and PEDFA in different block sizes. The last column is the improvement ratio of throughput. In Table 3, the improvement ratio is minimum when the block size is 4096, while it is at maximum when block size is 32. In other words, the improvement ratio of throughput decreases with the increase of block size. Although PEDFA conducts compression to transition entries, the matching complexity remains invariant. So the throughput can improve under PEDFA.

Table 4 shows the average throughput of six Regex sets used in the experiment, respectively. The improvement ratios range from 97.1% to 137.5% in six Regex sets, and the average improvement ratio reaches 114.7% under PEDFA as compared to CIDFA.

5. Conclusion

In TCAM-based implementations of regular expression matching, the structured storage of state transition entries in TCAM enables only those necessary blocks to be activated, while those non-involved blocks can stay in the dormant state. Thus, the TCAM power consumption is decreased. If every transition table is compressed, the TCAM power consumption can be further reduced. The assignment of identifiers to states in a DFA affects the compression ratio. Assigning sequential identifiers to those associated states can result in an improvement in compression ratio. The wildcard function provided in TCAM enables two rows in a transition table to be compressed into one row recursively. However, the compression could lead to

the interleaving problem, the special case problem and the identity problem. The special case problem can be resolved by exploiting the priority matching mechanism provided in TCAM. The interleaving problem and the identity problem can be identified. After two operations of compression are conducted respectively, if the source state codes in the two result rows are identical or interleaved, the operation of compression with lower compression ratio should be canceled, leading to maximum compression ratio. This paper presents a complete implementation of the identifier assignment and transition table compression. Experimental results on real-world Regex sets verify that the proposed scheme is significantly more effective, reducing 87.4% of power consumption, 93.2% of memory space, and improving throughput up to 114.7% on average compared to prior work.

Acknowledgments

The authors would like to sincerely thank the editor and the anonymous reviewers for their careful reviews and helpful improvement suggestions. This work was supported in part by national natural science foundation of China under grant 61272401, key science & technology plan of Hunan province under grant 2013GK2003, and the prospective research project on future networks of Jiangsu future networks innovation institute under grant by 2013095-1-05.

References

[1] M. Becchi, P. Crowley, A hybrid finite automaton for practical deep packet inspection, in: Proceedings of International Conference on Emerging Networking EXperiments and Technologies, ACM CoNEXT, 2007, doi:10.1145/1364654.1364656.
[2] S. Jayashree, N. Shivashankarappa, Deep packet inspection using ternary content addressable memory, in: Proceedings of International Conference on Comutational Intelligence for Modeling Control and Automation, IEEE CIMCA, 2014, pp. 441–447, doi:10.1109/CIMCA.2014.7057841.
[3] Y.-K. Chang, M.-L. Tsai, C.-C. Su, Improved TCAM-based pre-filtering for network intrusion detection systems, in: Proceedings of International Conference on Advanced Information Networking and Applications (AINA), 2008, pp. 985–990, doi:10.1109/AINA.2008.120.
[4] Y. Weinsberg, S. Tzur-David, D. Dolev, et al., High performance string matching algorithm for a network intrusion prevention system (NIPS), in: Proceedings of Workshop on High Performance Switching and Routing, 2006, doi:10.1109/HPSR.2006.1709697.
[5] Y.-Y. Gong, Q.-R. Liu, X. Shan, et al., A novel regular expression matching algorithm based on multi-dimentional finite automata, in: Proceedings of High Performance Switching and Routing, IEEE HPSR, 2014, pp. 90–97, doi:10.1109/HPSR.2014.6900887.
[6] M. Becchi, P. Crowley, Efficient regular expression evaluation: theory to practice, in: Proceedings of The 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM/IEEE ANCS, 2008, pp. 50–59, doi:10.1145/1477942.1477950.
[7] The bro network security monitor, 2015, http://www.bro.org.
[8] Snort rules, 2015, http://www.snort.org.
[9] Cisco Services for Intrusion Prevention System (IPS), 2015, http://www.cisco.com.
[10] S.K. Maurya, L.T. Clark, A dynamic longest prefix matching content addressable memory for IP routing, IEEE Trans. Very Large Scale Integr. Syst. 19 (6) (2011) 963–972, doi:10.1109/TVLSI.2010.2042826.
[11] Z. Ullah, M.K. Jaiswal, R.-C.-C. Cheung, Z-TCAM: an SRAM-based architecture for TCAM, IEEE Trans. Very Large Scale Integr. Syst. 23 (2) (2015) 402–406, doi:10.1109/TVLSI.2014.2309350.
[12] B. Agrawal, T. Sherwood, Modeling TCAM power for next generation network devices, IEEE Symposium on Performance Analysis of Systems and Software (2006) 120–129, doi:10.1109/ISPASS.2006.1620796.
[13] Y. Ma, S. Banerjee, A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification, ACM SIGCOMM (2012) 335–346, doi:10.1145/2342356.2342428.
[14] L.-X Ding, K. Huang, D.-F. Zhang, Low-power TCAMs for regular expression matching, J. Commun. 25 (8) (2014) 162–168, doi:10.3969/j.issn.1000-436x.2014.08.020.
[15] C.R. Meiners, J. Patel, E. Norige, et al., Fast regular expression matching using small TCAM, IEEE/ACM Trans. Netw. 22 (1) (2014) 94–109, doi:10.1109/TNET.2013.2256466.
[16] S. Kumar, S. Dharmapurikar, F. Yu, et al., Algorithms to accelerate multiple regular expressions matching for deep packet inspection, ACM SIGCOMM (2006) 339–350, doi:10.1145/1159913.1159952.
[17] M. Becchi, P. Crowley, An improved algorithm to accelerate regular expression evaluation, in: Proceedings of The 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM/IEEE ANCS, 2007, pp. 145–154, doi:10.1145/1323548.1323573.
[18] S. Kumar, J. Turner, J. Williams, Advanced algorithms for fast and scalable deep packet inspection, in: Proceedings of The 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM/IEEE ANCS, 2006, pp. 81–92, doi:10.1145/1185347.1185359.
[19] K. Peng, S. Tang, M. Chen, et al., Chain-based DFA deflation for fast and scalable regular expression matching using TCAM, in: Proceedings of The 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM/IEEE ANCS, 2011, pp. 24–35, doi:10.1109/ANCS.2011.13.
[20] K. Huang, L.X. Ding, G.G. Xie, et al., Scalable TCAM-based regular expression matching with compressed finite automata, in: Proceedings of The 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM/IEEE ANCS, 2013, pp. 83–93.
[21] O. Naoya, W.J. Gross, T. Hanyu, A low-energy variation-tolerant asynchronous TCAM for network intrusion detection systems, in: Proceedings of The International Symposium on Asynchronous Circuits and Systems IEEE ASYNC, 2013, pp. 8–15, doi:10.1109/ASYNC.2013.16.
[22] A. Bremler-Barr, D. Hay, Y. Koral, CompactDFA: scalable pattern matching using longest prefix match solutions, 22, 2014, pp. 415–428, doi:10.1109/TNET.2013.2253119.
[23] Regular expression processor, 2015, http://regex.wustl.edu/index.php/Main_Page.
[24] Lhmily/regex_compress_tcam, 2015, https://github.com/Lhmily/regex_compress_tcam.git.
[25] Memory modeling, 2015, http://www.cs.ucsb.edu/~arch/mem-model/.