Zurich Research Laboratory

High-Performance Pattern-Matching Engine for Intrusion Detection

A New Approach for Fast Programmable Accelerators

Jan van Lunteren
Ton Engbersen

Agenda

1. Overview
2. Programmable State Machine
3. String Matching
4. Pattern Compiler
5. Regular Expressions
6. Experimental Results
7. Summary

Overview

Patterns

- Strings, regular expressions
- Pattern conditions: case sensitivity, location in input stream, negation
- Multi-pattern conditions: order, distance
- Scalable to at least tens of thousands of patterns
- No basic limit on maximum pattern length (memory capacity)

Operation

- Deterministic processing rate, independent of input and number of patterns
- On-the-fly, single-pass processing of entire input stream(s)
- Detection of all patterns, including multiple occurrences and overlaps
- Dynamic updates of patterns by modifying memory contents

Performance

- Aggregate processing rate (single chip): 5-10 Gb/s (FPGA), >20 Gb/s (ASIC)
- Update time (insert/delete): approx. one millisecond per pattern
- Storage efficiency: 1,500 patterns / 25 K characters fit into <100 KB

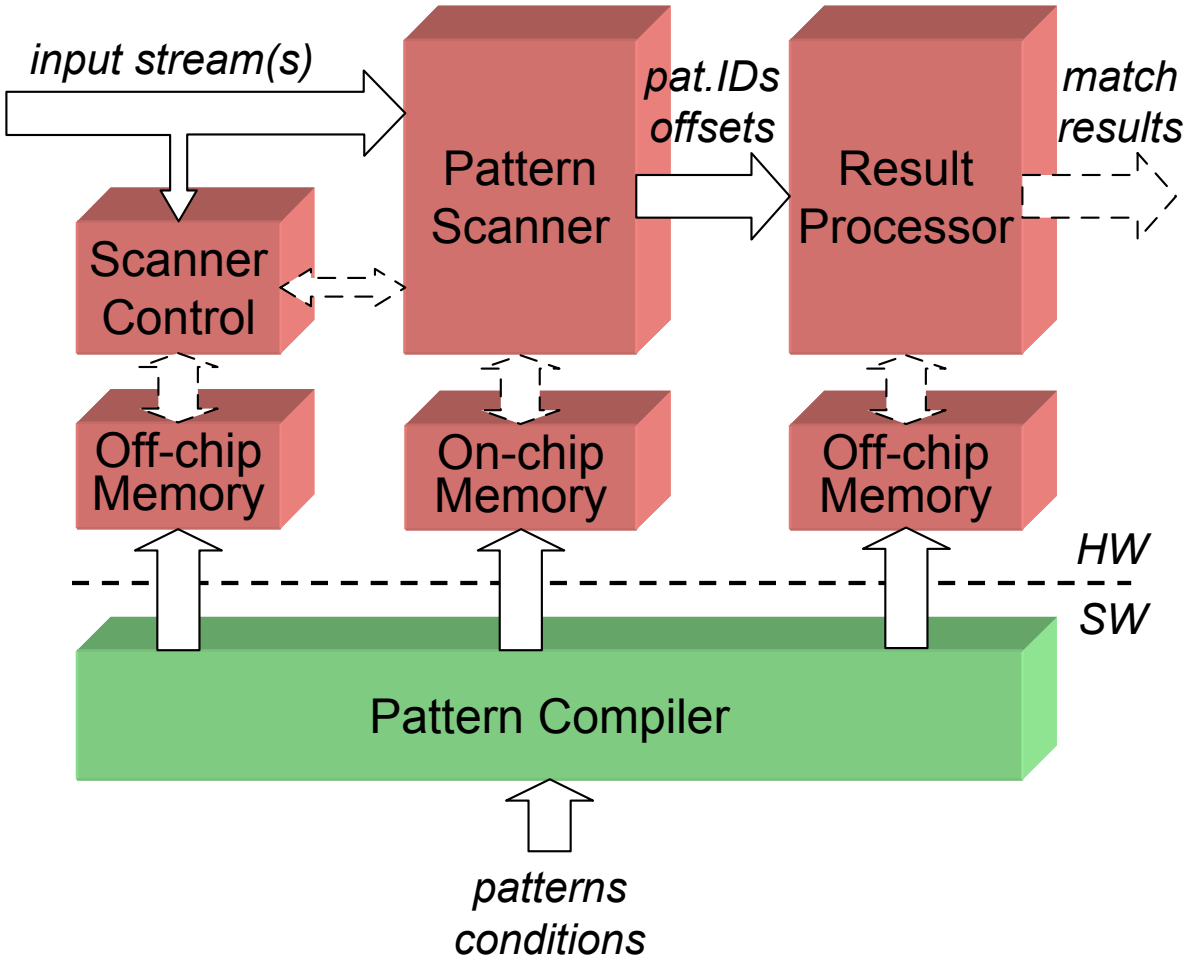
Block Diagram

Scanner Control
- Session management
- Pattern subset selection
- Resource allocation

Pattern Scanner
- String matching
- Regular expressions

Result Processor
- Pattern conditions
  - location, negation
  - order, distance
- Output generation

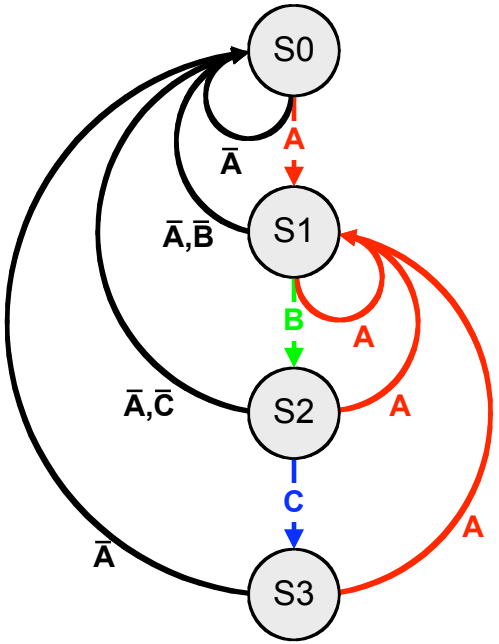
Pattern Compiler
- Generation and dynamic update of data structures



Programmable State Machine

Example

- Detection of all occurrences of a pattern *ABC* in the input stream



state transition diagram

- Can be described using *state transition rules* involving wildcards and priorities
- In each cycle, the highest-priority transition rule matching the current state and input is selected

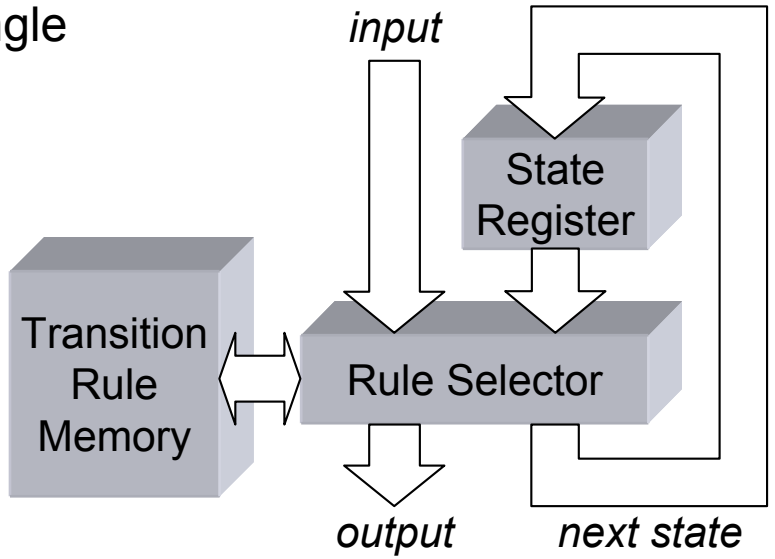
rule	state	input	->	state	priority
R0	*	*	->	S0	0
R1	*	A	->	S1	1
R2	S1	B	->	S2	2
R3	S2	C	->	S3	2

state transition rules

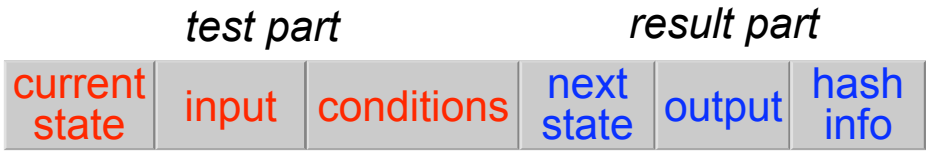
Programmable State Machine

B-FSM technology

- Based on *BaRT* routing-table search algorithm (hash function)
  - 72K IPv4 prefixes in ~500 KB, ≤4 memory accesses per lookup
- Finds highest-priority matching rule in a single clock cycle, at frequencies on the order of 100 MHz (FPGA) – 1 GHz (custom logic)
- High storage efficiency: typically linear relation between storage requirements and number of transition rules
- Scalable to hundreds of thousands of states and transitions
- Supports wide input and output vectors
- Fast dynamic updates

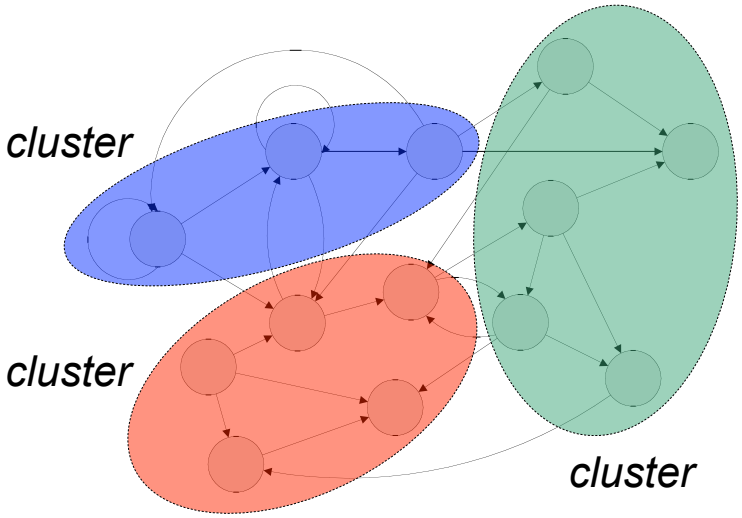


B-FSM engine

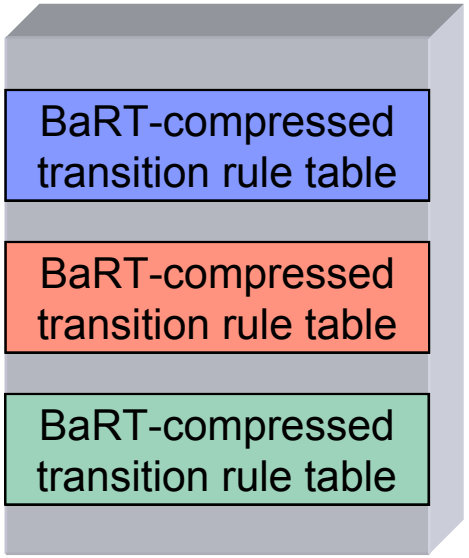


transition rule vector

Programmable State Machine



state transition diagram



transition rule memory

Optimizations

- State transition diagram is divided into state clusters
- Clusters are mapped on BaRT-compressed transition-rule tables
- Optimized state encoding within each cluster

☞ High performance: simplified hash function enables small and fast B-FSM logic
☞ High storage efficiency: optimum filling of hash tables

Pattern Scanner

Array of B-FSM engines

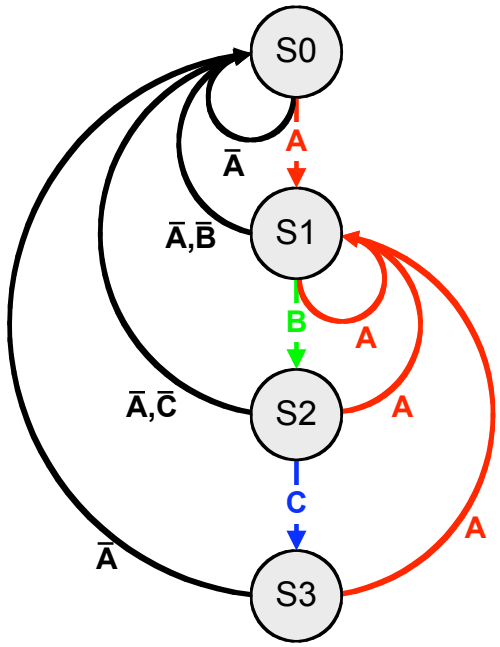
- Increased *performance:* smaller memories are faster
- Increased *storage efficiency*: selective distribution of patterns over engines (next slides)
- Increased *flexibility*: programmable allocation of B-FSM engines to input stream(s) and applications (next slides)

String Matching

- Example: detection of all occurrences of the pattern *ABC* anywhere in the input stream
- Pattern can be mapped directly on transition rules (state transition diagram is not necessary)

rule	state	input	->	state	prior.
R0	*	*	->	S0	0
R1	*	A	->	S1	1
R2	S1	B	->	S2	2
R3	S2	C	->	S3	2

- Avg. rules/character = 1.0 (not counting default rule)



state transition diagram (for illustration only)

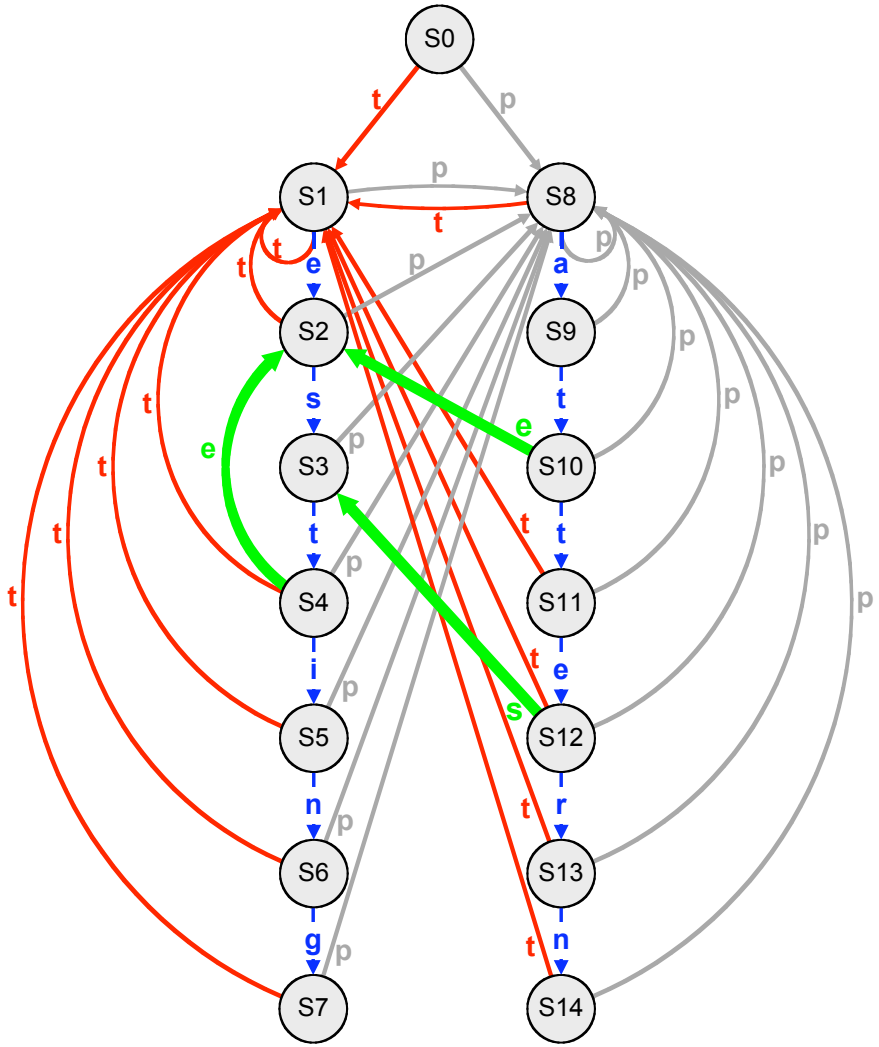
String Matching

- Example: detection of all occurrences of the patterns *testing* and *pattern*

rule	state	input	->	state	prior.
R0	*	*	->	S0	0
R1	*	t	->	S1	1
R2	S1	e	->	S2	2
R3	S2	s	->	S3	2
R4	S3	t	->	S4	2
R5	S4	i	->	S5	2
R6	S5	n	->	S6	2
R7	S6	g	->	S7	2
R8	*	p	->	S8	1
R9	S8	a	->	S9	2
R10	S9	t	->	S10	2
R11	S10	t	->	S11	2
R12	S11	e	->	S12	2
R13	S12	r	->	S13	2
R14	S13	n	->	S14	2
R15	S4	e	->	S2	2
R16	S10	e	->	S2	2
R17	S12	s	->	S3	2

- Pattern *conflicts*: rules 15-17 handle input streams containing *testesting*, *patesting* or *pattesting*
- Avg. rules/character = 1.2

("default" transitions to S0 are not shown)



state transition diagram (for illustration only)

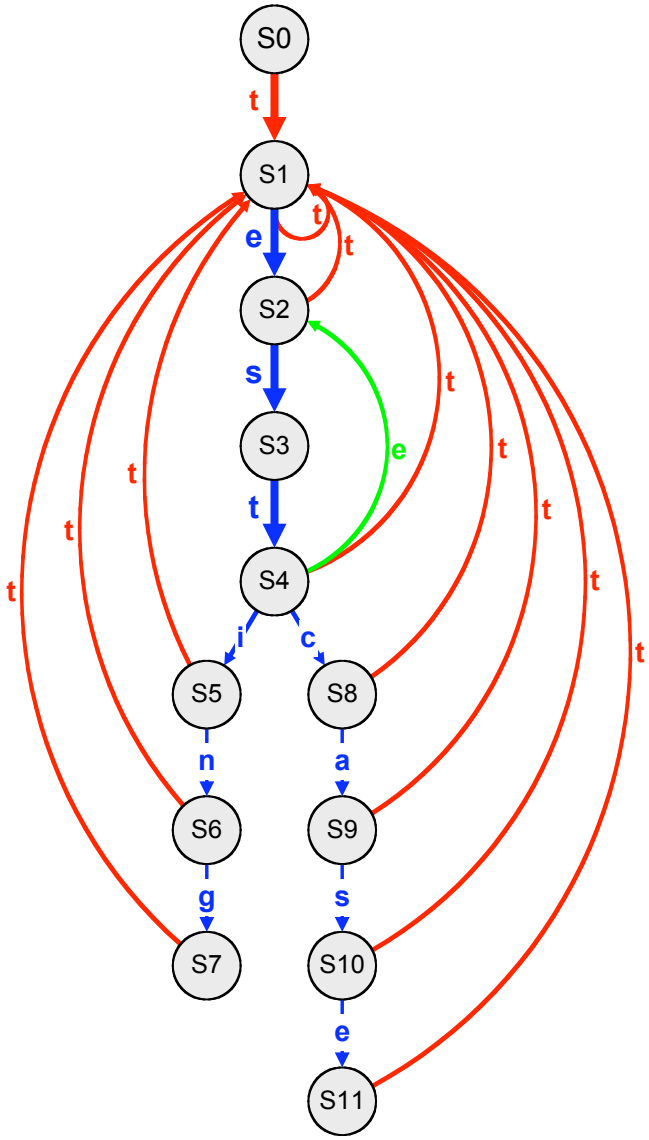
String Matching

- Example: detection of all occurrences of the patterns *testing* and *testcase*

rule	state	input	->	state	prior.
R0	*	*	->	S0	0
R1	*	t	->	S1	1
R2	S1	e	->	S2	2
R3	S2	s	->	S3	2
R4	S3	t	->	S4	2
R5	S4	i	->	S5	2
R6	S5	n	->	S6	2
R7	S6	g	->	S7	2
R8	S4	c	->	S8	2
R9	S8	a	->	S9	2
R10	S9	s	->	S10	2
R11	S10	e	->	S11	2
R12	S4	e	->	S2	2

- Common *prefix test*: rules 1-4 are shared by both patterns
- Avg. rules/character = 0.8

("default" transitions to S0 are not shown)

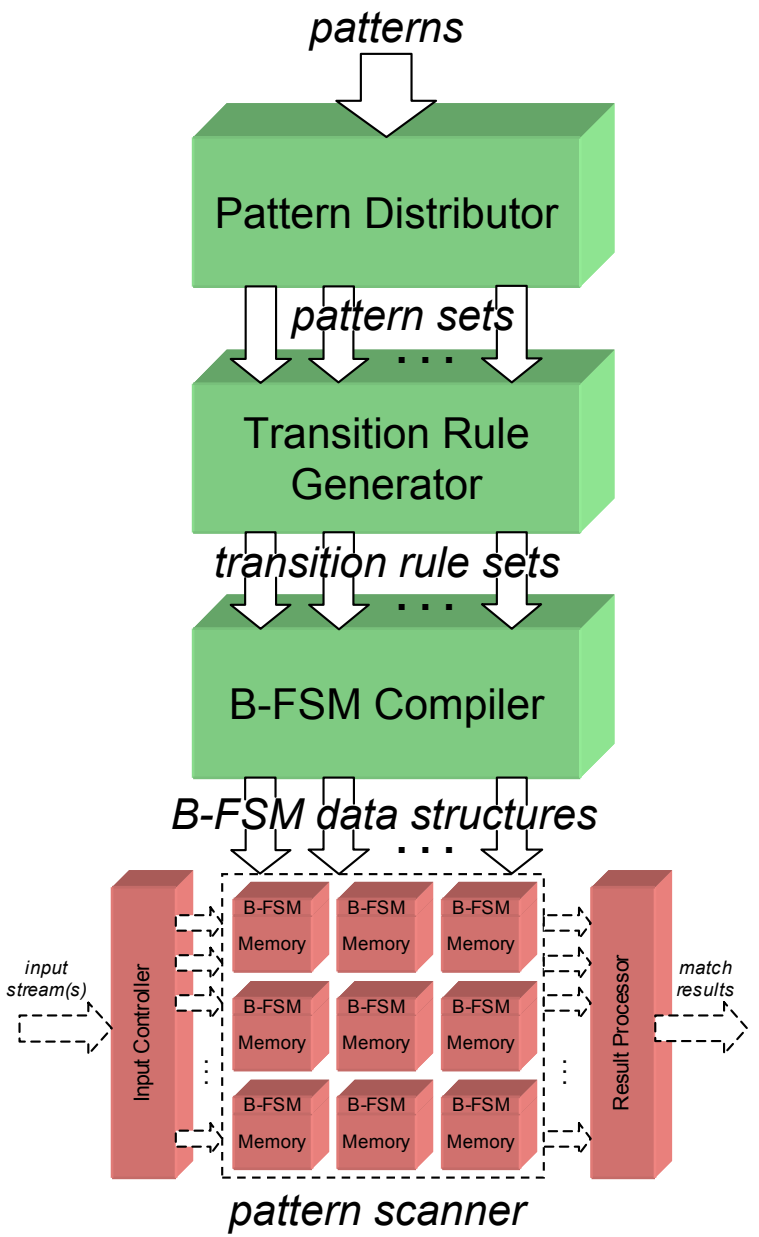


(for illustration only)

Pattern Compiler

Overview of Operation

- *Pattern distributor* distributes patterns over multiple pattern sets, based on pattern properties (conflicts, overlaps)
- *Transition rule generator* converts each pattern set into state transition rules, resolving intra/inter-pattern conflicts
- *B-FSM compiler* converts transition rules into B-FSM data structures using state clustering, state encoding, and BaRT compression
- Each step supports *incremental* updates



Pattern Compiler Performance

# B-FSMs	trans. rules	rules/char	memory	(allocated[1])	mem/char
4	39.5 K	1.57	183 KB	(188 KB)	7.4 B
6	32.1 K	1.27	149 KB	(152 KB)	6.0 B
8	25.8 K	1.02	116 KB	(120 KB)	4.7 B
12	22.0 K	0.87	99 KB	(104 KB)	4.0 B
16	18.9 K	0.75	83 KB	(92 KB)	3.4 B
20	18.8 K	0.74	82 KB	(93 KB)	3.3 B

Experiment

- 1.5 K strings / 25 K characters (case-insens.) extracted from Snort® rules (2004)
- Pattern length 1-100 characters; average: 16 characters
- Memory is allocated in buffers of ~1 KB ([1] includes free space for additional rules)

Comparison

- Based on: N.Tuck et al., *"Deterministic memory-efficient string matching algorithms for intrusion detection,"* Infocom'04
- 1.5 K strings / 20 K characters (Snort® rules, 2003)

Method	memory	mem/char
Aho-Corasick	53.1 MB	2.8 KB
Wu-Manber	29.1 MB	1.6 KB
bitmap-compr. Aho-Cor.	2.8 MB	154 B
path-compr. Aho-Cor.	1.1 MB	60 B

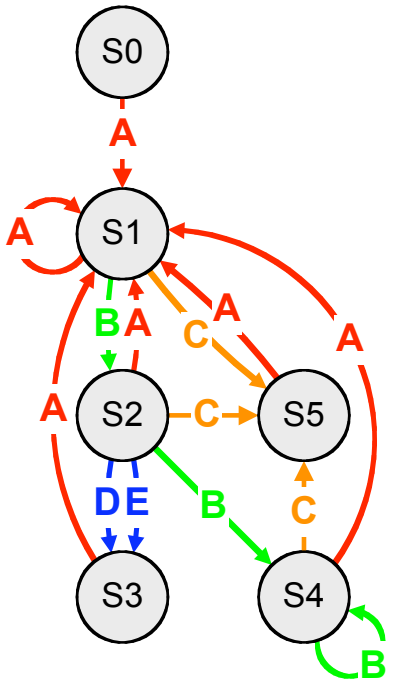
Snort is a registered trademark of Sourcefire, Inc.

Regular Expressions

- Example: detection of all matches of regular expressions AB[D|E] and AB*C

rule	state	input	->	state	prior.
R0	*	*	->	S0	0
R1	*	A	->	S1	1
R2	S1	B	->	S2	2
R3	S2	B	->	S4	2
R4	S4	B	->	S4	2
R5	S2	D	->	S3	2
R6	S2	E	->	S3	2
R7	S1	C	->	S5	2
R8	S2	C	->	S5	2
R9	S4	C	->	S5	2

("default" transitions to S0 are not shown)



state transition diagram (for illustration only)

Advanced Support

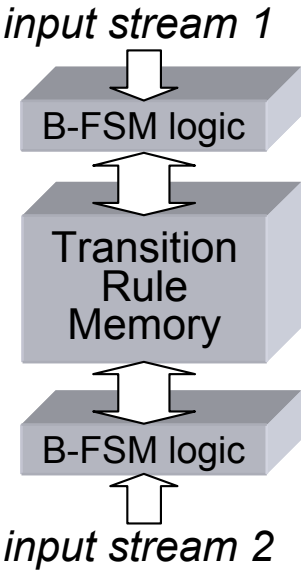
- Enhanced B-FSMs provide efficient support for
  - character classes
  - counters

Experimental Results

FPGA Implementation

- Xilinx® Virtex-4™ technology
  - two B-FSMs per dual-port transition rule memory
  - one transition per clock cycle @ 125 MHz
  - one input byte per transition

☞ Two *independent* pattern-matching channels per transition rule memory, each running at 1 Gb/s



Configuration options (example)

- Assuming linear storage increase
  - each 100 KB can hold ~ 1.5 K patterns / 25 K characters
- A single-chip FPGA with 400 KB transition rule memory can hold:

# copies of data structure	patterns	characters	# channels	throughput
4	1.5 K	25 K	8	8 Gb/s
2	3 K	50 K	4	4 Gb/s
1	6 K	100 K	2	2 Gb/s

Xilinx and Virtex-4 are registered trademarks of Xilinx, Inc.

Summary

Novel Concept

- The problem of simultaneously detecting thousands of patterns in an input stream can be mapped very efficiently on transition rules involving wildcard conditions and priorities

Pattern Compiler

- Maps patterns on transition rules
- Converts transition rules into hash-table structures executed directly in HW
- Optimizes at the level of transition rules and hash tables

Pattern-Matching Engine

- Deterministic processing rate, independent of input and number of patterns
  - 5-10 Gb/s (FPGA), >20 Gb/s (ASIC)
- High storage efficiency: 1,500 patterns / 25 K characters fit into <100 KB
- Fast incremental updates: approx. one millisecond per pattern
- Scalable to at least
  - tens of thousands of patterns
  - hundreds of thousands of characters

B-FSM Technology

- Enables fast *programmable* accelerators for a wide range of applications

For more information, contact:

Jan van Lunteren (jvl@zurich.ibm.com)
Ton Engbersen (apj@zurich.ibm.com)

IBM Research GmbH
Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon
Switzerland

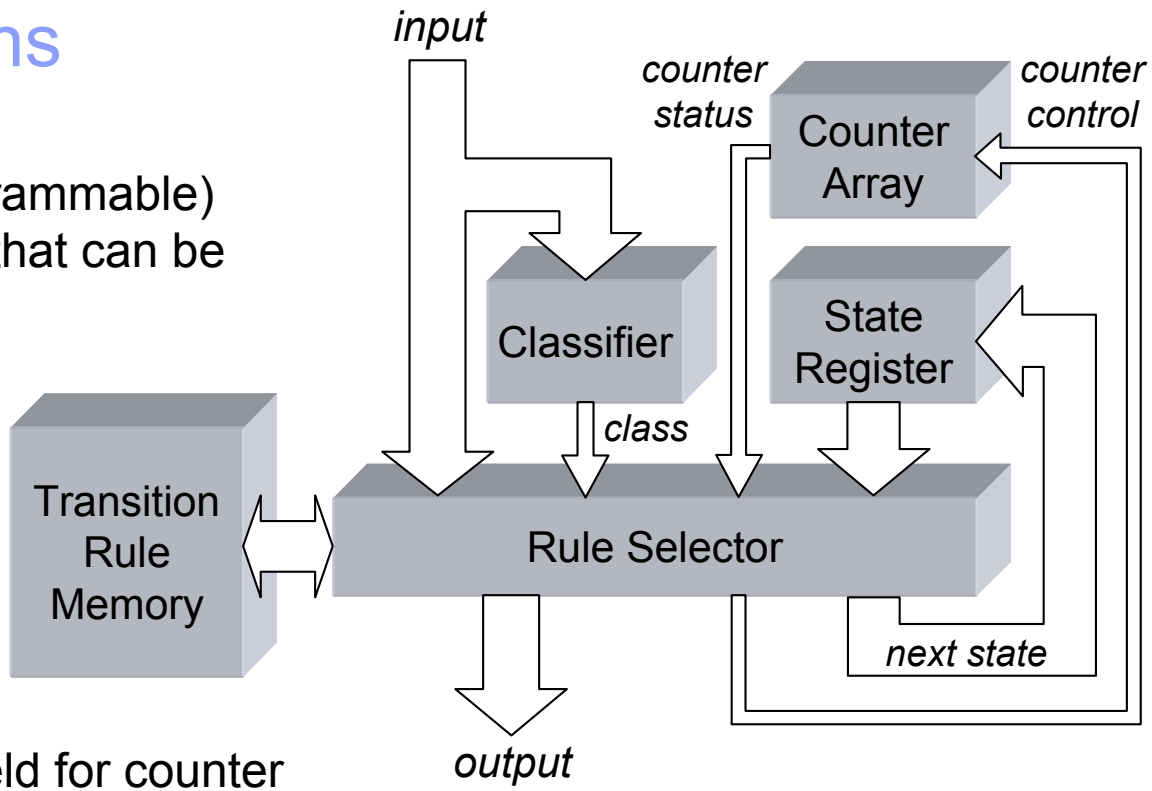
Phone: +41 44 724 8111
Fax: +41 44 724 8911

Backup

Regular Expressions

Classifier

- Classifies input into (programmable) sets of character classes that can be tested by transition rules
- Examples:
  - `\d` numeric
  - `\w` alphanumeric
  - `\s` whitespace



B-FSM engine

Counter support

- Transition rules include field for counter control (reset, load, increment, decrement)
- Condition field includes bits for testing counter status

☞ B-FSM support for wide input and output vectors

transition rule vector

Publications

BaRT

- J. van Lunteren and A.P.J. Engbersen, “Fast and scalable packet classification,” *IEEE Journal of Selected Areas in Communications*, vol. 21, no. 4, pp. 560-571, May 2003.

- J. van Lunteren, “Searching very large routing tables in wide embedded memory,” *Proceedings of IEEE GLOBECOM*, vol. 3, pp. 1615-1619, November 2001.

B-FSM

- J. van Lunteren et al., “XML accelerator engine,” *First Int. Workshop on High Performance XML Processing*, in conjunction with WWW2004, May 2004.

