Reorganized and Compact DFA for Efficient Regular Expression Matching

Kai Wang[1,2], Yaxuan Qi[1,2], Yibo Xue[2,3], Jun Li[2,3]
[1]Department of Automation, Tsinghua University, Beijing, China
[2]Research Institute of Information Technology, Tsinghua University, Beijing, China
[3]Tsinghua National Lab for Information Science and Technology, Beijing, China
wang-kai09@mails.tsinghua.edu.cn, {yaxuan, yiboxue, junl}@tsinghua.edu.cn

***Abstract*—Regular expression matching has become a critical yet challenging technique in content-aware network processing, such as application identification and deep inspection. To meet the requirement of processing heavy network traffic at line rate, Deterministic Finite Automata (DFA) is widely used to accelerate regular expression matching at the expense of large memory usage.**

In this paper, we propose a DFA-based algorithm named RCDFA (Reorganized and Compact DFA), which dramatically reduces the memory usage while maintaining fast and deterministic lookup speed. Based on the dissection of real-life DFA tables, we observe that almost every state has multiple similar states, i.e. they share identical next-state transitions for most input characters. However, these similar states often distribute at nonadjacent positions in the original DFA table. RCDFA aims at reorganizing all similar states into adjacent entries, so that identical transitions become consecutive along the *state* dimension, then compresses the reorganized DFA table utilizing bitmap technique. Coupled with mapping along the *character* dimension, RCDFA is not only efficient in DFA compression, but also effective for hardware implementation.

Experiment results show that RCDFA has superior performance in terms of high processing speed, low memory usage and short preprocessing time. RCDFA consistently achieves over 95% compression ratio for existing real-life rule sets. Implemented in a single Xilinx Virtex-6 FPGA platform, RCDFA matching engine achieved 12Gbps throughput.

***Key Words*—Deterministic Finite Automata (DFA), Regular Expression Matching, Deep Inspection**

I. Introduction

Nowadays, regular expression has been widely used in signature-based content-aware network processing [1], such as application identification for charged traffic and deep inspection for malicious attacks. Application-level protocol signatures in Linux L7-filter [2], and attack signatures in Snort [3] and Bro [4] intrusion detection systems are principally described as regular expressions.

With the rapid increase in network bandwidth and applications, the number of signatures is growing dramatically in popular network security software tools [3, 4] and devices [5]. Therefore, the matching speed and the memory storage for multiple regular expressions are becoming the bottleneck of content-aware network processing.

To meet the requirement of processing heavy traffic at line rate in modern high-bandwidth networks, regular expression matching engine with predictable and acceptable memory bandwidth is increasingly essential. Because of the superior O(1) processing time complexity, Deterministic Finite Automata (DFA) [6] is widely used in existing work to accelerate regular expression matching, and a considerable amount of DFA-based solutions [7-14] have been proposed.

As a state-of-the-art work, Kumar et al. first observed the similarity of DFA states [10], i.e. almost every state can share the same next-state transitions with multiple other states for most input characters. They proposed the D^2FA algorithm that significantly eliminates the redundant transitions within similar states via default transitions. Becchi et al. further improved D^2FA by choosing default transitions "backwards" to their ancestor states, and achieved better average search speed and shorter preprocessing time [12].

Although D^2FA-related algorithms show very good performance in terms of DFA compression ratio, they suffer from nondeterministic memory access per input character. D^2FA can have predictable memory access by limiting the maximum default path length [11], but at the expense of inefficient and unstable compression performance. As well as D^2FA, many other solutions including the one proposed by Tuck et al. [15] are infeasible in practice due to the unacceptable memory usage.

In this paper, we propose an effective solution for efficient regular expression matching, named RCDFA (Reorganized and Compact DFA). Our contributions include:

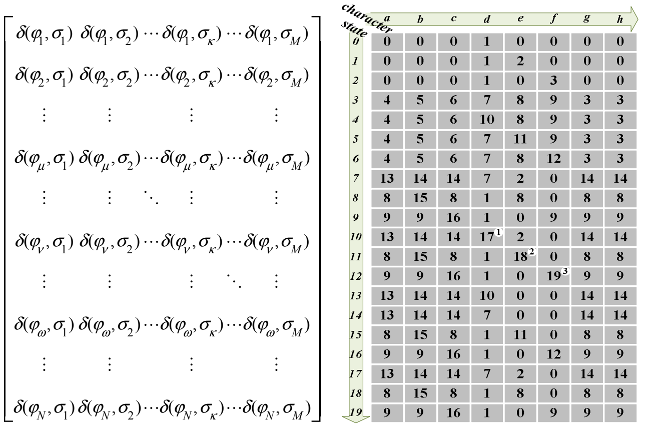
- *A high-performance and low-complexity compression algorithm:* RCDFA can exploit the state similarity by reorganizing the DFA table, and efficiently eliminate the redundancy in the DFA table with bitmap along the *state* dimension and mapping along the *character* dimension. Above 95% compression ratio can be stably achieved for real-life rule sets. Furthermore, RCDFA only needs $O(N)$ space complexity and $O(N\log N)$ time complexity for the preprocessing of an N-state DFA.
- *A high-throughput and low-latency hardware matching engine:* RCDFA can be effectively implemented in hardware for a deterministic number of memory accesses per input character. Based on low-latency pipelined architecture with interleaved input, and parallel SRAM access enabled by the single Xilinx Virtex-6 FPGA platform, RCDFA matching engine can achieve more than 12Gbps single chip throughput.

The rest of this paper is organized as follows. The motivation for our work is presented in section II. The detailed RCDFA algorithm and the matching engine architecture are illustrated in section III. The results of algorithm and hardware evaluation are provided in section IV. Section V is the conclusion.

II. Motivation

A. Problem Description

Assuming a DFA of N-state set Φ, is generated from multiple regular expressions over the alphabet Σ with M characters. It can be represented by a two-dimensional table Δ_{N*M} shown in Figure 1(a). Each element $\delta(\varphi_w, \sigma_\kappa)$ in Δ_{N*M} stands for the next-state transition of state φ_w corresponding to input character σ_κ. If current state and input character is φ_w and σ_κ respectively, and suppose $\delta(\varphi_w, \sigma_\kappa) = \varphi_\mu$, we can get to state φ_μ, consisting of M transitions $\{\delta(\varphi_\mu, \sigma_\kappa), \sigma_\kappa \in \Sigma\}$ in the μth row, with taking the transition $\delta(\varphi_w, \sigma_\kappa)$. Figure 1(b) further illustrates a real DFA table, where each number, i.e. transition, is equivalent to the element $\delta(\varphi_w, \sigma_\kappa)$ in Δ_{N*M}.



$$\begin{bmatrix}
\delta(\varphi_1,\sigma_1) & \delta(\varphi_1,\sigma_2) & \cdots\delta(\varphi_1,\sigma_\kappa) & \cdots\delta(\varphi_1,\sigma_M) \\
\delta(\varphi_2,\sigma_1) & \delta(\varphi_2,\sigma_2) & \cdots\delta(\varphi_2,\sigma_\kappa) & \cdots\delta(\varphi_2,\sigma_M) \\
\vdots & \vdots & \vdots & \vdots \\
\delta(\varphi_\mu,\sigma_1) & \delta(\varphi_\mu,\sigma_2) & \cdots\delta(\varphi_\mu,\sigma_\kappa) & \cdots\delta(\varphi_\mu,\sigma_M) \\
\vdots & \vdots\ \ddots & \vdots & \vdots \\
\delta(\varphi_v,\sigma_1) & \delta(\varphi_v,\sigma_2) & \cdots\delta(\varphi_v,\sigma_\kappa) & \cdots\delta(\varphi_v,\sigma_M) \\
\vdots & \vdots & \vdots\ \ddots & \vdots \\
\delta(\varphi_\omega,\sigma_1) & \delta(\varphi_\omega,\sigma_2) & \cdots\delta(\varphi_\omega,\sigma_\kappa) & \cdots\delta(\varphi_\omega,\sigma_M) \\
\vdots & \vdots & \vdots & \vdots \\
\delta(\varphi_N,\sigma_1) & \delta(\varphi_N,\sigma_2) & \cdots\delta(\varphi_N,\sigma_\kappa) & \cdots\delta(\varphi_N,\sigma_M)
\end{bmatrix}$$

state \ character	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
0	0	0	0	1	0	0	0	0
1	0	0	0	1	2	0	0	0
2	0	0	0	1	0	3	0	0
3	4	5	6	7	8	9	3	3
4	4	5	6	10	8	9	3	3
5	4	5	6	7	11	9	3	3
6	4	5	6	7	8	12	3	3
7	13	14	14	7	2	0	14	14
8	8	15	8	1	8	0	8	8
9	9	9	16	1	0	9	9	9
10	13	14	14	17^1	2	0	14	14
11	8	15	8	1	18^2	0	8	8
12	9	9	16	1	0	19^3	9	9
13	13	14	14	10	0	0	14	14
14	13	14	14	7	0	0	14	14
15	8	15	8	1	11	0	8	8
16	9	9	16	1	0	12	9	9
17	13	14	14	7	2	0	14	14
18	8	15	8	1	8	0	8	8
19	9	9	16	1	0	9	9	9

Figure 1: (a) Abstract DFA table Δ_{N*M} (b) Real DFA table recognizing regular expressions: ***def*[^*ef*]*add**, ***def*[^*df*]*bee** and ***def*[^*de*]*cff** over alphabet {***a***,***b***,***c***,***d***,***e***,***f***,***g***,***h***}. (state *17*, state *18* and state *19* are the matching states corresponding to previous 3 regular expressions respectively)

According to our observation on the DFA table over real-life rule sets, almost every state can share the same next-state transitions with multiple other states for most input characters, as illustrated in Figure 1(b). We aim at exploiting this state similarity to obtain a compact DFA structure with as few transitions as possible, while at the same time sustaining deterministic lookup per input character for effective hardware implementation.

B. Bitmap Compression

Bitmap compression technique, utilized in existing solutions to reduce the identical transitions inside each state [15], can meet the requirement of compression ability with guaranteed deterministic and acceptable memory access.

As Figure 2 shows, consecutively identical transitions inside state φ_w can be compressed by sharing one unique transition. For state lookup with input character σ_κ, we only need to access bitmap λ_ω once to count the number of 1's (defined as τ) in the first κ bits of the bitmap, and access transition once for taking the τth transition $\delta(\varphi_w, \sigma_\kappa)$ of the unique transitions, equivalent to taking the κth transition of the original transitions.

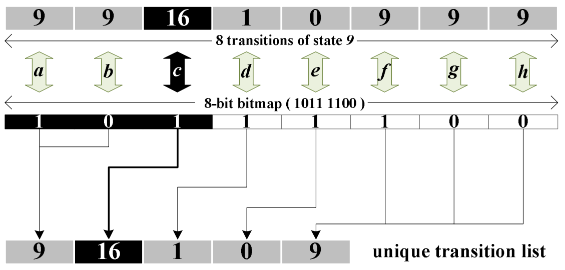


Figure 2: Bitmap compression technique

C. Our Proposal

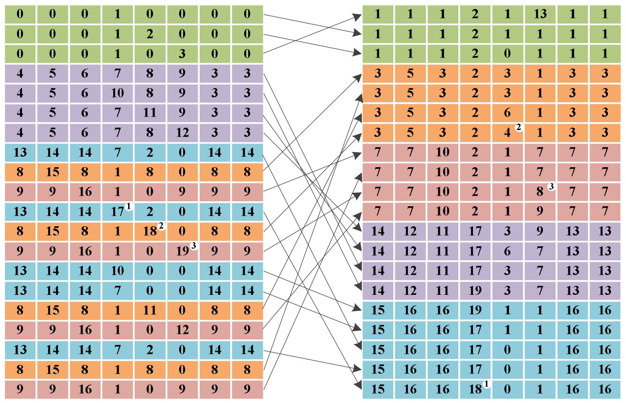
Due to the large redundancy of identical transitions within similar states in DFA, we can employ the bitmap technique to do compression along the *state* dimension, called *state bitmap*.

However, in the real-life DFA table, identical transitions are not consecutive along the *state* dimension in most cases, as described in Figure 1(b), while bitmap compression is only effective for consecutively identical transitions. Therefore, *state bitmap* cannot be applied directly. A more general solution is proposed in this paper, which we call RCDFA.

III. Algorithm and Architecture

A. DFA Reorganization

From Figure 3(a), which illustrates similar states in the same color, the rows identically colored have identical transitions for most input characters, but their distribution is scattered along the *state* dimension. To make as many identical transitions consecutive as possible, the most effective way is clustering these identically colored rows adjacent. Therefore, reordering the rows of the original DFA table is required.



0	0	0	1	0	0	0	0
0	0	0	1	2	0	0	0
0	0	0	1	0	3	0	0
4	5	6	7	8	9	3	3
4	5	6	10	8	9	3	3
4	5	6	7	11	9	3	3
4	5	6	7	8	12	3	3
13	14	14	7	2	0	14	14
8	15	8	1	8	0	8	8
9	9	16	1	0	9	9	9
13	14	14	17^1	2	0	14	14
8	15	8	1	18^2	0	8	8
9	9	16	1	0	19^3	9	9
13	14	14	10	0	0	14	14
13	14	14	7	0	0	14	14
8	15	8	1	11	0	8	8
9	9	16	1	0	12	9	9
13	14	14	7	2	0	14	14
8	15	8	1	8	0	8	8
9	9	16	1	0	9	9	9

1	1	1	2	1	13	1	1
1	1	1	2	1	1	1	1
1	1	1	2	0	1	1	1
3	5	3	2	3	1	3	3
3	5	3	2	3	1	3	3
3	5	3	2	6	1	3	3
3	5	3	2	4^2	1	3	3
7	7	10	2	1	7	7	7
7	7	10	2	1	7	7	7
7	7	10	2	1	8^3	7	7
7	7	10	2	1	9	7	7
14	12	11	17	3	9	13	13
14	12	11	17	6	7	13	13
14	12	11	17	3	7	13	13
14	12	11	19	3	7	13	13
15	16	16	19	1	1	16	16
15	16	16	17	1	1	16	16
15	16	16	17	0	1	16	16
15	16	16	17	0	1	16	16
15	16	16	18^1	0	1	16	16

Figure 3: (a) Original DFA table with state *0* as the initial state (b) Reorganized DFA table with state *1* as the initial state

However, in the original DFA table Δ_{N*M}, the value of each transition $\delta(\varphi_w, \sigma_\kappa)$ is the next-state index, thus $\delta(\varphi_w, \sigma_\kappa)$ must be modified accordingly when the position of next state is changed. For original $\delta(\varphi_w, \sigma_\kappa) = \varphi_\mu$, while state φ_μ is swapped with state φ_v, then we should make $\delta(\varphi_w, \sigma_\kappa) = \varphi_v$. In terms of matrix, it is equivalent to a series of elementary row transformation, demonstrated in Figure 4.

Therefore, for DFA reorganization, it is not only necessary to adjust the positions of states according to their similarity, but also to renumber the transitions with new positions of corresponding next states. If original position μ of states φ_μ is changed into new position v after reordering, all transitions $\{\delta(\varphi_w, \sigma_\kappa) \mid \delta(\varphi_w, \sigma_\kappa) = \varphi_\mu, \varphi_w \in \Phi, \sigma_\kappa \in \Sigma\}$ should be modified by $\{\delta(\varphi_w, \sigma_\kappa) \mid \delta(\varphi_w, \sigma_\kappa) = \varphi_v, \varphi_w \in \Phi, \sigma_\kappa \in \Sigma\}$. Figure 3(b) describes the effect of DFA reorganization, where we can find identically colored rows are clustered in the reorganized DFA table Δ'_{N*M}, and the value of each transition has been exchanged.

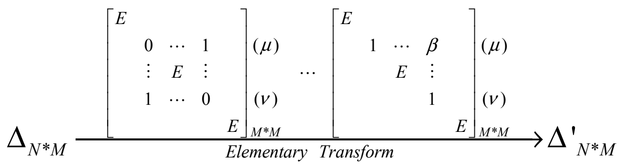


Figure 4: Elementary transformation of DFA table Δ_{N*M}

Figure 3(b) gives the optimal result of DFA reorganization, which is the upper limit of all heuristic algorithms. To help understand the detailed procedure, the pseudo code of one feasible DFA reorganization algorithm is presented below.

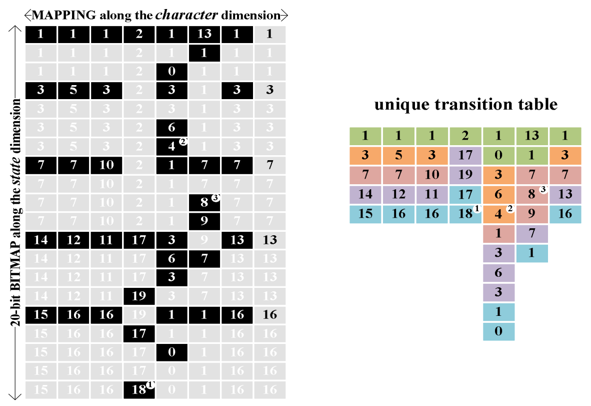
```
PROCEDURE dfa_reorganization( DFA dfa = Δ(Φ, Σ) )
    for (state s > 0 && s • Φ)
        int commax = 0;
        state ds = s;
        for (state t > s && t • Φ)
            int common = # of same transitions between (s-1) and t;
            if (common > commax)
                commax = common;
                ds = t;
            fi
        rof
        if (ds != s)
            for (char σ • Σ)
                swap transitions δ(s, σ) and δ(ds, σ);
                for (state φ • Φ)
                    exchange transitions satisfying δ(φ, σ) == s or δ(φ, σ)
== ds into δ(φ, σ) = ds or δ(φ, σ) = s respectively;
                rof
            rof
        fi
    rof
END
```

B. Compression with State Bitmap and Character Mapping

With DFA reorganized, most identical transitions are clustered adjacent along with the similar states, thus we can utilize *state bitmap* to compress the consecutively identical transitions along the *state* dimension efficiently. Because there are N transitions along the *state* dimension in reorganized DFA table, N-bit bitmap is introduced for *state bitmap*. Besides, we find that for some common input characters $\{\sigma_\mu, \sigma_v \mid \delta(\varphi_w, \sigma_\mu) = \delta(\varphi_w, \sigma_v),\ \varphi_w \in \Phi, \sigma_\mu, \sigma_v \in \Sigma\}$, the original transitions along the *state* dimension are exactly the same, so the unique transition table can be compressed further by mapping σ_μ and σ_v to the same character index, called *character mapping*.

Figure 5 describes the effect of *state bitmap* and *character mapping*, and gives the compact table with unique transitions. We can observe that most bitmaps $\{\lambda_\mu, \lambda_v \mid \lambda_\mu = \lambda_v, \sigma_\mu, \sigma_v \in \Sigma\}$ over alphabet Σ are the same, such as bitmaps for *a*, *b*, *c*, *g* and *h* in Figure 5. Therefore, less than M unique bitmaps are necessary to be reserved in the unique bitmap table for sharing.

In practice, the number of bitmaps can be further reduced by *bitmap combination*. Multiple bitmaps can be combined into one bitmap by "OR" each bit of them. Assuming bitmap λ_μ and λ_v is only different in the ω^{th} bit and have 0 and 1 respectively, if we use combined bitmap $\lambda_\kappa = (\lambda_\mu \mid \lambda_v)$ to replace λ_μ and λ_v, the ω^{th} transition of the original transitions corresponding to input character σ_μ will become a unique transition, and should be added in the ultimate unique transition table. Therefore, the cost of *bitmap combination* lies in raising extra transitions, which is not much for the 1's are very sparse in each bitmap.



MAPPING along the character dimension
20-bit BITMAP along the state dimension
unique transition table

Figure 5: Effect of *state bitmap* and *character mapping*

C. Hardware Matching Engine Design

Based on RCDFA algorithm, matching engine in hardware architecture can be designed as shown in Figure 6. The blue part stands for the functional logic, the green part with solid frame stands for the on-chip SRAMs for storing index mapping table <*bitmapID*, *base*>, bitmap table <*offset*, *bitmap*> and transition table <*state_out*>, the green part with dash frame stands for the demonstrated operation of state lookup.

For the convenience to index, we can store all transitions linearly, with recording *base* to locate each unique transition list corresponding to each input character $\sigma_\kappa \in \Sigma$. The *base* for characters under *character mapping* is identical, in order to locate the shared unique transition list.

In addition, because the bitmap size is determined by the number of states N, and τ counting of various-bit bitmap is inefficient for hardware implementation, we prefer to divide each N-bit bitmap into $\lceil N/k \rceil$ k-bit sub-bitmaps, with recording the number of 1's in all previous (κ-1) sub-bitmaps as the *offset* for the κ^{th} sub-bitmap, and the last sub-bitmap should keep k bits with padding 0.

With the sum of *base*, *offset* and sub-bitmap counting τ, we can locate the destination transition to get to the next state. In matching engine as Figure 6 shows, according to current state *state_in* and current input character *char_in*, we can find the next state *state_out* by calculating the required *ID*s in functional logic, and taking sequential and deterministic SRAM access for fetching index, bitmap and transition respectively.

For RCDFA matching engine with U unique transitions, L unique bitmaps and k-bit sub-bitmap size, the complexity of

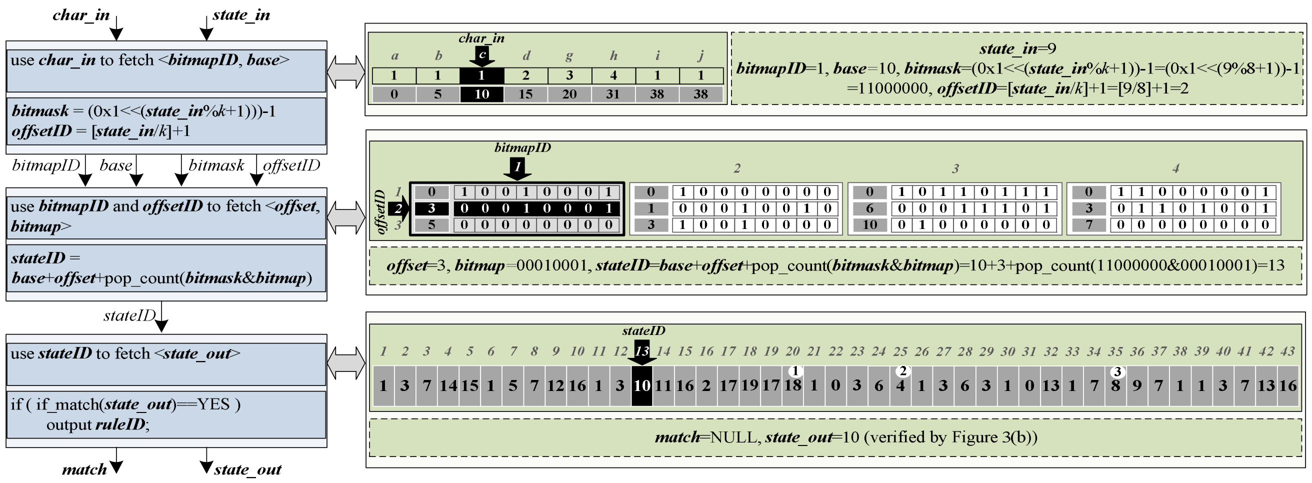

char_in
state_in
use char_in to fetch <bitmapID, base>
bitmask = (0x1<<(state_in%k+1)))-1
offsetID = [state_in/k]+1
bitmapID
base
bitmask
offsetID
use bitmapID and offsetID to fetch <offset, bitmap>
stateID = base+offset+pop_count(bitmask&bitmap)
stateID
use stateID to fetch <state_out>
if (if_match(state_out)==YES)
output ruleID;
match
state_out
state_in=9
bitmapID=1, base=10, bitmask=(0x1<<(state_in%k+1))-1=(0x1<<(9%8+1))-1
=11000000, offsetID=[state_in/k]+1=[9/8]+1=2
offsetID
offset=3, bitmap=00010001, stateID=base+offset+pop_count(bitmask&bitmap)=10+3+pop_count(11000000&00010001)=13
match=NULL, state_out=10 (verified by Figure 3(b))

Figure 6: Hardware matching engine based on RCDFA algorithm (the storage and the access of match list for finding *match* are kept the same as original DFA)

memory usage is O($NL\log_2 U/k$), and the throughput can be effectively raised with decreasing the bitmap size k. Besides, making each SRAM accessed in parallel with interleaving input characters, we can enhance the throughput with low latency of 3-stage pipeline.

IV. Performance Evaluation

A. Data Sets and Test Platform

The real-life rule sets used for our test include regular expression rules *snort24.re* (24 rules from Snort), *snort40.re*, *bro217.re* (217 rules from Bro), *linux13.re* (13 rules from Linux L7-filter), *linux30.re* and string matching rules *snortPart.str* (1,982 short signatures from Snort), *snortAll.str* (all 5,766 signatures from Snort). The network traffic used for our test is the 925MB trace publicly available at MIT Lincoln Lab [16].

Algorithm evaluation test is based on a 2.0 GHz dual-core Intel XEON L5335 server with Ubuntu 9.04 operation system. Hardware performance test is based on a single Xilinx Virtex-6 (XC6VSX475T) FPGA platform [17].

B. Algorithm Evaluation

The algorithms used for performance comparison include bitmap [15] along the *character* dimension, D^{2}FA [10] with limiting maximum default path length to 1, and improved D^{2}FA [12], which are denoted as Bitmap, D^{2}FA and D^{2}FA-improved respectively in the following test result.

1) Memory Access

As TABLE 1 shows, compared to D^{2}FA and D^{2}FA-improved, RCDFA has better 1 transition access, as well as state access, per input character. D^{2}FA may take once more default state access when no labeled transition in labeled state is matched, and D^{2}FA-improved has finite average state access but nondeterministic 6~13 for certain time. Besides, for once state lookup in D^{2}FA and D^{2}FA-improved, around 100 labeled transitions have to be fetched and compared for default state, but 1~10 for labeled state. Hence D^{2}FA and D^{2}FA-improved are both infeasible to be implemented in hardware, while RCDFA is effective for predictable and acceptable memory bandwidth.

TABLE 1
NUMBER OF STATE ACCESSED PER INPUT CHARACTER

Rule Sets	# of states	D^{2}FA		D^{2}FA-improved		RCDFA
		Avg.	Worst	Avg.	Worst	
Snort24.re	8335	1.09	2	1.98	9	1
Snort40.re	19019	1.94	2	1.98	7	1
Bro217.re	6533	1.18	2	1.43	9	1
Linux13.re	4871	1.03	2	1.73	6	1
Linux30.re	43547	1.04	2	1.99	13	1
SnortPart.str	5662	1.39	2	1.74	7	1
SnortAll.str	56280	1.44	2	1.78	10	1

2) Compression Rate

From TABLE 2 we can find that RCDFA consistently achieves 95~99% transition compression ratio. In comparison, Bitmap and D^{2}FA have unstable 20~95% and 50~90% compression ratios respectively. D^{2}FA-improved cannot work efficiently when the similarity of most states is intermediate, while D^{2}FA even cannot work when space reduction graph [10] contains too many edges to process.

TABLE 2
COMPRESSION RATE (% TRANSITION REDUCTION)

Rule Sets	# of transitions	Bitmap	D^{2}FA	D^{2}FA-improved	RCDFA
Snort24.re	2133760	88.87	91.04	98.73	99.01
Snort40.re	4868864	92.10	82.68	98.13	98.89
Bro217.re	1672448	70.91	76.09	98.94	98.57
Linux13.re	1246976	96.17	57.12	27.86	96.56
Linux30.re	11148032	90.84	80.94	34.88	96.79
SnortPart.str	1449472	31.88	83.88	99.21	98.69
SnortAll.str	14407680	17.86	84.11	99.22	98.90

3) Memory Usage

The memory usage of RCDFA is principally determined by the number of transitions and bitmaps, so *bitmap combination* is necessary. TABLE 3 shows that the transition compression ratio is slightly affected by *bitmap combination*, while the number of bitmaps is reduced dramatically. The overall space compression performance is improved and kept beyond 95% in practice.

TABLE 3
EFFECT OF BITMAP COMBINATION

Rule Sets	# of bitmaps before comb.	# of bitmaps after comb.	Compression rate change (%)
Snort24.re	62	32	↓ 0.39
Snort40.re	78	32	↓ 0.59
Bro217.re	107	32	↓ 1.48
Linux13.re	81	32	↓ 0.24
Linux30.re	102	32	↓ 0.75
SnortPart.str	181	32	↓ 2.98
SnortAll.str	256	32	↓ 3.34

The memory usage of RCDFA in hardware (see Figure 6) is tested with limiting the bitmap size to 256 bit and the number of bitmaps to 32, experiment result is shown in Figure 7.

Compared to D^2FA, RCDFA has about an order of magnitude space performance increase. And RCDFA is much better than D^2FA-improved for selected Linux L7-filter rules. Memory usage less than 1MB shows that RCDFA algorithm is suitable to be applied in on-chip SRAMs.

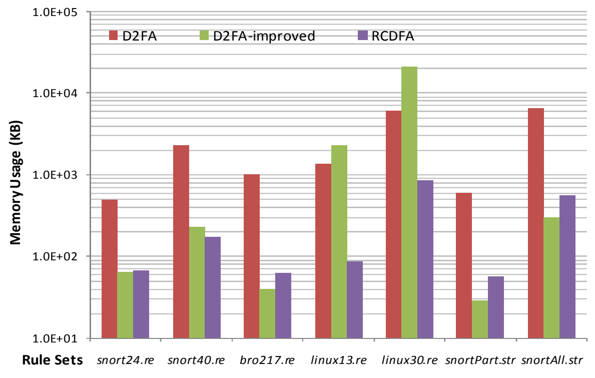


Figure 7: Comparison of memory usage

4) *Preprocessing Time*

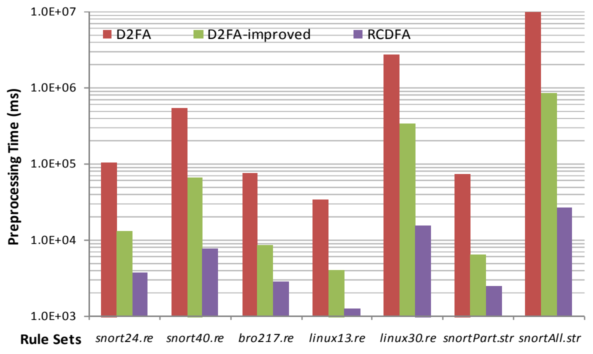


Figure 8: Comparison of preprocessing time

The time complexity of RCDFA for DFA compression (DFA construction not included) is $O(N\log N)$, which is more efficient than $O(N^2\log N)$ of D^2FA and $O(N^2)$ of D^2FA-improved. Figure 8 verifies RCDFA has the shortest preprocessing time.

C. *Hardware Performance*

Timing analysis in Xilinx ISE development tool shows, the maximum clock frequency depends on the bitmap size operated by *pop_count* instruction. We can achieve a maximum clock frequency of 156.01MHz and 2.4Gbps throughput of single RCDFA matching engine for the 256-bit bitmap. With parallelization of multiple RCDFA matching engines in the test FPGA with 5MB block RAMs, we can further achieve at least 2.4*5 = 12Gbps throughput for the real-life rule sets in test.

V. CONCLUSION

A DFA-based algorithm and hardware architecture for efficient regular expression matching, called RCDFA, is presented in this paper. Based on the observation that similar states are widely existing and often nonadjacent in a DFA table, RCDFA reorganizes the DFA table to make similar states adjacent entries, so that most identical transitions become consecutive along the *state* dimension. RCDFA then efficiently compresses the reorganized DFA with *state bitmap* and *character mapping*, and achieves over 95% compression ratio for real-life rule sets. Compared with existing algorithms, RCDFA has superior performance in terms of higher processing speed, less memory usage and shorter preprocessing time. With prototype implemented in a single Xilinx Virtex-6 FPGA platform, RCDFA achieved over 12Gbps throughput.

REFERENCES

[1] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in Proc. of ACM CCS, 2003.
[2] Application Layer Packet Classifier for Linux, http://l7-filter.sourceforge.net/
[3] Snort, http://www.snort.org/
[4] Bro Intrusion Detection System, http://www.bro-ids.org/
[5] Cisco Family of Security Appliance, http://www.cisco.com/
[6] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison Wesley, 1979.
[7] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in Proc. of FCCM, 2001.
[8] B. C. Brodie, R. K. Cytron and D. E. Taylor, "A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching," in Proc. of ISCA, 2006.
[9] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman and R. H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection," in Proc. of ACM/IEEE ANCS, 2006.
[10] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in Proc. of ACM SIGCOMM, 2006.
[11] S. Kumar, J. Turner and J. Williams, "Advanced Algorithms for Fast and Scalable Deep Packet Inspection," in Proc. of ACM/IEEE ANCS, 2006.
[12] M. Becchi and P. Crowley, "An Improved Algorithm to Accelerate Regular Expression Evaluation," in Proc. of ACM/IEEE ANCS, 2007.
[13] S. Kumar, B. Chandrasekaran, J. Turner and G. Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia," in Proc. of ACM/IEEE ANCS, 2007.
[14] R. Smith, C. Estan and S. Jha, "XFA: Faster Signature Matching with Extended Automata," in Proc. of IEEE SSP, 2008.
[15] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection," in Proc. of IEEE INFOCOM, 2004.
[16] DARPA Intrusion Detection Data Sets, http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html
[17] Vertex-6 FPGA Family, http://www.xilinx.com/products/virtex6/