High-Performance Pattern Matching for Intrusion Detection

Nslab Seminar, Mar. 13rd, 2013

Presenter: Kyle WANG

Outline

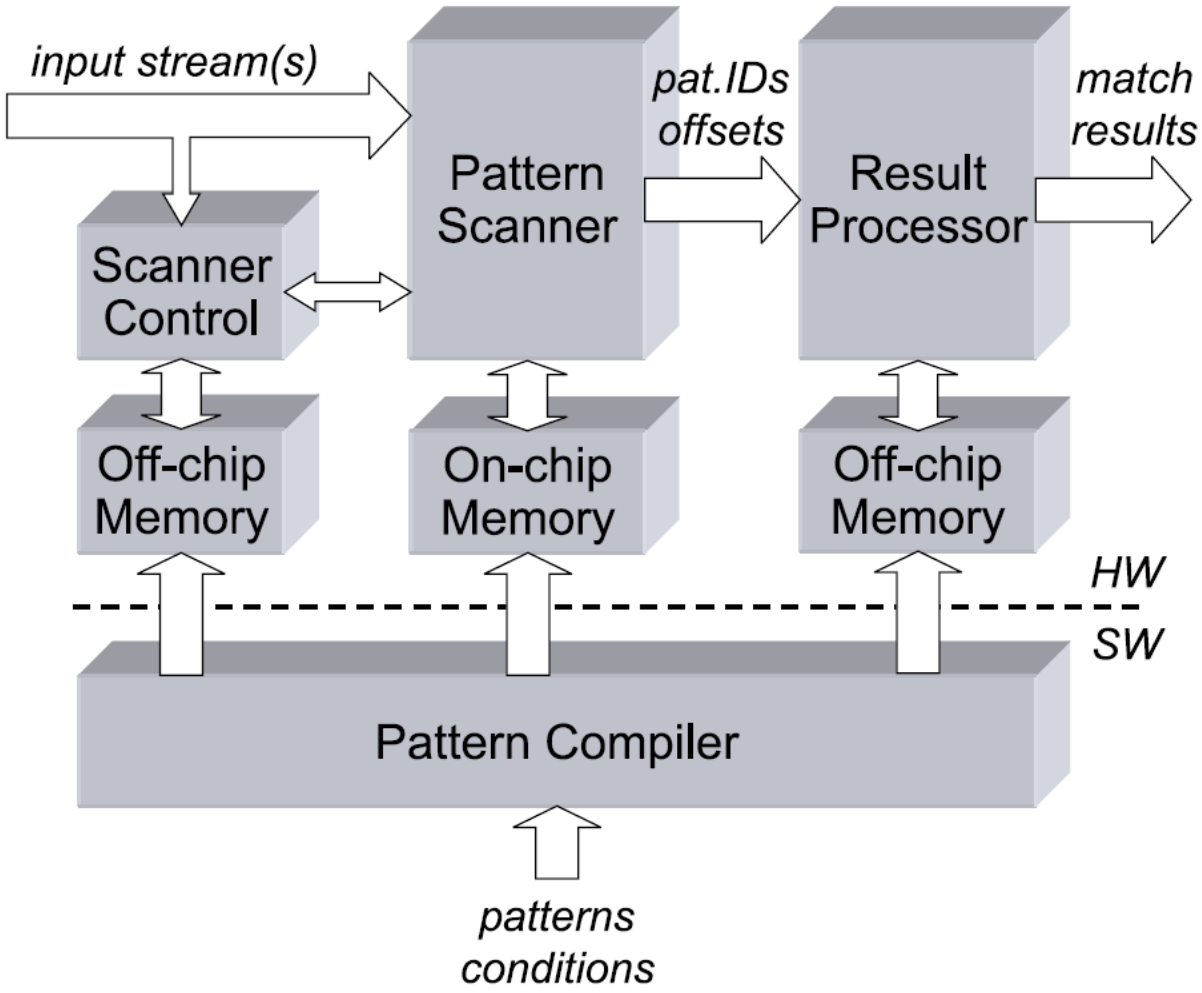
- Background
  - Pattern Matching
- Introduction
  - Balanced Routing Table (BART)
- Scheme
  - BART-based Finite State Machine (B-FSM)
  - B-FSM-based Pattern Matching (BFPM)
- Experiment
  - Performance Evaluation
  - Experiment Results
  - Algorithm Comparison
- Conclusion

Background: Pattern Matching

Design Objectives

- Deterministic performance
  - Independent of the input stream
- Storage requirements
  - Use fast but more expensive on-chip memory
- Fast dynamic updates
  - Add and remove patterns dynamically from the memory
- Patterns
  - Support strings and regular expressions
- Hardware implementation
  - Through modification of memory contents, suitable for FPGA and ASIC
- Scalability
  - Multi-chip implementation, scale to hundreds of thousands of patterns
- Others
  - E.g. Multi-session support (intermediate state)

Background: Pattern Matching

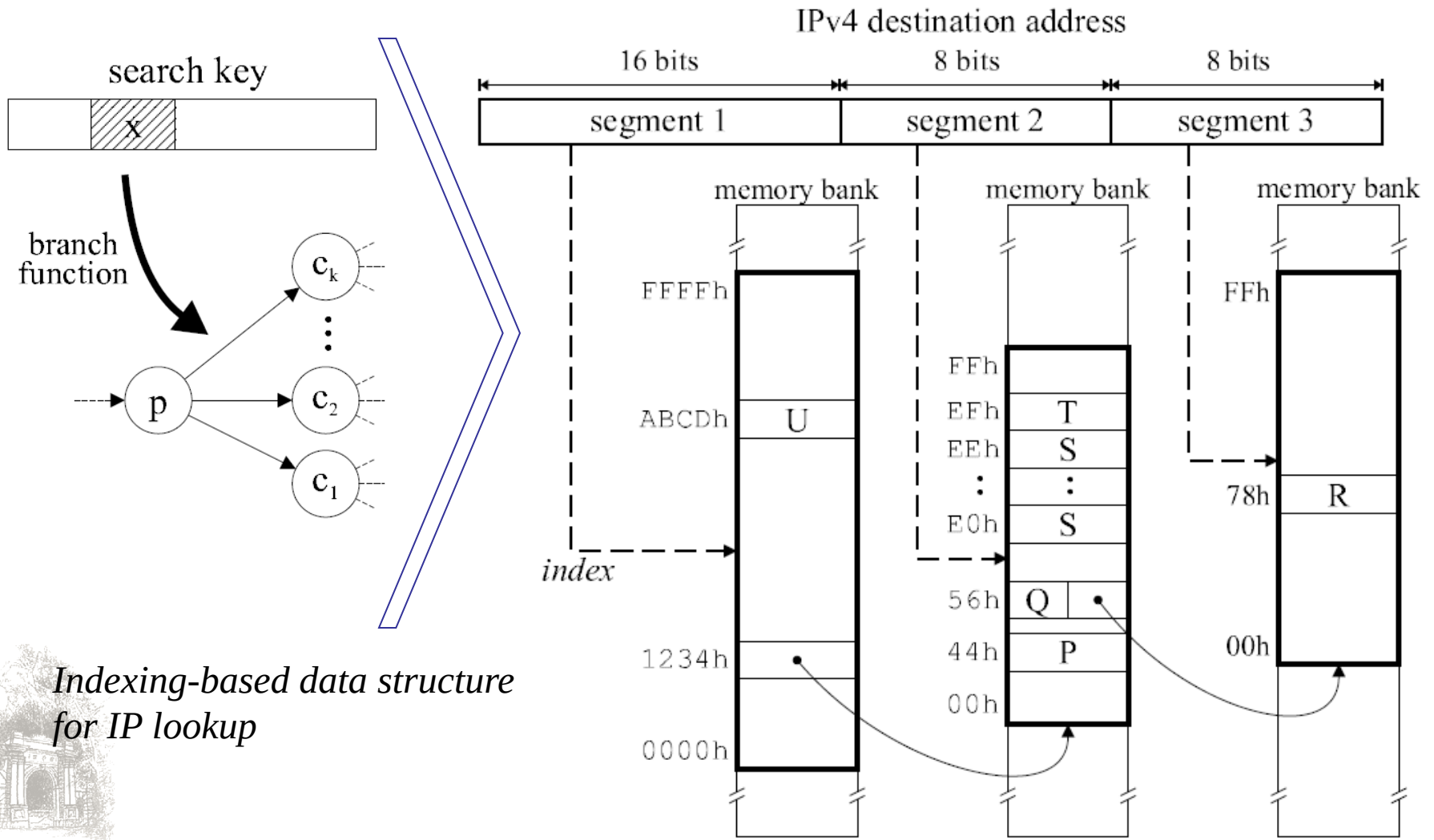


High-level block diagram of the pattern matching engine

Introduction

SAMPLE ROUTING TABLE.

prefix	length	result	prefix	length	result
123444h	24	P	1234Eh	20	S
123456h	24	Q	1234EFh	24	T
12345678h	32	R	ABCDh	16	U



Indexing-based data structure for IP lookup

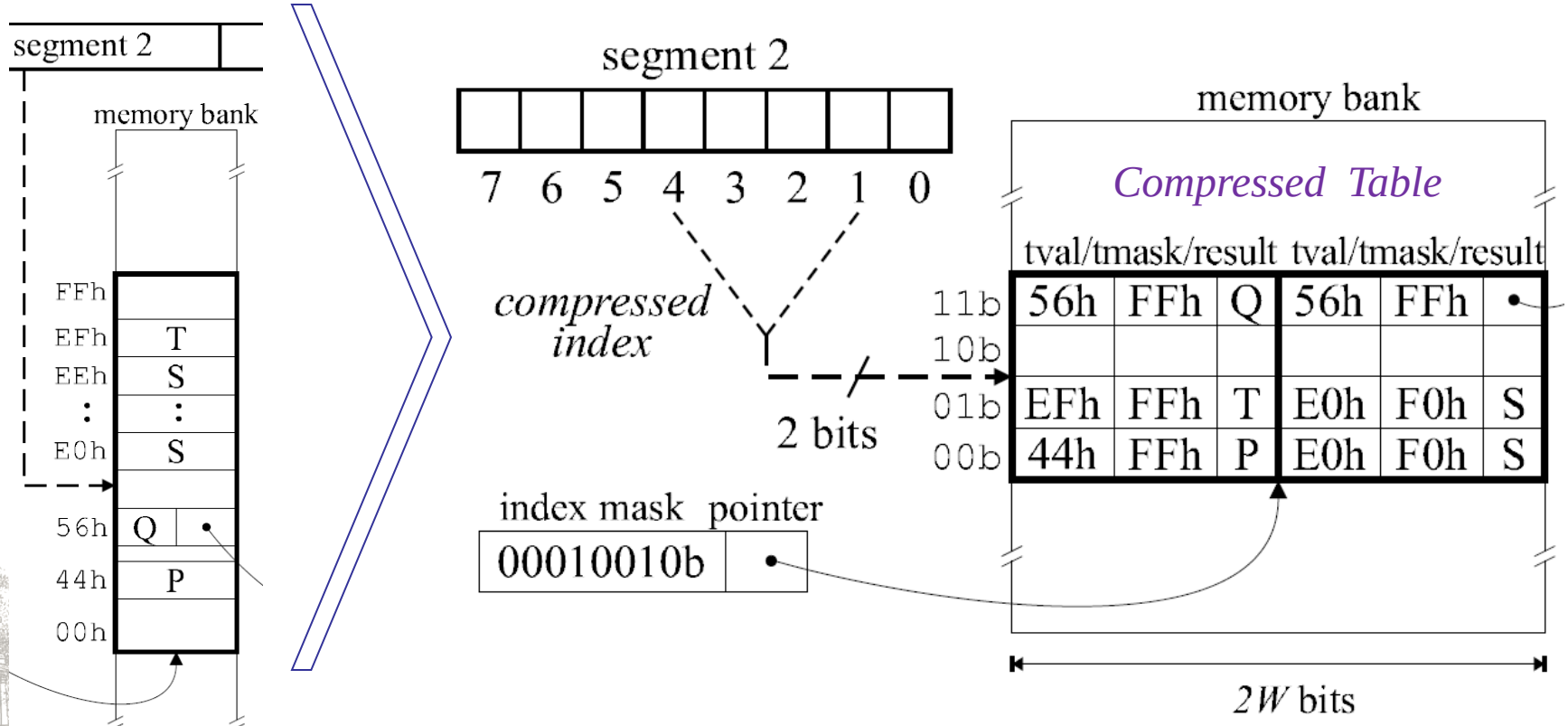
Intro: Balanced Routing Table (BART)

Compressed Index

```
01000100b   (44h)
01010110b   (56h - next hop Q)
01010110b   (56h - pointer)
1110xxxxb   (Eh)
11101111b   (EFh)
```

LOCAL PREFIXES.

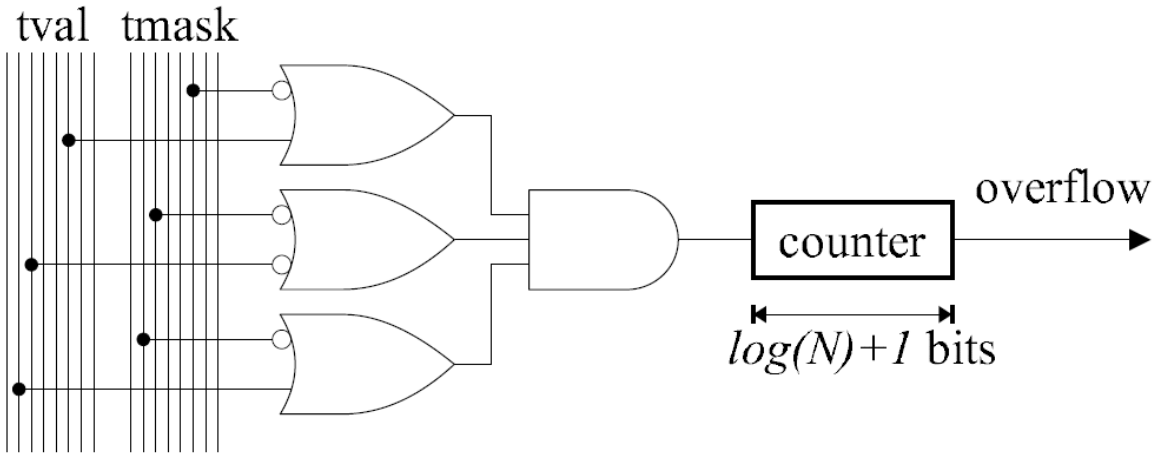
prefix	length	result
44h	8	P
56h	8	Q, ptr
Eh	4	S
EFh	8	T



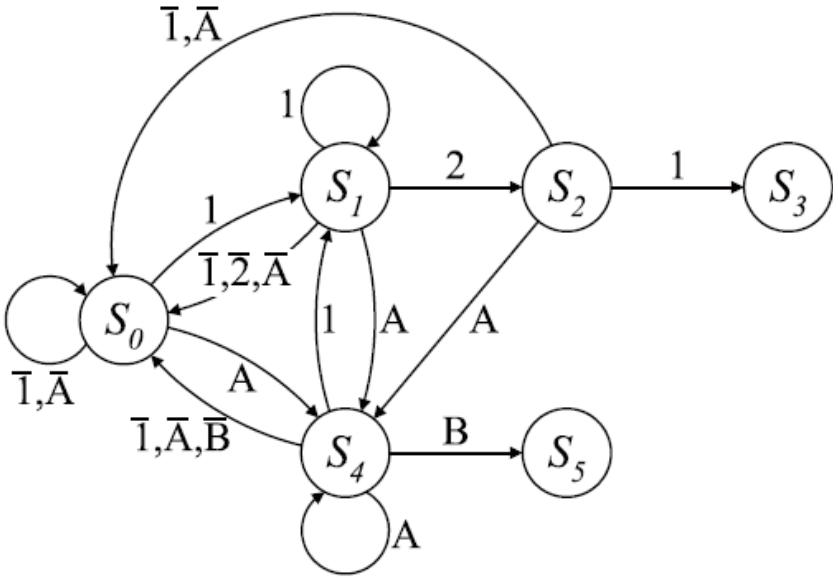
Intro: Compressed-Index Calculation

- Optimum compressed index
  - A "brute-force" count of the actual number of collisions that occur for each possible value of each possible compressed index
  - A counter array that includes one counter for every possible combination of a compressed index and a compressed-index value

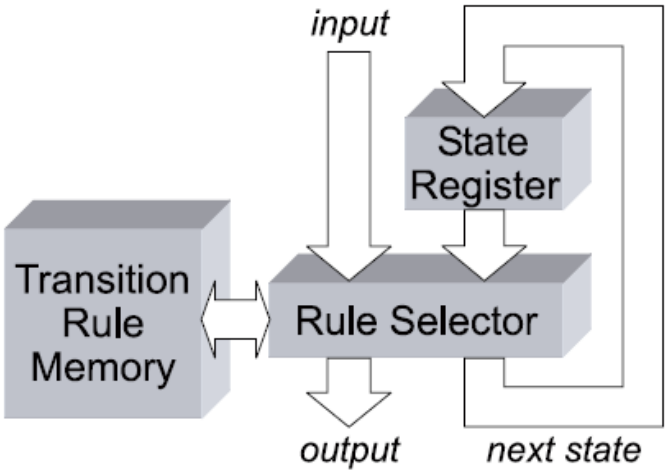
$$T = \sum_{k=0}^{s-\log(N)+1} \binom{s}{k} \times 2^k$$



Scheme: BART-based FSM (B-FSM)



(a) State-transition diagram.



(a) Block diagram.

	test part			result part	
current state	input	conditions	next state	table address	mask

(b) Transition-rule vector.

rule	current state	input	→	next state	priority
R_1	S_2	1h	→	S_3	2
R_2	*	1h	→	S_1	1
R_3	S_1	2h	→	S_2	1
R_4	S_4	Bh	→	S_5	1
R_5	*	Ah	→	S_4	1
R_6	*	*	→	S_0	0

Two patterns /**121**h/ and /**Ab**h/

Scheme: BART-based FSM

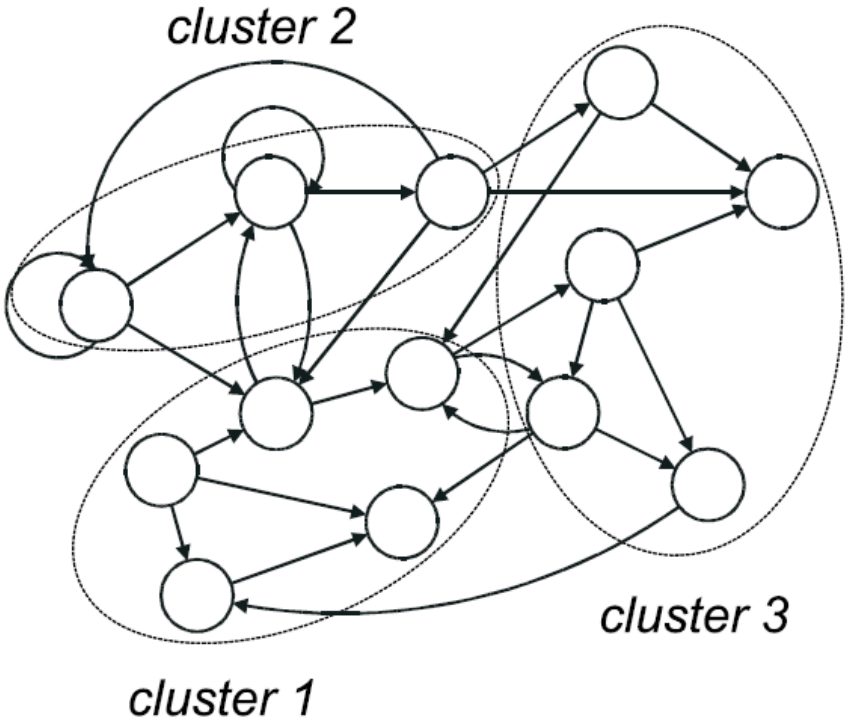
rule	current state	input	→	next state	priority
R_1	S_2	0001b	→	S_3	2
R_2	*	0001b	→	S_1	1
R_3	S_1	0010b	→	S_2	1
R_4	S_4	1011b	→	S_5	1
R_5	*	1010b	→	S_4	1
R_6	*	xxxxb	→	S_0	0

(a) Transition rules.

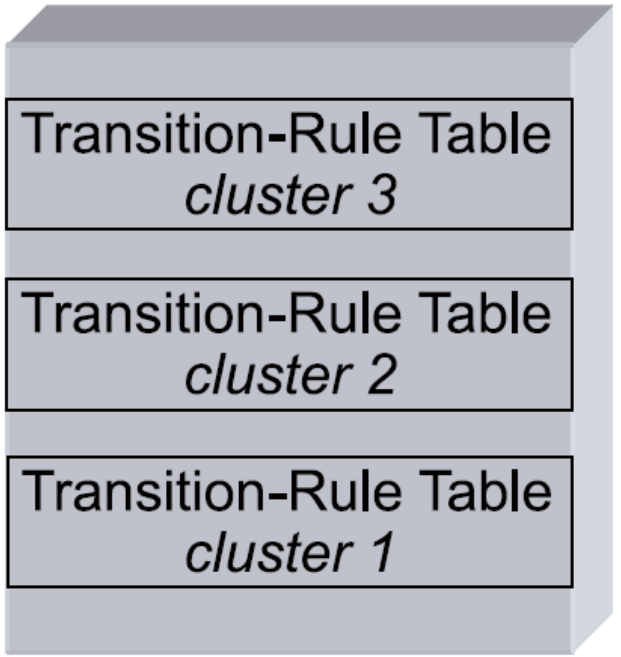
1	rule R_4	rule R_5	rule R_6	
0	rule R_1	rule R_2	rule R_3	rule R_6

(b) BART-compressed transition-rule table.

Scheme: State Clusters



(a) State-transition diagram.



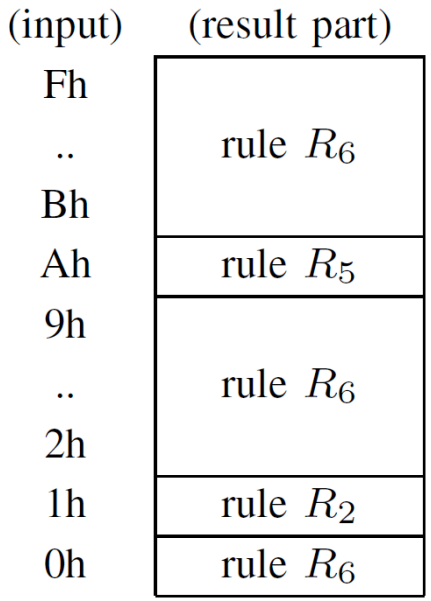
(b) Transition-rule memory.

Scheme: Optimizations

- Don't-care rules
  - (1) A default rule having a wildcard condition for the current state and input, and priority 0
  - (2) transition rules having a wildcard condition for the current state, an exact-match condition for the input, and priority 1
  - (3) transition rules having exact-match conditions for the current state and input, and priority 2
  - For (1) and (2), only lookup on the input value
  - The remaining transition rules of the third

  type are stored in a BART-compressed

  transition-rule table as described above



Scheme: Optimizations

- State Encoding and Index Calculation

$$index = (state' \textbf{ and not } mask) \textbf{ or } (input' \textbf{ and } mask)$$

  - The *mask* vector determines for each index bit if it is extracted *from either the current state value* (mask bit is zero) or *from the input symbol* (mask bit is one)
  - It means, arbitrary bit in *state* and *input* can be used to distribute the transition rule

  - (1) At most *P* transition rules for a state, *mask* can be zero
  - (2) Less than *P* transition rules, correctly fill with the valid transition rules from other state' (state' is the *mask* subset of the state vector)
  - (3) More than *P* transition rules, let *mask* select a minimum number of input bits for distribute these rules over multiple table entries with at most *P* rules per entry

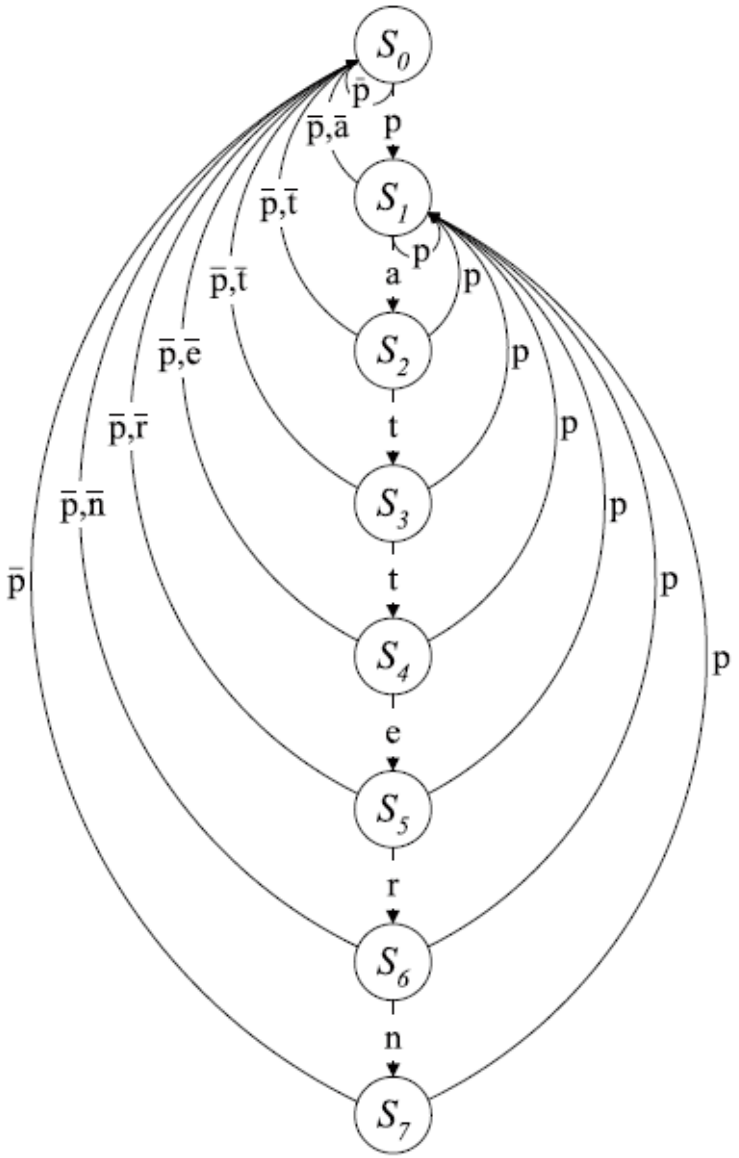
Scheme: Optimizations

- Transition-rule Table Construction Procedure
  - (1) The first state (Start with the states that have the most transition rules) will be placed in a new cluster and will be assigned a local state vector equal to 0.
  - (2) If the number of transition rules is larger than *P*, then an optimal index *mask* is calculated for distributing these transition rules over multiple table entries. Otherwise an index mask equal to 0 is used.
  - (3) Check which of the remaining states and corresponding transition rules, can be mapped on the empty entries in the transition-rule table based on the above index *mask*, by varying the encoding of the local state vector. All such states will be added to this state cluster.
  - After all states have been processed, iterate (1), (2) and (3) for the remaining states and transition rules , by creating a new cluster and transition-rule table.

Scheme: B-FSM-based PM (BFPM)

A Distributed B-FSM Approach

Scheme: String Matching



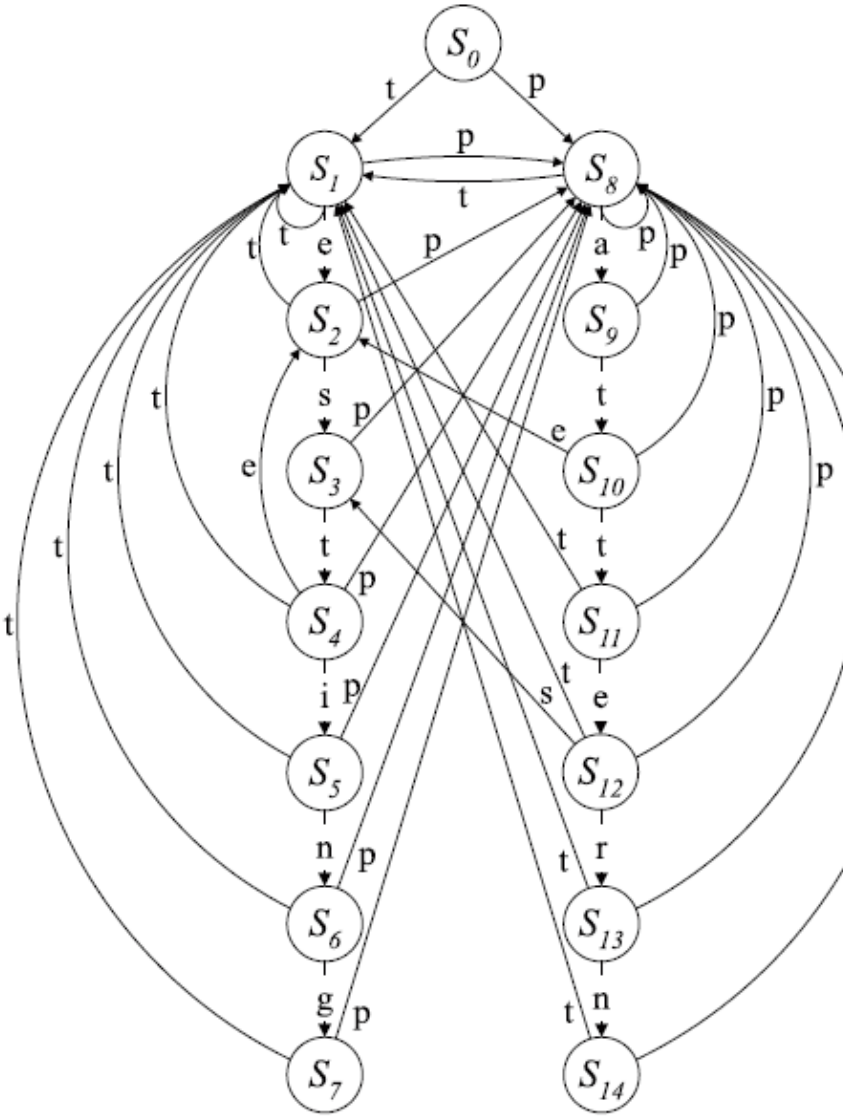
(a) State-transition diagram (prior-art) for example 1.

Example 1: "pattern"

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	p	→	S_1	1
R_2	S_1	a	→	S_2	2
R_3	S_2	t	→	S_3	2
R_4	S_3	t	→	S_4	2
R_5	S_4	e	→	S_5	2
R_6	S_5	r	→	S_6	2
R_7	S_6	n	→	S_7	2

(b) Transition rules for example 1.

Scheme: String Matching



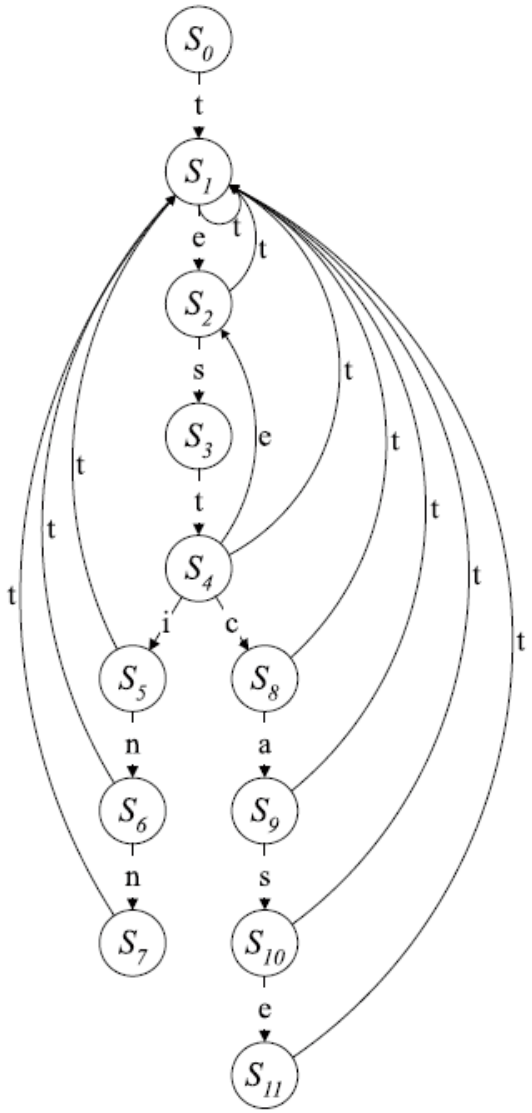
(a) State-transition diagram (prior-art) for example 2 ("default" transitions to state S0 are not shown).

Example 2: "testing" and "pattern"

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	t	→	S_1	1
R_2	S_1	e	→	S_2	2
R_3	S_2	s	→	S_3	2
R_4	S_3	t	→	S_4	2
R_5	S_4	i	→	S_5	2
R_6	S_5	n	→	S_6	2
R_7	S_6	g	→	S_7	2
R_8	*	p	→	S_8	1
R_9	S_8	a	→	S_9	2
R_{10}	S_9	t	→	S_{10}	2
R_{11}	S_{10}	t	→	S_{11}	2
R_{12}	S_{11}	e	→	S_{12}	2
R_{13}	S_{12}	r	→	S_{13}	2
R_{14}	S_{13}	n	→	S_{14}	2
R_{15}	S_4	e	→	S_2	2
R_{16}	S_{10}	e	→	S_2	2
R_{17}	S_{12}	s	→	S_3	2

(b) State-transition rules for example 2.

Scheme: String Matching



(a) State-transition diagram (prior-art) for example 3 ("default" transitions to state S0 are not shown).

Example 3: "testing" and "testcase"

rule	current state	input	→	next state	priority
R_0	*	*	→	S_0	0
R_1	*	t	→	S_1	1
R_2	S_1	e	→	S_2	2
R_3	S_2	s	→	S_3	2
R_4	S_3	t	→	S_4	2
R_5	S_4	i	→	S_5	2
R_6	S_5	n	→	S_6	2
R_7	S_6	g	→	S_7	2
R_8	S_4	c	→	S_8	2
R_9	S_8	a	→	S_9	2
R_{10}	S_9	s	→	S_{10}	2
R_{11}	S_{10}	e	→	S_{11}	2
R_{12}	S_4	e	→	S_2	2

(b) State-transition rules for example 3.

Scheme: String Matching

state	prefix	state	prefix
S_1	*t*	S'_1	*t*
S_2	*te*	S'_2	*te*
S_3	*tes*	S'_3	*tes*
S_4	*test*	S'_4	*test*
S_5	*testi*	S'_5	*testc*
S_6	*testin*	S'_6	*testca*
S_7	*testing*	S'_7	*testcas*
		S'_8	*testcase*

(a) States (step 1).

state	prefix	state	prefix
S_1	*t*	S_8	*testc*
S_2	*te*	S_9	*testca*
S_3	*tes*	S_{10}	*testcas*
S_4	*test*	S_{11}	*testcase*
S_5	*testi*		
S_6	*testin*		
S_7	*testing*		

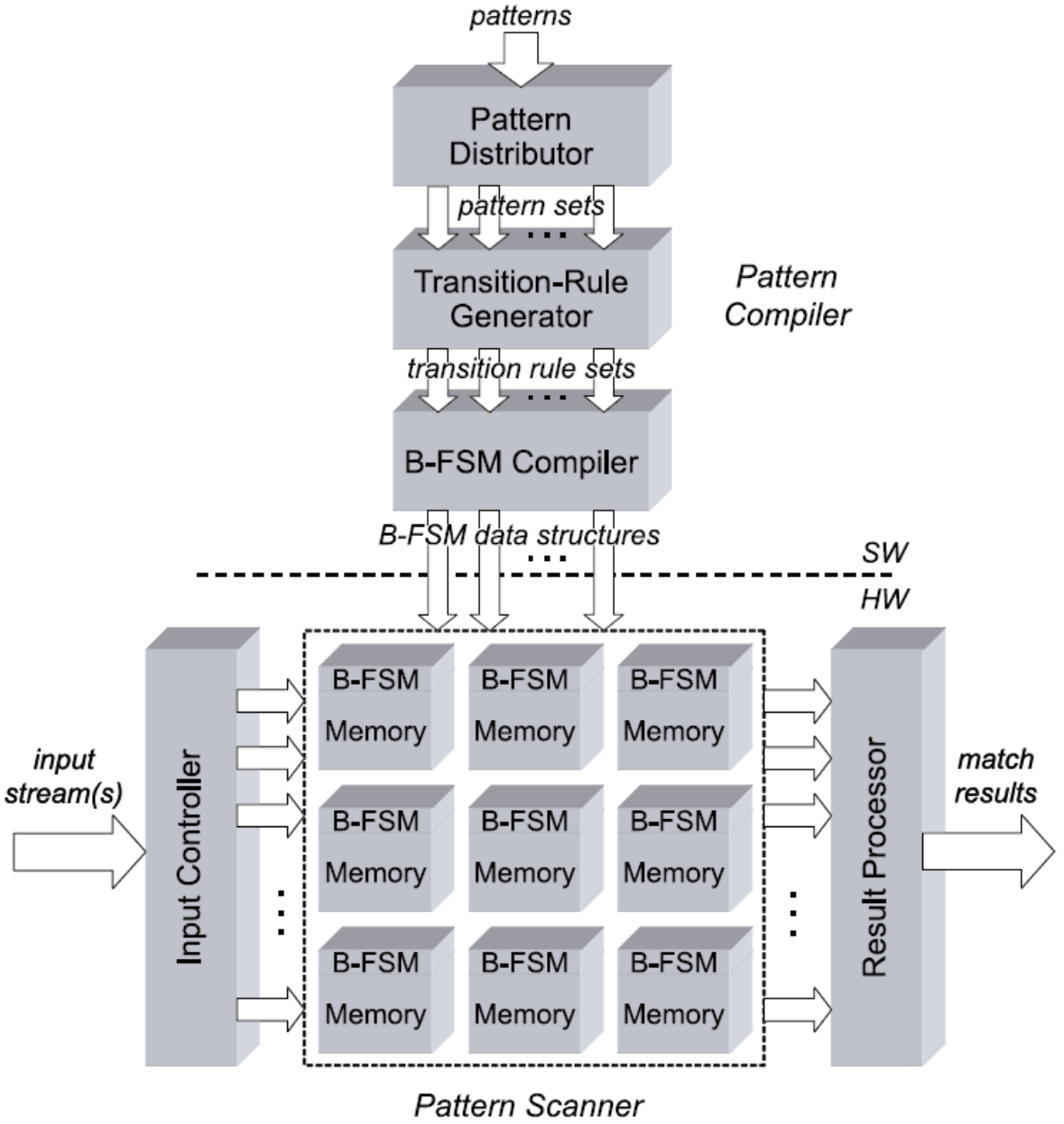
(b) Filtered states (step 2).

Transition-Rule Generation

state	prefix	state	prefix	rule
S_1	*t*	S_2	*te*	R_2
S_2	*te*	S_3	*tes*	R_3
S_3	*tes*	S_4	*test*	R_4
S_4	*test*	S_2	*te*	R_{12}
S_4	*test*	S_5	*testi*	R_5
S_4	*test*	S_8	*testc*	R_8
S_5	*testi*	S_6	*testin*	R_6
S_6	*testin*	S_7	*testing*	R_7
S_8	*testc*	S_9	*testca*	R_9
S_9	*testca*	S_{10}	*testcas*	R_{10}
S_{10}	*testcas*	S_{11}	*testcase*	R_{11}

(c) State combinations (step 5).

Scheme: Pattern Compiler



Scheme: Dynamic Incremental Update

- (1) By creating copies of the modified transition rule tables which are then linked into the data structure using atomic write operations

- (2) By writing the entire updated B-FSM data structure into the transition-rule memory of an additional non-active B-FSM, which is then activated

Experiment: Evaluation

- Processing Rate
- Storage Requirements
  - A random set of n patterns, comprising a total of m characters, an average pattern length of $k = m/n$ characters, and N is the number of B-FSMs
  - The average number of intra-pattern conflicts will increase with the pattern length according to $O(k)$
  - Total number of intra-pattern conflicts is $O(n/N*k)*N = O(m/N)*N = O(m)$
  - The average number of inter-pattern conflicts will increase with the number of patterns and its pattern length according to $O(n/N+ck)$
  - Total number of inter-pattern conflicts is $O((n/N)^2+cn/N*k)*N = O((n/N)^2+cm/N)*N = O(n^2/N+cm)$
  - Totally $O(m+c'n^2/N)$, if N increases with n, then $O(m+c''n)$.
- Update Performance
  - At least $O(n^2)$

Experiment: Result and Comparison

- Storage Requirements

PATTERN COMPILER PERFORMANCE.

B-FSMs	1.5 K case-insensitive patterns, 25.2 K characters					0.48 K case-sensitive patterns, 6.4 K characters				
	rules	rules/char	memory	(allocated)	mem/char	rules	rules/char	memory	(allocated)	mem/char
4	39.5 K	1.57	183 KB	(188 KB)	7.4 B	11.1 K	1.75	49 KB	(52 KB)	7.9 B
6	32.1 K	1.27	149 KB	(152 KB)	6.0 B	8.6 K	1.36	38 KB	(39 KB)	6.1 B
8	25.8 K	1.02	117 KB	(120 KB)	4.7 B	7.7 K	1.21	34 KB	(37 KB)	5.4 B
12	22.0 K	0.87	99 KB	(104 KB)	4.0 B	6.6 K	1.03	29 KB	(36 KB)	4.6 B
16	18.9 K	0.75	83 KB	(92 KB)	3.4 B	6.2 K	0.97	27 KB	(36 KB)	4.3 B
20	18.8 K	0.74	82 KB	(93 KB)	3.3 B	5.8 K	0.92	26 KB	(35 KB)	4.1 B

- Processing Rate
  - Virtex-4, 100~125MHz, dual-port, 1.6 to 2 Gb/s for each B-FSM
  - 1MB BRAM, 2K patterns size128KB, 12~16 Gb/s for single chip
- Comparison
  - Turk 8 Gb/s in ASIC
  - BFPM 20 Gb/s in ASIC

STORAGE-EFFICIENCY COMPARISON (BASED ON [14]).

method	memory	mem/char
Aho-Corasick [6]	53.1 MB	2.8 KB
Wu-Manber [9]	29.1 MB	1.6 KB
Bitmap compr. Aho-Corasick [14]	2.8 MB	154 B
Path compr. Aho-Corasick [14]	1.1 MB	60 B

Thanks & Questions