Scalable TCAM-based Regular Expression Matching with Compressed Finite Automata

Kun Huang[1], Linxuan Ding[2], Gaogang Xie[1], Dafang Zhang[2], Alex X. Liu[3], Kave Salamatian[4]

[1]Institute of Computing Technology, CAS
Beijing 100190, China
{huangkun09, xie}@ict.ac.cn

[2]Hunan University
Changsha 410082, China
{lxding, dfzhang}@hnu.edu.cn

[3]Michigan State University
East Lansing 48823, U.S.A
alexliu@cse.msu.edu

[4]Universite de Savoie
Annecy 74944, France
kave.salamatian@univ-savoie.fr

ABSTRACT

Regular expression (RegEx) matching is a core function of deep packet inspection in modern network devices. Previous TCAM-based RegEx matching algorithms a priori assume that a deterministic finite automaton (DFA) can be built for a given set of RegEx patterns. However, practical RegEx patterns contain complex terms like wildcard closure and repeat character, and it may be impossible to build a DFA with a reasonable number of states. This results in prior work to being infeasible in practice. Moreover, TCAM-based RegEx matching is required to scale to a large-scale set of RegEx patterns. In this paper, we propose a compressed finite automaton implementation called (CFA) for scalable TCAM-based RegEx matching. CFA is designed to reduce TCAM space by using three compression techniques: transition, character, and state compressions. Experiments on realistic RegEx pattern sets show CFA highly outperforms previous solutions in terms of TCAM space, matching throughput, and TCAM power consumption.

Categories and Subject Descriptors

C.2.5 [**Computer Communication Networks**]: Local and Wide-Area Networks-Internet: C.2.6 [**Computer Communication Networks**]: Internetworking-Routers

General Terms

Algorithms, Design, Performance, Security

Keywords

Intrusion detection, signature matching, regular expression, compressed finite automaton

1. INTRODUCTION

Regular expression (RegEx) matching is a key function of deep packet inspection in modern network devices on the Internet ranging from firewall to network intrusion detection and prevention systems (NIDPS). Due to their powerful and flexible expressiveness, RegExes are typically used to represent complex patterns such as worm/virus signatures [1]. Several popular NIDPS such as Snort [7], Bro [8], Tipping Point X505, and Cisco network security appliances, have used RegEx matching algorithms to detect security sensitive scenarios by comparing a packet's payload against a set of predefined rules expressed as RegEx patterns. Application layer (layer 7) RegEx matching based filters have also been implemented in the Linux operating system.

To keep up with line speeds, RegEx patterns must be matched in a single pass over the input. RegEx matching is typically performed using either deterministic finite automata (DFAs) or non-deterministic finite automata (NFAs). DFAs require one state traversal per input character, but their construction leads to an explosion of the number of states in the state space. In contrast, NFAs make brief representations of RegEx patterns with only a few states, but require keeping track of multiple possible active states for each input character, leading to higher computation overhead. In other words, DFAs are time-efficient but space-inefficient, while NFAs are space-efficient but time-inefficient. Therefore, fast and scalable RegEx matching is a challenging issue, and the key is to implement a time/space-efficient finite automaton.

Recently, TCAM-based RegEx matching algorithms [2-5] exploiting TCAM's parallelism and wildcard search abilities have been proposed as a promising approach to achieve high speeds. These TCAM-based solutions use one TCAM entry to encode each line of a state transition table that is built from a set of RegEx pattern. Each TCAM entry is composed of two parts: a TCAM part stores a source state and an input character, and a companion SRAM part is thereafter used to store a destination state. Each TCAM part is encoded in ternary format, where each bit is either 0, 1 or * (don't care). TCAM circuits compare in parallel a given search key against all TCAM entries, and report the position of the first-matching entry. Using TCAM, RegEx matching algorithms work by iteratively searching the combination of the current state of DFA representing RegEx patterns and the input character to find the index of the position in SRAM that contains the next state of DFA.

978-1-4799-1640-5/13/$31.00 ©2013 IEEE

While TCAM chips facilitate high speeds and concise representations, they are confronted for a large-scale set of RegEx patterns, with scalability issues. As TCAM chips are scarce and expensive, TCAM-based RegEx matching is required to scale well in space, throughput, and power consumption. The number of searched entries in a TCAM dominates the matching throughput and power consumption. Therefore it is critical to reduce TCAM space needed to represent a set of RegEx patterns. This is achieved by encoding multiple transitions in a single TCAM entry, which in turn can improve RegEx matching throughput and reduce TCAM power consumption.

Several TCAM-based DFA implementations [4, 5] have been proposed for fast and scalable multi-pattern matching. Unfortunately, RegEx patterns in practice contain complex terms like wildcard closure ".*", repeat character "{}", etc., and it may be impossible to build a DFA with a reasonable number of states. This results in these TCAM-based DFA implementations to being inapplicable in general settings.

To eliminate state explosion, extended finite automata (XFAs) have been proposed in the literature [9, 10]. XFA extends the standard DFA with auxiliary variables and simple instructions attached to states. However, while XFA achieves significant reductions in state space, the TCAM-based XFA implementation suffers from a transition-space explosion problem. This is caused by a large number of redundant transitions in XFA, which requires a large amount of TCAM space. Furthermore, multi-stride XFA leads to an exponential increase in transition space, which exacerbates the space explosion problem.

In this paper, we propose a compressed finite automaton implementation called CFA for space-efficient TCAM-based RegEx matching. CFA is a concise representation of XFA in a small TCAM. To achieve CFA, we propose three compression techniques: transition, character, and state compressions. These techniques reduce TCAM space by encoding multiple transitions in one TCAM entry. First, transition compression uses transition grouping to combine multiple transitions with the same input character and destination state by encoding multiple source states in a wildcard state. Second, character compression uses character grouping and merging to combine multiple transitions with the same source and destination state by encoding multiple input characters in a ternary character. Third, state compression uses masked state encoding to combine multiple transitions with the same input character and destination state by encoding multiple source states in a ternary masked code. We greatly improve RegEx matching throughput by generalizing to multi-stride CFA.

Experiments on realistic RegEx pattern sets show that CFA dramatically outperforms previous solutions in terms of space, matching throughput, and power consumption. In our test, CFA reduces TCAM space as well as TCAM power consumption by up to 83% and up to 95% compared to optimized DFA [4] and XFA respectively. In addition, CFA achieves RegEx matching throughput of up to 10.9Gbps due to significant space reductions.

The main contributions of this paper are the following:

- We propose CFA, a space-efficient TCAM-based implementation of RegEx matching. CFA targets a large-scale set of RegEx patterns for which a DFA cannot be built. To reduce TCAM space, we propose three compression techniques on transition, character, and state, respectively. We also generalize to multi-stride CFA.
- We conduct experiments on realistic RegEx pattern sets obtained from Snort and Bro. The results show that CFA outperforms previous solutions in terms of space, matching throughput, and power consumption. CFA reduces TCAM space by up to 83% and up to 95% compared to optimized DFA [4] and XFA respectively. CFA achieves faster RegEx matching speeds as well as smaller TCAM power consumptions.

The rest of this paper is organized as follows. Section 2 discusses the related work. We describe CFA with three compression techniques and extend to multi-stride CFA in Section 3. Section 4 reports experimental results on real-world RegEx pattern sets. Section 5 concludes the paper.

2. RELATED WORK

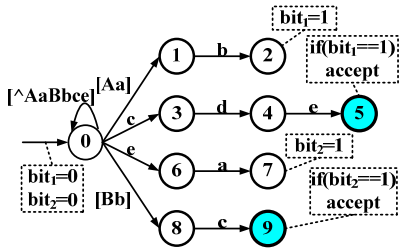
As it plays an important role in several network functions such as intrusion detection and prevention, and application-level traffic identification, RegEx matching has attracted intensive research in recent years. Hardware-based RegEx matching solutions have been proposed to handle high-speed packets. These solutions fall into two categories: FPGA/ ASIC-based [9-28] and TCAM-based [2-5].

To achieve high speeds, RegEx matching algorithms are generally implemented in FPGA circuits or ASIC chips. FPGAs are programmable with intrinsic reconfiguration capability and parallelism. Many efforts have focused on the implementations of FPGA-based NFAs [11-14] or parallel DFAs [15], where each state is encoded in a flip-flop to allow each character to be processed in constant time. While FPGA-based solutions can achieve high throughput of about 10Gbps, they are limited by high deployment cost and update overhead, and have slower clock speeds than ASIC chips.

ASIC-based solutions implement a DFA in RAM. The main issue is to use small SRAMs to represent a large-scale set of patterns. Prior work on ASIC-based string matching [16-19] and RegEx matching [20-28] has targeted reducing the amount of memory needed to represent a DFA. In particular, Kumar et al. [21, 22] have proposed D^2FA to reduce DFA memory requirements at the cost of longer matching times by replacing multiple transitions between states with a single default transition. But D^2FA cannot overcome the

Table 1: Number of DFA states for RegEx pattern sets.

RegEx Pattern Set	Snort1	Snort2	Bro1	Bro2	Bro3
# RegExes	23	40	227	79	406
# RegExes Using ".*"	23	15	18	79	406
# RegExes Using "{}"	0	1	3	3	18
# States	49,128	39,502	>600,000	3,644	>563,475



	A	a	B	b	c	d	e	^Σ
0	1	1	8	8	3	0	6	0
1	1	1	8	2	3	0	6	0
2	1	1	8	8	9	0	6	0
3	1	1	8	8	3	4	6	0
4	1	1	8	8	3	0	5	0
5	1	7	8	8	3	0	6	0
6	1	7	8	8	3	0	6	0
7	1	1	8	2	3	0	6	0
8	1	1	8	8	9	0	6	0
9	1	1	8	8	3	4	6	0

Figure 1: XFA for two RegEx patterns /.*[Aa]b.*cde/ and /.*ea.*[Bb]c/.

state-space explosion problem that comes along with DFA. However, these ASIC-based solutions achieve space efficiency at the cost of introducing overhead and inefficiency in the matching process, limiting the throughput of RegEx matching.

More recently, TCAM-based implementations have been proposed as a promising approach to fast and scalable multi-pattern matching. Early TCAM-based techniques [2, 3] are based on the Aho-Corasick algorithm [6] and designed for plain string matching. Yu et al. [2] have proposed a TCAM-based multi-byte multiple-string matching algorithm with limited support for wildcards. Alicherry et al. [3] have proposed a state-encoding scheme for implementing an efficient multi-character multiple-string matching algorithm in small TCAMs.

However, these TCAM-based solutions above could not efficiently handle a full set of RegEx patterns with specific terms. Meiners et al. [4] have proposed the first TCAM-based RegEx matching algorithm called optimized DFA. By using the optimization methods such as transition sharing and table consolidation, this scheme reduces TCAM space and improves RegEx matching speeds. Peng et al. [5] have proposed a chain-based DFA deflation method for a scalable TCAM-based implementation by exploiting the structural connection between NFA and DFA. These TCAM-based RegEx matching solutions a priori assume that a DFA can be built for a given set of RegEx patterns. Unfortunately, RegEx patterns in practice contain complex terms like wildcard closure ".*", repeat character "{}", etc., and it may be impossible to build a DFA with a reasonable number of states. Table 1 shows the size of resulting DFAs for five RegEx pattern sets obtained from Snort [7] and Bro [8]. We see that DFAs are too large to be built for two of five pattern sets. Consequently, this results in prior TCAM-based DFA implementations to being infeasible in general settings.

XFAs [9, 10] are proposed to eliminate state explosion. XFA extends the standard DFA with auxiliary variables and simple instructions attached to states for manipulating these variables. Fig. 1 illustrates a simple example of XFA for two RegEx patterns /.*[Aa]b.*cde/ and /.*ea.*[Bb]c/. "[Aa]" and "[Bb]" use the class characters "[]", indicating that a class of characters, i.e., 'A' or 'a' for "[Aa]", is permissible. States 2 and 7 in XFA have an assignment instruction that indicates that a sub-string (i.e., "[Aa]b" or "ea") has appeared in the input by setting a variable bit_1 or bit_2 to 1. States 5 and 9 are accepting states associated with a check instruction that checks the variable bit_1 or bit_2 to test acceptance conditions. Fig. 1 also depicts the state transition table of XFA with from state 0 to 9 and an alphabet set Σ = {A, a, B, b, c, d, e}.

XFAs can be built directly from a set of RegEx patterns, avoiding the state-space explosion of DFA construction. Thus, XFA instead of DFA can be implemented in TCAMs for fast and scalable RegEx matching. However, while XFA achieves significant reductions in state space, the TCAM-based XFA implementation has a transition-space explosion problem. This is caused by the large number of redundant transitions in XFA, which requires a large amount of TCAM space. For instance, RegEx terms like character classes (e.g., "[a-z]") and negation (e.g., "[^0-9]") can be translated into a large number of transitions, each one corresponding to one possible input character. In addition, multi-stride XFA is typically used to improve RegEx matching throughput by using multiple characters per transition. However, this results in an exponential increase in transition space, which can worsten the space explosion problem, and require more TCAM space. Hence, it is crucial to efficiently implement an XFA in a small TCAM.

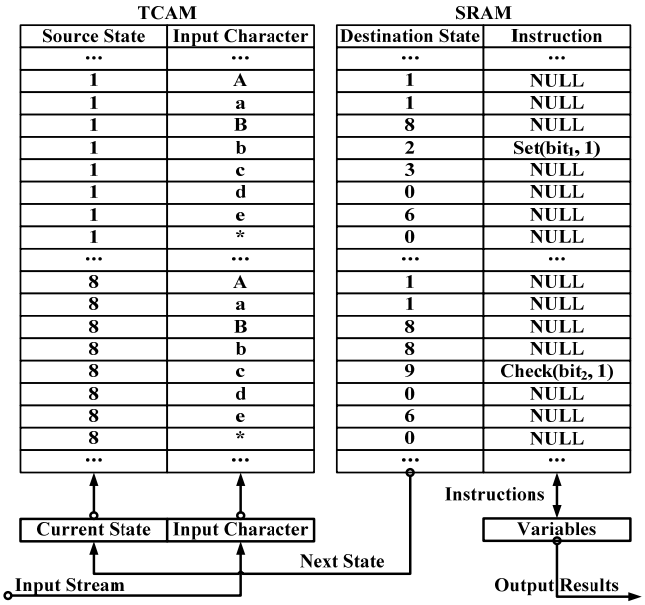


Figure 2: TCAM-based architecture for implementing an XFA

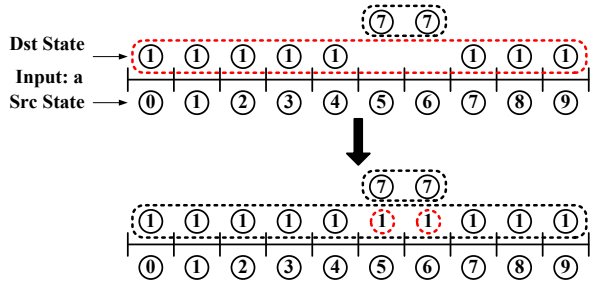
3. COMPRESSED FINITE AUTOMATA

In this section, we first present CFA, a TCAM-based RegEx matching architecture that can efficiently implement an XFA with each transition of its state transition table encoded in a single TCAM entry. We then describe three compression techniques to reduce the number of TCAM entries needed for implementing a CFA. Finally, we extend these techniques to multi-stride CFA.

3.1 TCAM-based Architecture

Let's first describe a TCAM-based architecture for implementing XFA-based RegEx matching. As shown in Fig. 2, the architecture consists of a TCAM table and a logic circuit associated with some registers. A TCAM entry is used to encode a transition in XFA, i.e., the number of TCAM entries is equal to the number of transitions. Each TCAM entry is physically composed of a TCAM and a SRAM part. The TCAM part stores a source state and an input character that are matched against a search key. The SRAM part stores the corresponding destination state and attached instructions that are returned as the lookup result when a match occurs in TCAM. The logic circuit executes the instructions associated with the returned state, and the registers store auxiliary variables. There are two instructions: *Set()* is to set a variable (i.e., bit_1 or bit_2) to 0 or 1, and *Check()* is to check a variable value.

The matching circuits concatenate the current state and a character from the input stream to form a search key, and then compare the key against all TCAM entries. If the search key is matched, TCAM outputs the index of the first-match entry, and then feeds the values at the same index in SRAM, i.e., the destination state and the instructions, to the logic circuits, updating the registers and the current state.



TCAM

Source State	Input Character
…	…
5	a
6	a
*	a
…	…

SRAM

Destination State	Instruction
…	…
7	Set(bit_2, 1)
7	Set(bit_2, 1)
1	NULL
…	…

Figure 3: Transition grouping on input charatacter 'a'.

TCAM

Source State	Input Character
5	a
6	a
1	b
7	b
2	c
8	c
3	d
9	d
4	e
*	A
*	a
*	B
*	b
*	c
*	e
*	*

SRAM

Destination State	Instruction
7	Set(bit_2, 1)
7	Set(bit_2, 1)
2	Set(bit_1, 1)
2	Set(bit_1, 1)
9	Check(bit_2, 1)
9	Check(bit_2, 1)
4	NULL
4	NULL
5	Check(bit_1, 1)
1	NULL
1	NULL
8	NULL
8	NULL
3	NULL
6	NULL
0	NULL

Figure 4: TCAM table after transition compression.

Next we will describe three compression techniques in CFA, which can greatly reduce the number of needed entries in TCAMs.

3.2 Transition Compression

We observe that transitions coming from different source states transit to the same destination state on the same input character. Transition compression is to combine multiple transitions similar to be above into a single TCAM entry. This is done in two steps: transition grouping and merging. Transition grouping divides all the transitions on the same input character into groups, each with the same destination state. In the merging step, all the transitions in the largest group are merged into a single transition with its source state set to a wildcard "*" (don't care) state, while the transitions in other groups remain unchanged.

Transition compression is used to remove many original transitions and replace them with few merged transitions. This results in semantic ambiguity caused by several TCAM entries matching a query. Our solution is to apply a most specific first-match approach (similar to longest prefix match in IP lookup). This is done by using a sorting method that reorders the transitions in the TCAM table such that the

CompressTransition(S, Σ, δ, Instruction)

```
//S is a finite set of XFA states.
//Σ is a finite set of input characters called the alphabet.
//δ is a transition function of XFA
1: TransitionGrouping();
//Divides all transitions having the same input character into groups.
2: TransitionMergeing() {
3:  for (all transitions in the same group) do
4:    if (δ in the largest group) then
5:      δ. Priority = 1;
6:      δ. SRC ID = *;
7:    else
8:      δj. Priority = 2;
9:    end if
10: end for
11: for (all transitions.DST ID =0) do
12:   δ. Priority = 0;
13:   δ. SRC ID = *;
14:   δ. Input = *;
15: end for }
16: Sort (δ.Begin(),δ.End(),δ. Priority Desc);
17: return TCAMresult;
```

Algorithm 1: Transition compression

most specific entry is returned by the first-match operation of TCAM search. This method works well by assigning three levels of priority to all TCAM entries. All transitions without a wildcard state are assigned the highest priority 2. Merged transitions with wildcard "*" only in the source state are assigned the middle priority 1. Other merged transitions with wildcard in both the source state and the input character are assigned the lowest priority 0. After that, we sort the TCAM table in descending order of the priority. The algorithm of transition compression is given in Algorithm 1.

Fig. 3 illustrates an example of transition grouping on input character 'a'. We observe that on input character 'a' all the states in XFA (see Fig. 1) transit to state 1 or 7. So these transitions are divided into two groups: one group with eight transitions to state 1, and a second with two transitions to state 7. Using transition grouping we combine the eight transitions in the first group into a merged transition with wildcard source state "*", and assign it a lower priority 1. Other two transitions remain unchanged. Consequently, eight transitions are removed from the TCAM table, and replaced by a single merged transition added at the tail of the table.

Fig. 4 shows the complete TCAM table after transition compression. XFA in Fig. 1 requires initially a total of 10× 8=80 entries. Using transition compression we reduce the number of entries to 16 in the TCAM table as seen in Fig. 4. The first nine transitions remain unchanged with a priority 2, the next eight merged transitions are added with a priority 1, and the last is a wildcard transition with a priority 0. In this example transition compression achieves a reduction in the number of TCAM entries by a factor of five.

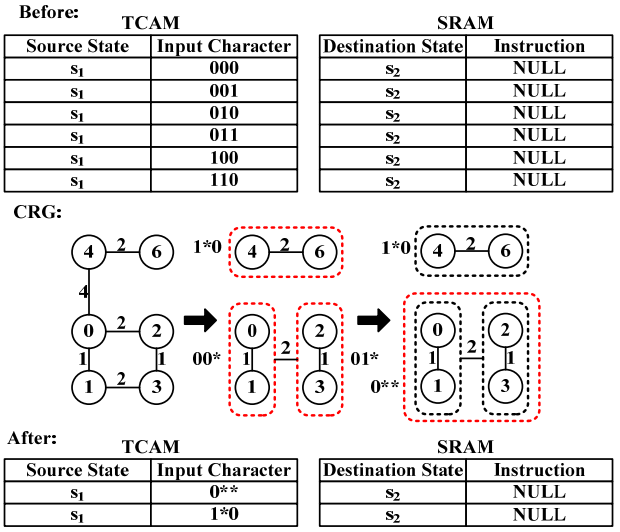


Figure 5: TCAM table using heuritic character merging.

3.3 Character Compression

We also observe that many transitions have the same source and destination state but happen with different input characters. Character compression uses the idea of grouping input characters into a ternary character and merging the related transitions into a single one. Character compression consists of two steps: character grouping and merging. Character grouping divides the transitions with the same source state into groups, each with the same destination state. All the transitions in the largest group are combined into a merged transition with wildcard character "*" (don't care), and assigned a priority 1. These original transitions are removed, and the merged transition is added into the TCAM table in descending order of the priority.

Character merging is the second step that combines the transitions in the groups besides the largest one by encoding potential multiple input characters in ternary format. This step is applied to the transitions that share the same source and destination state and differ only in some bits of their input characters. For instance, as shown in Fig. 5, the first two transitions are from the same source state s_1 to the same destination state s_2 but on different input binary characters "000" and "001". We therefore merge the two transitions into a single one with ternary character "00*". However, we must adopt a fast way of detecting these characters that can be merged and represented jointly by ternary characters.

We propose a heuristic approach to character merging. This is done by iteratively encoding two input characters in a ternary string. The merging procedure works as follows. For a group of transitions with the same source and destination state, we construct a character relationship graph (CRG), where each vertex denotes a character, and each edge denotes a prefix distance between characters (the size of common bit prefixes between characters). Each edge value is essentially calculated as the hamming distance between characters only if the edit distance between them is one.

CompressCharacters($TCAM_{result}$)

1: **CharactersGrouping**();
//Divides all transitions having the same source and destination state into groups.
2: **for** (δ in the largest group) **do**
3: δ. Priority = 1;
4: Merging transitions with *;
5: **end for**
6: **CalculateDistance**(δ_i, δ_j) {
7: Computing prefix distance;
8: Construct **CRG**; }
9: **HeuristicMerging**(**CRG**) {
10: **Sort**(edges, ***Edge.value Desc***);
11: Build Sub-graphs according to the edge value;
12: Repeated until no more partitions;
13: Merging edges with *; }
14: **return** $TCAM_{result}$;

Algorithm 2: Character compression.

Thereafter, we iteratively partition the CRG into disjoint sub-graphs according to the edge value, i.e., the edge with the smallest value is preferentially chosen to partition all the vertexes into groups. We then merge all the characters in each sub-graph into a ternary string, and continue to compute the prefix distance between any pair of sub-graphs. This process is repeated until no more partitions are possible.

Fig. 5 illustrates a simple example of heuristic character merging. Before character merging, the TCAM table had six transitions from source state s_1 to destination state s_2 on input binary characters "000", "001", "010", "011", "100", and "110" respectively. We construct a CRG for these transitions as shown in Fig. 5. As the smallest edge value is 1, the CRG is divided into three sub-graphs. We see that vertexes 0 and 1 are merged into ternary codes"00*", vertexes 2 and 3 are merged in "01*", and vertexes 4 and 6 are merged in "1*0". We continue to compute the prefix distance between all sub-graphs, and iteratively merge sub-graphs into a single sub-graph, i.e., "00*" and "01*" are merged into "0**", while "1*0" remains unchanged. Using character merging, the TCAM table has only two remaining entries with input characters "0**" and "1*0". The algorithm of character compression and heuristic merging is shown in Algorithm 2.

Fig. 6 shows the complete TCAM table after character compression. Before character compression, there are 16 entries in the table, among which two transitions are on input upper-case characters (i.e., 'A' or 'B'), and another two transitions are on lower-case input characters (i.e., 'a' or 'b'). We use character compression to encode input upper-case and lower-case characters in ternary codes. Consequently, as seen in Fig. 6, four original transitions are combined in two merged transitions, and the table remains only 14 entries.

TCAM		SRAM	
Source State	**Input Character**	**Destination State**	**Instruction**
5	**01100001**	**7**	**Set(bit_2, 1)**
6	**01100001**	**7**	**Set(bit_2, 1)**
1	**01100010**	**2**	**Set(bit_1, 1)**
7	**01100010**	**2**	**Set(bit_1, 1)**
2	**01100100**	**9**	**Check(bit_2, 1)**
8	**01100100**	**9**	**Check(bit_2, 1)**
3	**01100100**	**4**	**NULL**
9	**01100100**	**4**	**NULL**
4	**01100101**	**5**	**Check(bit_1, 1)**
*****	**01*00001**	**1**	**NULL**
*****	**01*00010**	**8**	**NULL**
*****	**01100011**	**3**	**NULL**
*****	**01100101**	**6**	**NULL**
*****	************	**0**	**NULL**

Figure 6: TCAM table after character compression.

CompressState ($TCAM_{result}$)

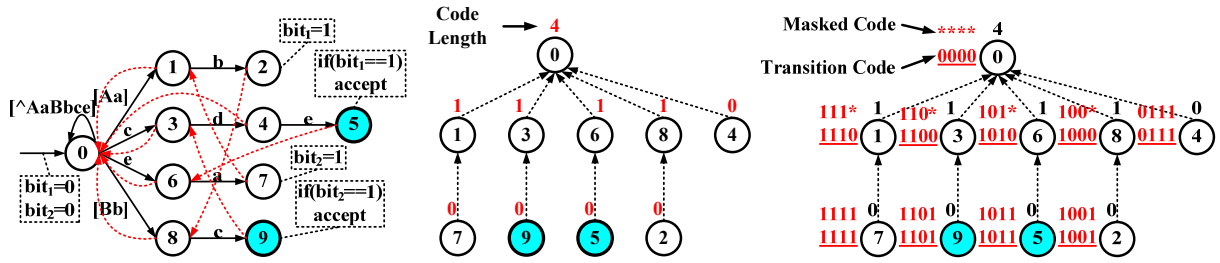
1: $T_{forward}$=**BuildForwardTree**(XFA);
//directed edges from a state of depth i to a state of depth i+1.
2: Build Forward Transitions ;
// a directed edge from a state of depth i to a state of depth j (j<i).
3: Tree = **BuildTree**(Forward Transitions);
4: **CalculateCodeLength**(Tree) {
5: **if** (TreeNode.childCount = 0) **then**
6: codeLength = 0;
7: **else**
8: codeLength = log2(1+$\sum_i$(**Pow**(2, $codeLength_i$)));
//Compute Code Length for each state in a ***bottom-up*** manner.
9: **end if }**
10:**CalculateMaskAndTransition**(Tree) {
11: **CreateBinaryTree**(Depth= **Max**(codeLength));
12: **Traverse**(Tree ***top-down***);
13: Sort(States, ***Code Length Desc***);
14: MaskedCode = the larger value not yet assigned;
15: TransitionCode= the lower value not yet assigned; }
16:SRC ID = MaskedCode;
17:DST ID = TransitionCode;
18:**return** $TCAM_{result}$;

Algorithm 3: State compression.

3.4 State Compression

Even though we use transition compression to settle the status of transitions with the same input character and destination state, further compression can be achieved by using masked state encoding. It is what we target to achieve state compression by encoding multiple source states in ternary codes. This is based on the observation that a state ID in XFA is chosen arbitrarily without specific meaning, which provides an opportunity for state encoding. Using masked state encoding, we would be possible to combine multiple transitions into a single TCAM entry.

We assign to each state in XFA two codes that replace its ID: transition code and masked code. Transition code is a unique code consisting of 0 and 1 that is used to indicate a destination state. Masked code is a ternary code consisting of 0, 1, or * that is used to indicate multiple source states. The algorithm of state compression is shown in Algorithm 3. Masked state encoding consists of five steps:



(a) Backward tree construction. (b) Code length computation. (c) Codes assignment.

Figure 7: Masked state encoding.

1) Using XFA, we construct a forward tree that contains forward transitions, i.e., directed edges from a state of depth i to a state of depth $i+1$. Each forward transitions is essentially a basic transition of XFA.

2) Based on the forward tree, we construct a backward tree that consists of backward transitions. Each backward transition is a directed edge from a state of depth i to a state of depth $j<i$, representing a default transition between two states in XFA. The default transition is used to replace multiple transitions to the same destination state on the same input character. In particular, for each transition with a priority 1, we produce a backward transition from its destination state to the start state.

3) We compute a masked code length for each state in a bottom-up manner. The code length of state j indicates the number of its descendent states. The code length of each leaf state is set to 0 in the backward tree, and then the code length of each ancestor state is recursively computed as follows:

$$c_j = \left\lceil \log_2(1+\sum_i 2^{c_i}) \right\rceil$$

where c_j is the code length of state j, and c_i is the code length of j's child state i.

4) We assign both the transition code and the masked code to each state starting from the root in a top-down manner. Starting from the top of the backward tree, we propose an upper- and lower-bound assignment method to recursively assign codes to each state at each level. We first sort all the states at each level in descending order of the code length (assigned in Step 3). For each state, the masked code is then assigned from upper codes in the range covered by the code of its parent, and the transition code uses a lower code in the range covered by its masked code.

5) We combine multiple transitions into one TCAM entry by encoding their source states in ternary codes. For each state in the TCAM table, we use the masking code as its source state, and use the transition code as its desintation state. Thus, all the transtions to the same destination state on the same input character are combined into a single TCAM entry by using the masked code of a state to cover other source states.

TCAM		SRAM	
Source State	Input Character	Destination State	Instruction
101*	01100001	1111	Set(bit_2, 1)
111*	01100010	1001	Set(bit_1, 1)
100*	01100100	1101	Check(bit_2, 1)
110*	01100100	0111	NULL
0111	01100101	1011	Check(bit_1, 1)
****	01*00001	1110	NULL
****	01*00010	1000	NULL
****	01100011	1100	NULL
****	01100101	1010	NULL
****	********	0000	NULL

Figure 8: TCAM entry table after state compression.

Fig. 7 illustrates an example of masked state encoding. As seen in Fig. 7(a), we construct a forward tree from XFA in Fig. 1, where forward transitions are depicted by real lines from a state of depth i to a state of depth $i+1$, i.e., a transition from state 1 to 2 on input character 'b'. Using this forward tree, we then construct a backward tree, where backward transitions are depicted by dotted lines, i.e., a transition from state 2 to 8. For each transition with a wildcard source state and a priority 1 (see in Fig. 6), the corresponding backward transition is produced from its destination state (i.e., states 1, 8, 3, and 6) to the start state (i.e., state 0). Next, we use a bottom-up approach to recursively compute the code length of each state. As seen in Fig. 7(b), leaf states 7, 9, 5, 2, and 3 have the code length of 0, non-leaf states 1, 3, 6, and 8 have the code length of 1, and the start state 0 has the code length of 4. Using the upper- and lower-bound assignment method, we recursively assign both a masking code and a transition code to each state in a top-down approach as seen in Fig. 7(c). For instance, due to the code length of 1, the masked code of state 1 is assigned to 111* from upper codes in the range [0000, 1111] covered by state 0, and the transition code is assigned to 1110 by using a lower code in the range [1110, 1111] covered by its masked code 111*.

Fig. 8 illustrates the complete TCAM table after state compression. Before state compression, there are 14 entries in the TCAM table as seen in Fig. 6. We observe that there are multiple transitions to the same destination state on the same input character, e.g., the first two transitions from states 5 and 6 to 7 on input binary character "01100001". Using masked state encoding, we assign both a masked

Table 2: Results for optimized DFA and CFA on RegEx pattern sets.

RegEx Pattern Set	Snort1		Snort2		Bro2	
	Optimized DFA	CFA	Optimized DFA	CFA	Optimized DFA	CFA
# TCM Entries	59,712	9,718	81,352	50,256	3,700	925
# Bits Per TCAM Entry	25	20	25	21	21	17
TCAM Space (Mbits)	2.05	0.33	2.80	1.73	0.13	0.03
TCAM Throughput (Gbps)	2.72	3.63	2.42	2.72	5.45	10.90
TCAM Power (nJ)	33.62	5.88	45.61	28.38	2.54	1.01

Table 3: Results for XFA and CFA on RegEx pattern sets.

RegEx Pattern Set	Bro1		Bro3	
	XFA	CFA	XFA	CFA
# TCAM Entries	722,277	55,605	588,630	26,998
# Entries Per State	111	8.6	105	4.8
TCAM Space (Mbits)	24.80	1.91	20.21	0.93
TCAM Throughput (Gbps)	1.82	2.72	1.82	3.11
TCAM Power (nJ)	400.53	31.32	326.53	15.46
# Instructions Per TCAM Entry	0.35	0.70	0.24	0.43

code and a transition code to each state, and use them to merge the source states. For instance, states 5 and 6 are encoded in ternary source state 101*, and then two transitions from the source state are combined into a single transition from state 101* to state 1111 on character "01100001". The resulting TCAM table has now only 10 entries.

3.5 Multi-Stride CFAs

Multi-stride RegEx matching is widely used to improve inspection throughput. A k-stride XFA processes k characters per transition, achieving RegEx matching speed-up by up to k times. However, multi-stride XFA has a transition-space explosion problem. Each state in a k-stride XFA has up to 256k potential transitions, resulting in a prohibitive amount of TCAM space. In order to eliminate transition explosion, we generalize to multi-stride CFA by extending these above compression techniques.

Transition Compression: Similarly to a single stride, transition grouping divides all the transitions with the same input sub-string of characters into multiple groups. We then merge the largest group into a single transition with wildcard source state "*", and assign it a priority 1. The transitions from wildcard state to the start state are combined into a wildcard transition and assigned a priority 0. Finally, we reconstruct a smaller TCAM table by sorting all transitions in descending order of the priority.

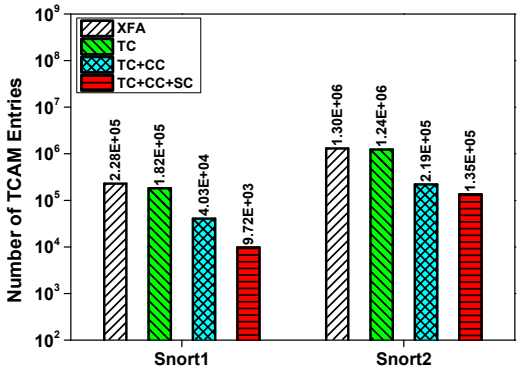
Character Compression: Character grouping and merging is also extended to encode multiple input sub-strings of characters in a single ternary sub-string. Character grouping is used to merge the most frequent transitions with the same source and destination state into a single transition with wildcard input character "*" and assign it a priority 1. Similarly, we use heuristic character merging to combine other transitions by iteratively merging the input sub-strings in ternary sub-strings.

State Compression: Masked state encoding is also used to encode multiple source states in a ternary code. First, we construct a backward tree from a multi-stride XFA, where a backward transition is used as a default transition to replace multiple transitions to a specific state. Second, we compute the code length of each state in a bottom-up manner, and assign it a transition code and masked code in a top-down manner. Finally, we combine multiple transitions with the same input sub-string and destination state in one TCAM entry by using a ternary masked code of their source states.

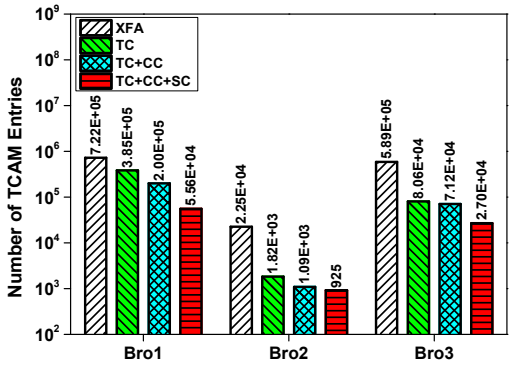
4. EXPERIMENTAL EVALUATION

In this section, we conduct experiments on realistic RegEx pattern sets to evaluate CFA in comparison with optimized DFA [4] and XFA. We have implemented XFA and CFA directly from the RegEx patterns, and developed their associated TCAM tables. To estimate the RegEx matching throughput, the power consumption, and the latency of a single TCAM lookup in TCAM-based solutions, we use the TCAM power and delay model [29] assuming that each TCAM chip is manufactured with a 0.18um process. For our test, we mainly measure three performance metrics: TCAM space, matching throughput, and TCAM power consumption.

We have obtained five RegEx pattern sets from Snort and Bro: Snort1, Snort2, Bro1, Bro2, and Bro3. As shown in



(a) Snort pattern sets.



(b) Bro pattern sets

Figure 9: Number of TCAM entries after different compression techniques

Table 1, Snort1, Bro2, and Bro3 contain a full set of RegEx patterns all with wildcard closures ".*", while Snort2 has 15 out of 40 patterns containing wildcard closures ".*", and Bro1 has 18 out of 217 such patterns. We can build DFAs from Snort1 (49,128 states), Snort2 (39,502 states), and Bro2 (3,644 states), but could not build DFAs from Bro1 and Bro3 due to state explosion.

4.1 Results on 1-Stride Finite Automata

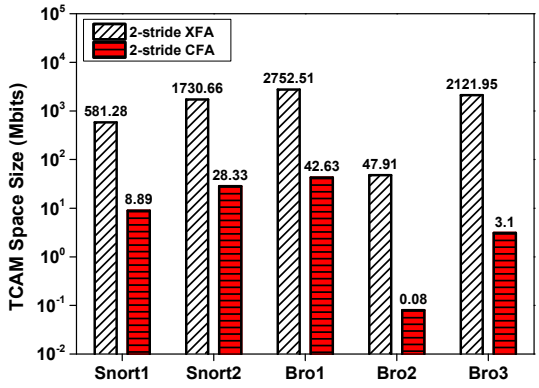
We conducted experiments to evaluate the performance of 1-stride finite automata using TCAMs. For Snort1, Snort2, and Bro2, DFAs can be built, and Table 2 shows the results for 1-stride optimized DFA and 1-stride CFA on these three sets. We see that CFA outperforms optimized DFA in terms of TCAM space, matching throughput, and TCAM power consumption. For TCAM space, CFA requires 83%, 38%, and 75% less the number of TCAM entries than optimized DFA on Snort1, Snort2, and Bro2, respectively. In addition, CFA also requires fewer bits per TCAM entry than optimized DFA. For instance, optimized DFA on Snort1 requires 25 bits per TCAM entry, while CFA requires only 20 bits per TCAM entry. However as the minimal width of TCAM entry is 36 bits, both CFA and optimized DFA have to choose TCAM width 36, and use the same space for each entry. For matching throughput, CFA achieves a TCAM throughput of up to 10.9Gbps using 0.03Mb TCAM space, and doubles the throughput of optimized DFA. The main component of TCAM power consumption is proportional to the number of searched entries. Thus, the TCAM power consumption of CFA is reduced by 83%, 38%, and 60% compared to optimized DFA respectively. Table 2 demonstrates that CFA achieves significant improvements in space, throughput, and power compared to optimized DFA.

For these RegEx patterns where DFAs cannot be built, one can construct XFAs and use CFAs to implement TCAM-based RegEx matching. Table 3 compares the performance of 1-stride XFA and CFA on Bro1 and Bro3. From the table, we see that CFA dramatically reduces the TCAM space requirements over XFA. In particular, CFA requires 92% and 95% less the number of TCAM entries on Bro1 and Bro3 respectively. In return, CFA requires two times more instructions per TCAM entry than XFA. As these simple instructions can be efficiently operated by TCAM logic circuits, the operation cost is small and even negligible. We also see that CFA increases matching throughput by 1.5-1.7 times. Therefore, Table 3 validates that CFA outperforms XFA in terms of TCAM space, matching throughput, and TCM power consumption.

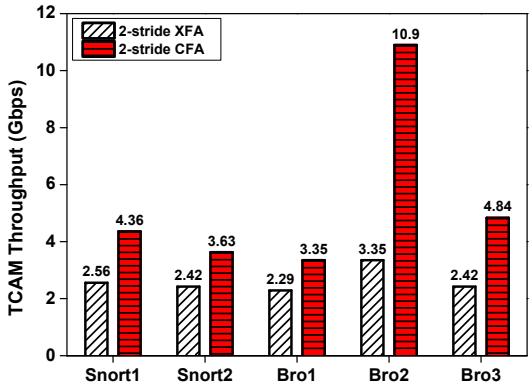
We also evaluate the effectiveness of each one of the three compression techniques proposed in this paper. Fig. 9 shows the number of TCAM entries after applying each compression technique. In the figure, we use TC, CC, and SC to indicate transition compression, character compression, and state compression. We see that each technique proposed in CFA is very effective in reducing the number of TCAM entries, and has an important effect against different RegEx pattern sets. For instance, as seen in Fig .9, using TC on Snort1 and Snort2 only reduces the number of TCAM entries by 20% and 12% respectively, while using TC+SC reduces the number of TCAM entries by 82% and 86%. In contrast, using TC on Bro2 and Bro3 reduces the number of TCAM entries by 92% and 86% respectively. Yet, using TC+SC further reduces the number of TCAM entries by 3% and 2% respectively. This can be explained by the fact that transition redundancy has different distributions in these RegEx pattern sets. Nevertheless, the application of the three techniques, i.e., TC+CC+SC, can achieve together a significant reduction of the number of TCAM entries by 91% and 96%.

4.2 Results on 2-Stride Finite Automata

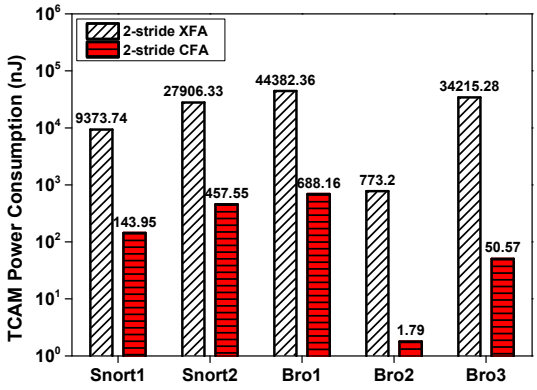
We also conducted experiments to compare 2-stride XFA to 2-stride CFA. Fig. 10 shows the results for 2-stride XFA and CFA. As seen in Fig. 10, CFA has even higher reduction performance on 2 strides than on 1 stride. This is



(a) TCAM space.



(b) Matching throughput.



(c) TCAM power consumption.

Figure 10: Results for 2-stride XFA and 2-stride CFA.

due to the fact that there are more redundancies in multi-stride XFA. For instance, 2-stride XFA on Bro3 requires 2121.95Mb, while 2-stride CFA requires only 3.1Mb. As a consequence, 2-stride CFA achieves significant improvements in matching throughput and power consumption. Fig. 10(b) shows that 2-stride CFA increases matching throughput by 1.5-3.3 times compared to XFA. As see in Fig. 10(c), 2-stride CFA decreases TCAM power consumption by 98-99%. Fig. 10 validates the fact that CFA dramatically outperforms XFA in case of multiple strides.

5. CONCLUSION

In this paper, we propose CFA, a space-efficient finite automaton implementation for scalable TCAM-based RegEx matching. CFA is designed by using three novel compression techniques: transition, character, and state compressions. CFA reduces the memory requirements of implementing an XFA in a small TCAM. CFA achieves significant reductions in TCAM space by leveraging the transitions, characters, and states redundancies for combining multiple transitions into a single TCAM entry. We also generalize these techniques to multi-stride CFA.

Experiments on realistic RegEx pattern sets show that CFA dramatically outperforms previous solutions in terms of space, throughput, and power. When a DFA can be built for a set of RegEx patterns, CFA reduces TCAM space and power consumption by up to 83% compared to optimized DFA, and achieves matching throughput of up to 10.9Gbps, twice that of optimized DFA. When DFA cannot be built, CFA reduces TCAM space as well as TCAM power consumption by up to 95%, and increases matching throughput by up to 1.7 times. Our results also show that 2-stride CFA achieves even higher reduction performance than 2-stride XFA.

6. ACKNOWLEDGMENTS

This work was supported in part by the National High-Tech R&D Program of China under Grant No.2013AA013501, the National Science and Technology Major Project of

China under Grant No.2012ZX03002016, the Instrument Developing Project of CAS under Grant No.YZ201229, the National Science Foundation of China under Grant No. 61100171, No.61173167, No.61061130562, No.61272546, and No.61370226, and the National Science Foundation under Grant Numbers CNS-1017598, CNS-1017588, CNS-0845513, CNS-0916044, and CCF-1347953.

7. REFERENCES

[1] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In Proc. ACM CCS, 2003.

[2] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In Proc. IEEE ICNP, 2004.

[3] M. Alicherry, M. Muthuprasanna, and V. Kumar. High speed pattern matching for network ids/ips. In Proc. IEEE ICNP, 2006.

[4] C. R. Meinters, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention. In Proc. USENIX Security, 2010.

[5] K. Peng, S. Tang, M. Chen, and Q. Dong. Chain-based dfa deflation for fast and scalable regular expression matching using tcam. In Proc. ACM/IEEE ANCS, 2011.

[6] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. Communications of the ACM, vol.18, no.6, pp.333-340, 1975.

[7] Snort. http://www.snort.org/.

[8] Bro. http: //www.bro-ids.org/.

[9] R. Smith, C. Estan, and S. Jha. XFA: faster signature matching with extended automata. In Proc. IEEE Symposium on Security and Privacy (S & P), 2008.

[10] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection. In Proc. ACM SIGCOMM, 2008.

[11] S. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In Proc. FCCM, 2001.

[12] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In Proc. FPL, 2003.

[13] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In Proc. FCCM, 2004.

[14] I. Sourdis and D. Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In Proc. FCCM, 2004.

[15] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In Proc. FCCM, 2003.

[16] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In Proc. IEEE INFOCOM, 2004.

[17] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In Proc. ISCA, 2005.

[18] J. van Lunteren. High-performance pattern-matching for intrusion detection. In Proc. IEEE INFOCOM, 2006.

[19] A. Bremler-Bar, D. Hay, and Y. Koral. CompactDFA: generic state machine compression for scalable pattern matching. In Proc. IEEE INFOCOM, 2010.

[20] B. Brodie, R. Cytron, and D. Taylor. A scalable architecture for high-throughput regular-expression pattern matching. In Proc. ISCA, 2006.

[21] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In Proc. ACM SIGCOMM, 2006.

[22] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In Proc. ACM/IEEE ANCS, 2006.

[23] M. Becchi and S. Cadami. Memory-efficient regular expression search using state merging. In Proc. IEEE INFOCOM, 2007.

[24] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In Proc. ACM CoNEXT, 2007.

[25] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In Proc. ACM/IEEE ANCS, 2008.

[26] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In Proc. ACM CoNEXT, 2008.

[27] M. Becchi, C. Wiseman, and P. Crowley. Evaluating regular expression matching engines on network and general purpose processors. In Proc. ACM/IEEE ANCS, 2009.

[28] J. van Lunteren and A. Guanella. Hardware-accelerated regular expression matching at multiple tens of gb/s. In Proc. IEEE INFOCOM, 2012.

[29] B. Agrawal and T. Sherwood. Ternary cam power and delay model: extensions and uses. IEEE Transactions on VLSI, vol.16, no.5, pp.554-564, 2008.