A Multi-Dimensional Progressive Perfect Hashing for High-Speed String Matching

Author : Yang Xu, Lei Ma, Zhaobo Liu, H. Jonathan Chao

Publisher : ANCS 2011

Presenter : Kyle Wang

Date : 2012/2/28

Outline

- Main idea
- Problem Statement
- P^2-Hashing Algorithm
- 2D P^2-Hashing Algorithm
- System Design
- Perfect Hash Table Construction Algorithm With Rule Table Support
- Performance Evaluation

Main Idea

- Our main objective in this paper is to store all the transitions of AC automaton in a perfect hash table.
- P^2-Hashing first divides all the transitions into many small sets based on the two-dimensional values of the hash keys, and then places the sets of transitions progressively into the hash table until all are placed.
- When placing a transition in a hash table and causing a collision, we can change the value of a dimension of the hash key to rehash the transition to a new location of the hash table.
- Hash collisions that occurred during the insertion of a transition will only affect the transitions **in the same set**.

Problem Statement

An AC automaton is formally defined as a 5-tuple $= (Q, \sum, g, f, T)$, which consists of

- A finite set of states, Q, where each state is represented by a number ranging from 0 to $|Q| - 1$, among which 0 is the start (root) state;
- A finite input character set, $\sum$, called alphabet;
- A set of accepting states, $T \subseteq Q$;
- A goto transition function that, $g: Q \times \sum \rightarrow Q \cup \{fail\}$;
- A failure function that, $f: Q - \{0\} \rightarrow Q$.

A hash table is a 3-tuple $H = \{K, h, S\}$, consisting of

- A set of keys, K, where each key is used as the input of the hash function to obtain the index of the hash table;
- A table S, which has at least $|K|$ entries, i.e., $|S| \geq |K|$;
- A hash function that, $h: K \rightarrow N$, where N is the set of natural numbers from 0 to $|S| - 1$; the hash function is called a perfect hash function if for $\forall a, b \in K$ and $a \neq b$, we have $h(a) \neq h(b)$. In this paper, we call $h(a)$ the hash index of key a.

- Format of hash table :

(source state ID, input character) => destined state ID

Progressive Perfect Hashing Algorithm(P^2-Hashing)

- (1) If a hash collision occurs when we try to place a new key into the hash table, the collision might be avoided if we could change the value of the key.
- (2) The ID of each state of AC automaton could be named as any values, as long as there are no two states being named with the value.
- Based on these two observations, we designed a scheme called **Progressive Perfect Hash** (P^2-Hashing) to place the transitions of AC automaton in a hash table without collision.

Progressive Perfect Hashing Algorithm(P^2-Hashing)

- The main idea of P^2-Hashing is to divide the goto transitions of a given AC automaton into multiple independent sets according to their source states, and place these transition sets in the hash table in decreasing order of their sizes.

- Any hash collision occurring during the placement of a set causes the set placement failure, and the already-placed transitions in this set are removed from the hash table. Then **the source state** shared by transitions in this set is renamed, and another set placement trial is performed.

- The renaming operation repeats until a successful set placement is achieved, and then the placement of the next transition set starts.

Progressive Perfect Hashing Algorithm(P^2-Hashing)

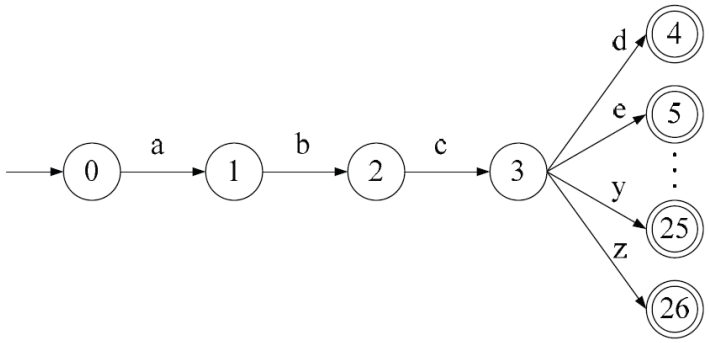

Figure 3. AC automaton for a set of rules with the same prefix. For simplicity, failure transitions are not shown (here all to the root state).

- It is easily seen that the success probability of this set placement is $\prod_{i=0}^{22} \frac{28-i}{28} \approx 1.3 \times 10^{-6}$. That means, on average, we have to rename state "3" by 10^6 times to achieve a successful set placement, and use 20 bits to name each state. Please note that ideally, 27 states of the AC automaton only require 5 bits for unique representation.

2D P²-Hashing Algorithm

Step 1

An AC automaton is formally defined as a 5-tuple $= (Q, \sum, g, f, T)$, which consists of

- A finite set of states, Q, where each state is represented by a number ranging from 0 to $|Q| - 1$, among which 0 is the start (root) state;
- A finite input character set, $\sum$, called alphabet;
- A set of accepting states, $T \subseteq Q$;
- A goto transition function that, $g: Q \times \sum \rightarrow Q \cup \{fail\}$;
- A failure function that, $f: Q - \{0\} \rightarrow Q$.

(1) In the first step, we model the AC automaton $M = (Q, \sum, g, f, T)$ as a bipartite graph, which is formally defined as a 3-tuple $B = (U, V, E)$, consisting of

- A set of nodes, U;
- A set of nodes, V;
- A set of edges, E; $\forall < u, v > \in E$ satisfies that $u \in U, v \in V$.

- Each state in AC automaton corresponds to a node in set U.
- Each character in AC automaton corresponds to a node in set V.
- Each transition in AC automaton corresponds to an edge in set E.

- Storing transitions of the AC automaton in a perfect hash table is equivalent to storing edges of the bipartite graph in the perfect hash table.

2D P²-Hashing Algorithm

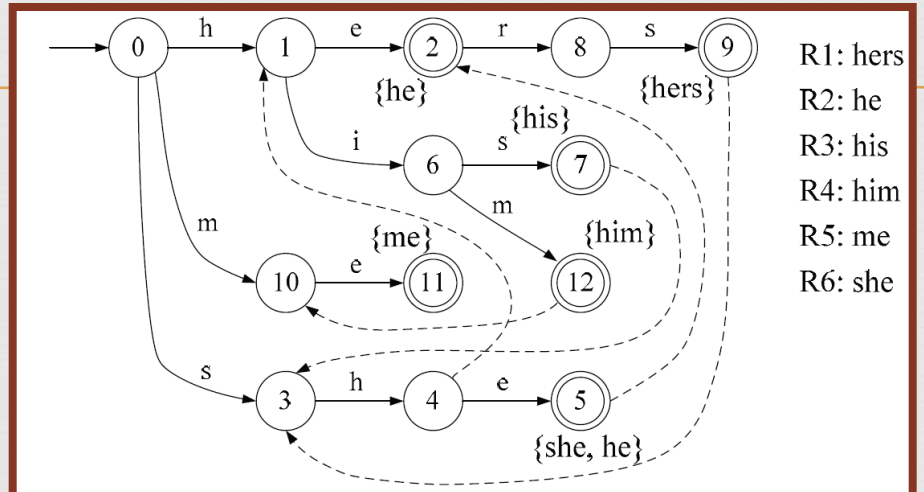

Figure 2. AC automaton for rule set {hers, he, his, him, me, she}. For simplicity, failure transitions to the root state are not shown.

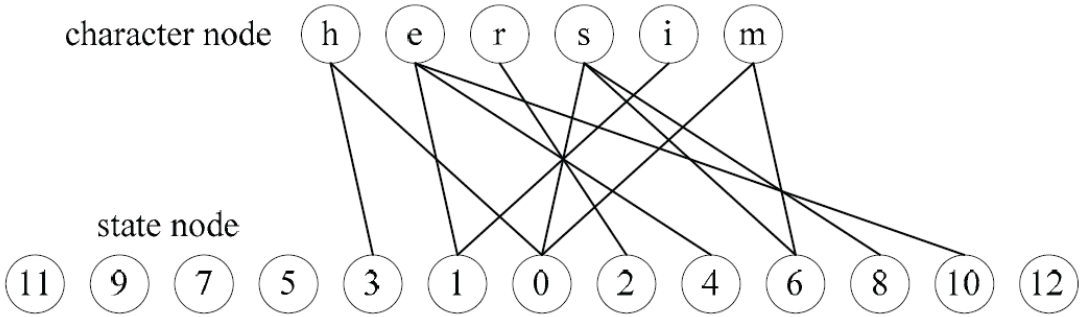

Figure 4. The bipartite graph model of the AC automaton in Figure 2.

2D P²-Hashing Algorithm

Step 2

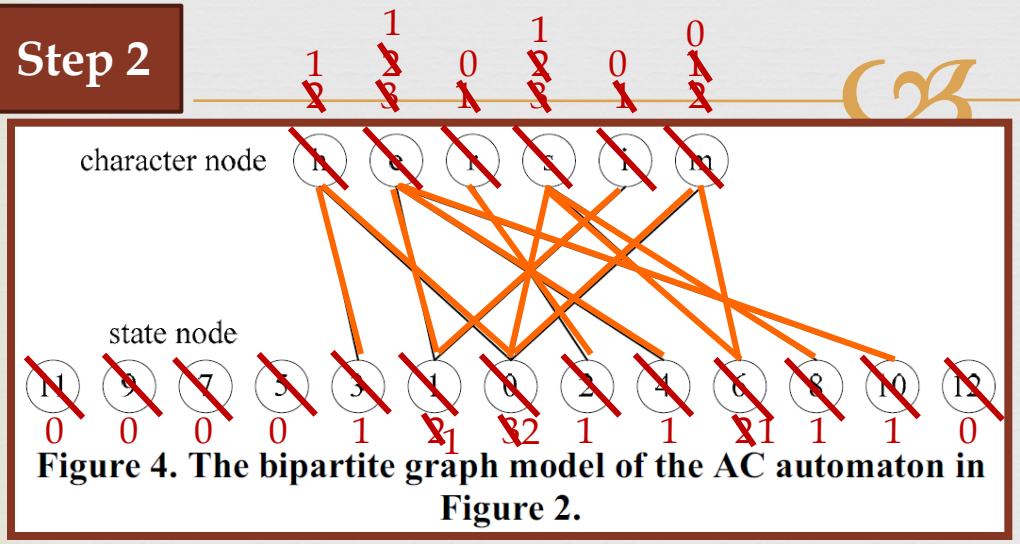

Figure 4. The bipartite graph model of the AC automaton in Figure 2.

TABLE II. Dependent Edge Sets of Nodes After The Bipartite Graph Decomposition (based on the bipartite graph in Figure 4)

Seq	1	2	3	4	5	6	7	8	9	10
Node	11	9	7	5	12	3	h	2	r	4
Dependent edge set						<3,h>	<0,h>	<2,r>		<4,e>
Seq	11	12	13	14	15	16	17	18	19	
Node	10	e	1	i	8	0	s	6	m	
Dependent edge set	<10,e>	<1,e>	<1,i>		<8,s>	<0,s> <0,m>	<6,s>	<6,m>		

Algorithm 1. Bipartite Graph Decomposition

Input:

Bipartite graph $B = (U, V, E)$;

Output:

A sequence number $N(v)$ for every node $v \in U \cup V$;
A dependent edge set $D(v)$ for every node $v \in U \cup V$;

Algorithm:

$N(v) := NULL\ (\forall\, v \in U \cup V)$;
$D(v) := NULL\ (\forall\, v \in U \cup V)$;
for $(j := 1; j \le |U| + |V|; j++)$;
{
Among all nodes in bipartite graph B, choose a node, say v, that has the least connected edges; if there are multiple qualified nodes, randomly select one;
$N(v) := j$;
$D(v)$:=the set of edges connected to node v;
Remove node v and its connected edges from the bipartite graph B;
}

2D P²-Hashing Algorithm

Step 3

TABLE II. Dependent Edge Sets of Nodes After The Bipartite Graph Decomposition (based on the bipartite graph in Figure 4)

Seq	1	2	3	4	5	6	7	8	9	10
Node	11	9	7	5	12	3	h	2	r	4
Dependent edge set						<3,h>	<0,h>	<2,r>		<4,e>
Seq	11	12	13	14	15	16	17	18	19	
Node	10	e	1	i	8	0	s	6	m	
Dependent edge set	<10,e>	<1,e>	<1,i>		<8,s>	<0,s> <0,m>	<6,s>	<6,m>		

Algorithm 2. Perfect Hash Table Construction

Input:

A sequence number $N(v)$ for every node $v \in U \cup V$;
A dependent edge set $D(v)$ for every node $v \in U \cup V$;
Name space NS_{state} and $NS_{character}$ *// contain available IDs for state nodes and character nodes, respectively.*

Output:

A perfect hash table H;
A Character Translation Table CTT, indexed by the ASCII codes of characters;

Algorithm:

Set H, CTT, and STT empty;
Sort nodes in $U \cup V$ in decreasing order of their sequence numbers;
for every node u in the sorted set $U \cup V$ **do**
//Without loss of generality, suppose u is a state node (the following code should be changed accordingly if u is a character node);
{
(1) Among all available IDs in NS_{state}, randomly choose an ID, say $id1$, which hasn't been tried by node u; if all IDs in NS_{state} have already been tried by node u, an error is returned;
Name node u as $id1$ and place all edges of $D(u)$ into hash table H; *//for every edge* $< u, v >$ *in* $D(u)$, *it's guaranteed that v has already been named;*
if no hash collision occurs during the placement of $D(u)$
remove $id1$ from NS_{state};
else
goto (1);
}

System Design

It is used to translate input characters from ASCII codes to the internal encodings.

It is used to store goto transitions and failure transitions and implemented as a perfect hash table.

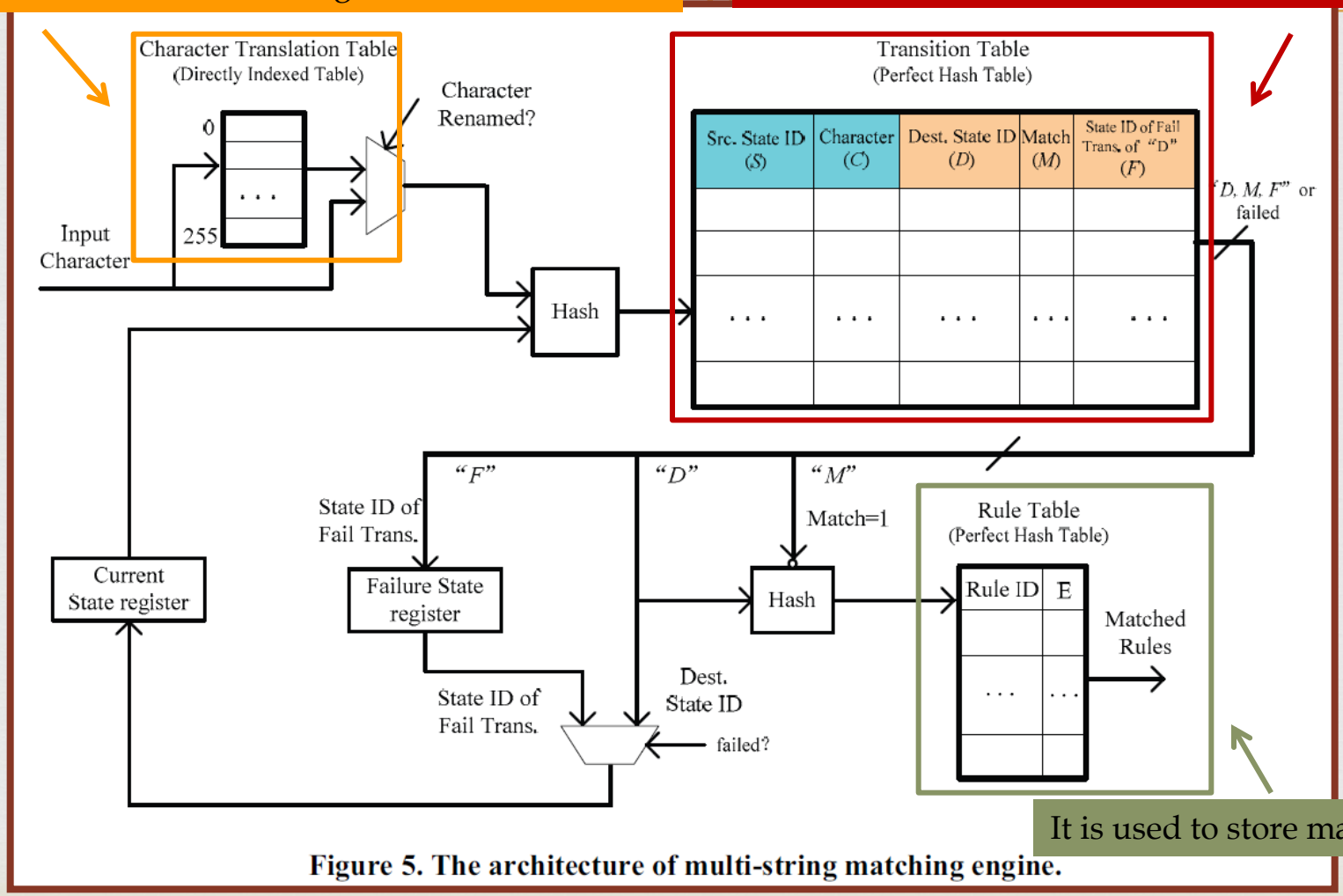

Figure 5. The architecture of multi-string matching engine.

It is used to store matching rules.

Perfect Hash Table Construction Algorithm With Rule Table Support

- To generalize the perfect hash table construction problem, we suppose that each rule R_i corresponds to a virtual character r_i.
- We now modify the 2D P²-Hashing algorithm to support the constructions of the two perfect hash tables (TT and RT).

Step 1

- AC automaton M = $(Q, \sum, g, f, T)$ → bipartite graph B = (U, V, E)
- Let $U = Q$

  $V = \sum \cup \{r_1, \ldots r_l\}$, where l is the number of rules.

  $E = E_1 \cup E_2$

  $E_1 = \{\langle q, c\rangle | \forall q \in Q, \forall c \in \sum, that\ g(q,c) \neq fail\}$

  $E_2 = \{\langle q, r_i\rangle | if\ state\ q\ matchs\ rule\ i\}$

Perfect Hash Table Construction Algorithm With Rule Table Support

Step 2

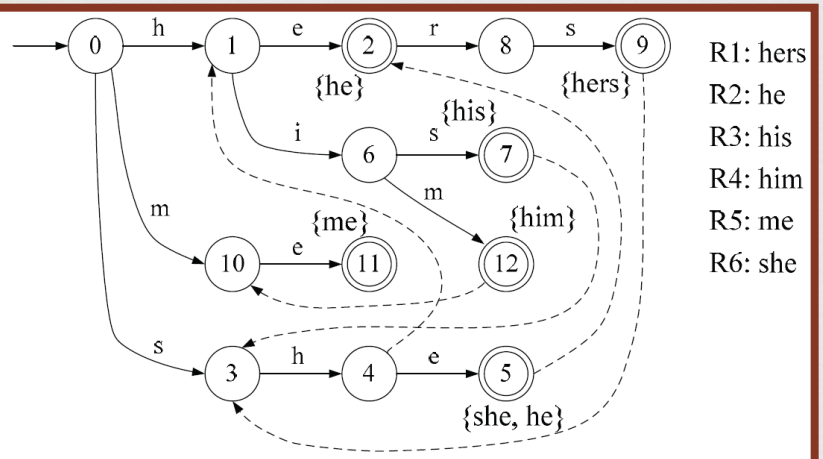

Figure 2. AC automaton for rule set {hers, he, his, him, me, she}. For simplicity, failure transitions to the root state are not shown.

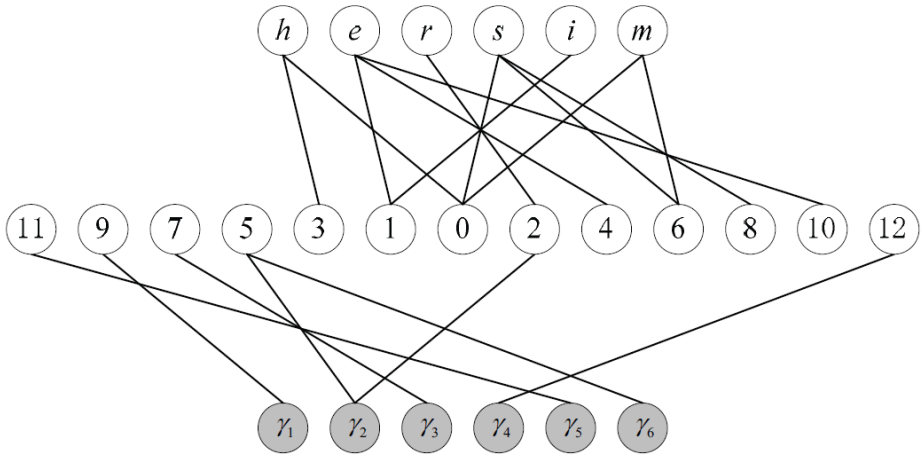

Figure 6. The bipartite graph model of AC automaton in Figure 2 with the consideration of matching rules. (Nodes in the first and third rows are in node set V.)

Perfect Hash Table Construction Algorithm With Rule Table Support

Step 3

- It is similar to 2D P^2-Hashing algorithm , except that each dependent edge set here might have two different types of edges.
- During the placement of each dependent edge set, edges are placed into the corresponding hash tables according to their types.

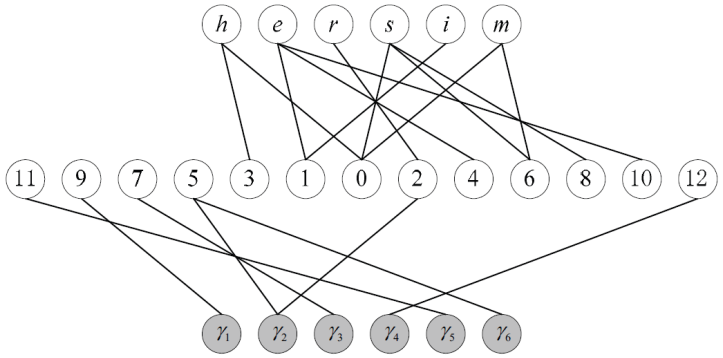

Figure 6. The bipartite graph model of AC automaton in Figure 2 with the consideration of matching rules. (Nodes in the first and third rows are in node set V.)

TABLE III. Decomposition result of the bipartite graph in Figure 6

Seq	1	2	3	4	5	6	7	8	9	10
Node	11	9	7	12	3	h	r	2	i	1
Dependent edge set	<11,γ_5>	<9,γ_1>	<7,γ_3>	<12,γ_4>	<3,h>	<0,h>	<2,r>	<2,γ_2>	<1,i>	<1,e>

Seq	11	12	13	14	15	16	17	18	19
Node	4	e	10	8	5	0	s	6	m
Dependent edge set	<4,e>	<10,e>		<8,s>	<5,γ_2> <5,γ_6>	<0,s> <0,m>	<6,s>	<6,m>	

Performance Evaluation

- The evaluated algorithms are implemented in C++ and tested on an AMD Athlon 64 X2 5200+ 2.60-GHz computer with 3-GB memory.

- Three string rule sets :

1. The first is extracted from the Snort rule set (June 2009), and includes **6.4K** string rules.
2. The second is extracted from the ClamAV rule set (June 2009), and includes **54K** string rules.
3. The third one is a subset of the first rule set including only **20 short rules with the same prefix**.

Performance Evaluation

TABLE IV. Three rule sets and their Automatons

	Snort	ClamAV	Small Set
Rule #	6.4K	54K	20
total character #	105K	6.49M	82
state # of AC automaton	77K	6.24M	24
(failure) transition # of AC automaton	77K	6.24M	23
trans. # of AC-DFA after trans. elim.	118K	7.62M	/
# of bits in state ID	19	25	7
# of bits in Character ID (if applicable)	9	9	6

Performance Evaluation

- Perfect Hash Table Construction Time

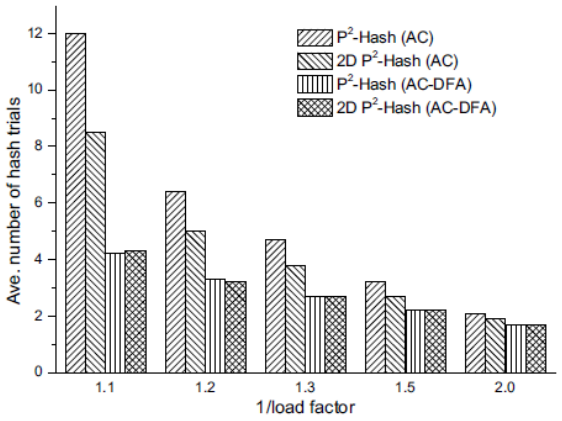

Figure 7. Average number of hash trials required for each transition placement under Snort rule set

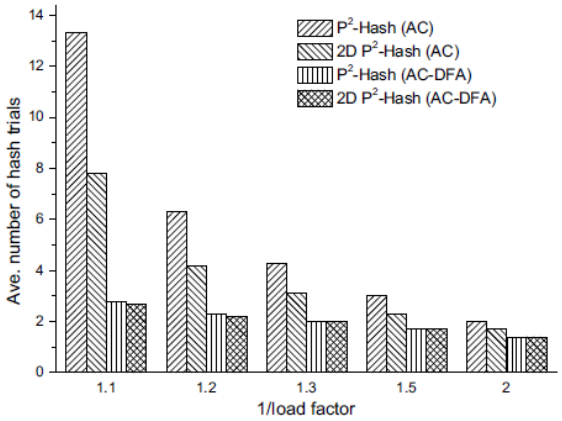

Figure 8. Average number of hash trials required for each transition placement under ClamAV rule set

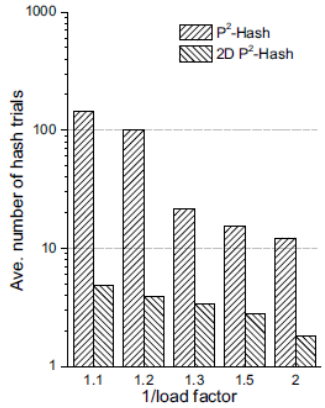

Figure 9. Average number of hash trials for each transition placement under small rule set for building AC automaton

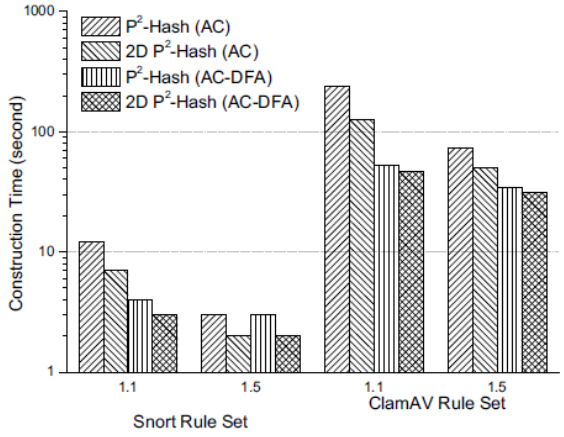

Figure 10. Perfect hash table construction time for different rule sets (load factors are set to 1/1.1 and 1/1.5)

- load factor = # of keys / # of entries of table

Performance Evaluation

- Storage Requirement

TABLE V. Memory usage comparison for Snort rule set

	AC Types	Rules #	# of Partitions	Total Characters	Total Memory	mem/ char
(2D) P^2-Hash	AC	6.4K	1	105K	699KB	7.6B
(2D) P^2-Hash	AC-DFA	6.4K	1	105K	767KB	8.33B
CDFA [17]	AC-DFA	1,785	2	29.0K	129KB~ 256KB	4.45B~ 8.2B
B-FSM [8]	AC-DFA	1.5K	4	25.2K	188KB	7.4B
Bitmap Compression [9]	AC	1.5K	1	18.2K	2.8MB	154B
Path Compression [9]	AC	1.5K	1	18.2K	1.1MB	60B

TABLE VI. Memory usage comparison for ClamAV rule set

	AC Types	Rule #	# of Partitions	Total Characters	Total Memory	mem/ char
(2D) P^2-Hash	AC	54K	1	6.49M	72.1MB	11.1B
(2D) P^2-Hash	AC-DFA	54K	1	6.49M	61.8MB	9.53B
CDFA[17]	AC-DFA	50K	32	4.44M	26.8MB	6.1B

- load factor : 90.9%