Scalable DFA Compilation for High-Performance Regular-Expression Matching

Jan van Lunteren
1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
At the IBM Zurich Research Laboratory, we are investigating new types of programmable accelerators that are optimized for applications that run into performance and power problems on conventional processors.

Focus is on accelerating applications that operate (typically on-the-fly) on streams of data such as pattern matching (intrusion detection, Big data analytics), encryption, compression, and networking.

Recently: new concept is used to realize an “intelligent” memory system that
1) can be programmed to adapt its operation to the workload characteristics
2) supports power-efficient and high-performance near-memory computation
Agenda

1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
Regular-Expression Matching

- Scanning data streams to detect patterns (e.g., character sequences, words, signatures) that are specified using regular expressions
- Important for (signature-based) intrusion detection and analytics workloads

Design Objectives for Regular-Expression Acceleration

- A regular-expression scanner and compiler, supporting
  1) large sets of string (~millions) and regular-expression patterns (>10K)
  2) high scan rates (tens to hundreds of Gbit/s)
  3) parallel scans, multi-session support (millions of active sessions)

Design challenges

- extremely efficient use of available memory capacity and bandwidth
  - compact data structure
  - minimize number of memory accesses to process each input character
- exploit limited amount of parallelism in order to keep small session state
- fast compilation times, despite compression and other optimizations the compiler needs to support in order to meet above challenges
Regular-Expression Acceleration

Example
- Sample regular expression: \((a|b)*ab\)

Non-deterministic Finite Automaton (NFA)
- Several possible next states can exist for given state and input combination
- Pro: low storage requirements
- Con: high processing complexity

Deterministic Finite Automaton (DFA)
- At most one possible next state exists for each state and input combination
- Pro: low processing complexity (more suitable for hardware implementation)
- Con: high storage requirements - “state explosion problem”

(state \(S_2\) is accept state: match found)
State Explosion Problem

- Certain combinations of regular-expression patterns can cause a state explosion when mapped on a single DFA

- Example:
  \[
  \text{ab.*cd} \\
  \text{ef[^\n]*gh} \\
  \text{k.lm}
  \]

DFA with 48 states and 242 transitions
Regular-Expression Acceleration

State Explosion Problem

- Certain combinations of regular-expression patterns can cause a state explosion when mapped on a single DFA

- Example:
  ```
  ab.*cd
  ef[\n]*gh
  k..lm
  ```

DFA with 96 states and 508 transitions
State Explosion Problem

- Certain combinations of regular-expression patterns can cause a state explosion when mapped on a single DFA

- Example:
  
  ```
  ab.*cd
ef[^\n]*gh
k...lm
  ```

DFA with 192 states and 1038 transitions
State Explosion Problem

- Certain combinations of regular-expression patterns can cause a state explosion when mapped on a single DFA

- Example:
  \[
  \text{ab.*cd} \\
  \text{ef[^\n]*gh} \\
  \text{k.{n}lm}
  \]

<table>
<thead>
<tr>
<th>3rd pattern</th>
<th>DFA size</th>
<th>#states</th>
<th>#transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>k.1m</td>
<td></td>
<td>48</td>
<td>242</td>
</tr>
<tr>
<td>k..1m</td>
<td></td>
<td>96</td>
<td>508</td>
</tr>
<tr>
<td>k...1m</td>
<td></td>
<td>192</td>
<td>1038</td>
</tr>
<tr>
<td>k.....1m</td>
<td></td>
<td>384</td>
<td>2098</td>
</tr>
<tr>
<td>k.......1m</td>
<td></td>
<td>768</td>
<td>4218</td>
</tr>
<tr>
<td>k........1m</td>
<td></td>
<td>1536</td>
<td>8458</td>
</tr>
<tr>
<td>k..........1m</td>
<td></td>
<td>3072</td>
<td>16938</td>
</tr>
</tbody>
</table>
Handling the State Explosion Problem

- Separate regular expressions that result in the largest state explosions
  - intelligent distribution of patterns over multiple parallel DFAs
Handling the State Explosion Problem

- Separate regular expressions that result in the largest state explosions
  - intelligent distribution of patterns over multiple parallel DFAs

Example:
\[
\text{ab.*cd}
\text{ef[^\n]*gh}
\text{k.lm}
\]
Handling the State Explosion Problem

- Separate regular expressions that result in the largest state explosions
  - intelligent distribution of patterns over multiple parallel DFAs

- Split complex regular expressions into simpler expressions and determine if original regular expression is matched from matches on the simpler partial expressions
  - extension of DFAs with post processing unit that executes instructions attached to state transitions

Example:

```
ab.*cd
ef[^\n]*gh
k.lm
```

Diagram:

- Input
- DFA 0
- DFA 1
- DFA 2
- DFA n
- Post processing
- Match results

Note: The diagram illustrates the process of handling state explosions by distributing patterns over multiple parallel DFAs and using a post processing unit to execute instructions attached to state transitions.
Regular-Expression Acceleration

Handling the State Explosion Problem

- Separate regular expressions that result in the largest state explosions
  - intelligent distribution of patterns over multiple parallel DFAs

- Split complex regular expressions into simpler expressions and determine if original regular expression is matched from matches on the simpler partial expressions
  - extension of DFAs with post processing unit that executes instructions attached to state transitions

Example:

```
ab.*cd
ef[^\n]*gh
k.lm
```

Instructions:

- Check if cd is detected after ab

Match results:

```
input

DFA 0
DFA 1
DFA 2
DFA n

ef[^\n]*gh
k.lm

post processing

match results
```
Handling the State Explosion Problem

- Separate regular expressions that result in the largest state explosions
  - intelligent distribution of patterns over multiple parallel DFAs

- Split complex regular expressions into simpler expressions and determine if original regular expression is matched from matches on the simpler partial expressions
  - extension of DFAs with post processing unit that executes instructions attached to state transitions

Focus of remainder of presentation: DFA extended with instructions
1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
B-FSM Programmable State Machine

state transition diagram

rule state input $\rightarrow$ state prior.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Prior State</th>
<th>Input</th>
<th>New State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>*</td>
<td>*</td>
<td>S0</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>*</td>
<td>A</td>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>S1</td>
<td>B</td>
<td>S2</td>
<td>2</td>
</tr>
<tr>
<td>R3</td>
<td>S2</td>
<td>C</td>
<td>S3</td>
<td>2</td>
</tr>
</tbody>
</table>

described by transition rules

executed by B-FSM engine
B-FSM Programmable State Machine

- HW-based programmable state machine: B-FSM
  - deterministic rate of one transition per clock cycle @ 2-3 GHz
  - powerful state transition specification using input conditions and priorities
    - exact match, character classes, case insensitive, ternary match, negation
  - programmable by loading compiled data structure into transition-rule memory

State transition diagram

<table>
<thead>
<tr>
<th>Rule</th>
<th>State Input</th>
<th>State Prior</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>*</td>
<td>S0</td>
</tr>
<tr>
<td>R1</td>
<td>*</td>
<td>S1</td>
</tr>
<tr>
<td>R2</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>R3</td>
<td>S2</td>
<td>S3</td>
</tr>
</tbody>
</table>

State Register

Rule Selector

Transition Rule Memory

B-FSM engine

described by transition rules

executed by B-FSM engine

input

output
B-FSM Programmable State Machine

- HW-based programmable state machine: B-FSM
  - deterministic rate of one transition per clock cycle @ 2-3 GHz
  - powerful state transition specification using input conditions and priorities
    - exact match, character classes, case insensitive, ternary match, negation
  - programmable by loading compiled data structure into transition-rule memory

<table>
<thead>
<tr>
<th>Rule</th>
<th>State</th>
<th>Input</th>
<th>Prior</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>*</td>
<td>*</td>
<td>S0 0</td>
</tr>
<tr>
<td>R1</td>
<td>*</td>
<td>A</td>
<td>S1 1</td>
</tr>
<tr>
<td>R2</td>
<td>S1</td>
<td>B</td>
<td>S2 2</td>
</tr>
<tr>
<td>R3</td>
<td>S2</td>
<td>C</td>
<td>S3 2</td>
</tr>
</tbody>
</table>
### B-FSM Programmable State Machine

- **Sample regular expressions:**
  
  \[ a[bB]^*c \]
  
  \[ a[bB][^c][0-8]^*9 \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>State Prior</th>
<th>Input</th>
<th>Result State</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₀</td>
<td>*</td>
<td>*</td>
<td>S₀</td>
</tr>
<tr>
<td>R₁</td>
<td>*</td>
<td>a</td>
<td>S₁</td>
</tr>
<tr>
<td>R₂</td>
<td>S₁</td>
<td>[bB]</td>
<td>S₂</td>
</tr>
<tr>
<td>R₃</td>
<td>S₁</td>
<td>c</td>
<td>S₅</td>
</tr>
<tr>
<td>R₄</td>
<td>S₂</td>
<td>[^abBc]</td>
<td>S₇</td>
</tr>
<tr>
<td>R₅</td>
<td>S₂</td>
<td>a</td>
<td>S₆</td>
</tr>
<tr>
<td>R₆</td>
<td>S₂</td>
<td>[bB]</td>
<td>S₃</td>
</tr>
<tr>
<td>R₇</td>
<td>S₂</td>
<td>c</td>
<td>S₅</td>
</tr>
<tr>
<td>R₈</td>
<td>S₃</td>
<td>[0-9]</td>
<td>S₇</td>
</tr>
<tr>
<td>R₉</td>
<td>S₃</td>
<td>9</td>
<td>S₈</td>
</tr>
<tr>
<td>R₁₀</td>
<td>S₃</td>
<td>[bB]</td>
<td>S₄</td>
</tr>
<tr>
<td>R₁₁</td>
<td>S₃</td>
<td>c</td>
<td>S₅</td>
</tr>
<tr>
<td>R₁₂</td>
<td>S₄</td>
<td>[bB]</td>
<td>S₄</td>
</tr>
<tr>
<td>R₁₃</td>
<td>S₄</td>
<td>c</td>
<td>S₅</td>
</tr>
<tr>
<td>R₁₄</td>
<td>S₆</td>
<td>[0-9]</td>
<td>S₇</td>
</tr>
<tr>
<td>R₁₅</td>
<td>S₆</td>
<td>9</td>
<td>S₈</td>
</tr>
<tr>
<td>R₁₆</td>
<td>S₆</td>
<td>[bB]</td>
<td>S₂</td>
</tr>
<tr>
<td>R₁₇</td>
<td>S₆</td>
<td>c</td>
<td>S₅</td>
</tr>
<tr>
<td>R₁₈</td>
<td>S₇</td>
<td>[0-9]</td>
<td>S₇</td>
</tr>
<tr>
<td>R₁₉</td>
<td>S₇</td>
<td>9</td>
<td>S₈</td>
</tr>
</tbody>
</table>
B-FSM Programmable State Machine

- **input**
- **State Reg.**
- **Table Reg.**
- **Mask Reg.**

- **Character Classifier**
- **Default Rule Table**

- **Rule Selector**

- **Address Generator**

- **Transition Rule Memory**
Step 1: *generate address based on current state and input values*
Step 2: read one line containing $P$ transition rules from the transition-rule memory
Step 2: read one line containing P transition rules from the transition-rule memory
Step 2: read one line containing $P$ transition rules from the transition-rule memory
Step 3: select transition by testing current state and input against P transition rules
B-FSM Programmable State Machine

Step 4: update state and other registers based on selected transition rule
Default rule table

<table>
<thead>
<tr>
<th>Rule</th>
<th>State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_0</td>
<td>*</td>
<td>*</td>
<td>S_0 0</td>
</tr>
<tr>
<td>R_1</td>
<td>*</td>
<td>a</td>
<td>S_1 1</td>
</tr>
<tr>
<td>R_2</td>
<td>S_1</td>
<td>[bB]</td>
<td>S_2 2</td>
</tr>
<tr>
<td>R_3</td>
<td>S_1</td>
<td>c</td>
<td>S_5 2</td>
</tr>
<tr>
<td>R_4</td>
<td>S_2</td>
<td>[^abBc]</td>
<td>S_7 2</td>
</tr>
<tr>
<td>R_5</td>
<td>S_2</td>
<td>a</td>
<td>S_6 2</td>
</tr>
<tr>
<td>R_6</td>
<td>S_2</td>
<td>[bB]</td>
<td>S_3 2</td>
</tr>
<tr>
<td>R_7</td>
<td>S_2</td>
<td>c</td>
<td>S_5 2</td>
</tr>
<tr>
<td>R_8</td>
<td>S_3</td>
<td>[0-9]</td>
<td>S_7 2</td>
</tr>
</tbody>
</table>

...
Character classifier

rule state input → state prior.

R₀ * * → S₀ 0
R₁ * a → S₁ 1
R₂ S₁ [bB] → S₂ 2
R₃ S₁ c → S₅ 2
R₄ S₂ [^abBc] → S₇ 2
R₅ S₂ a → S₆ 2
R₆ S₂ [bB] → S₃ 2
R₇ S₂ c → S₅ 2
R₈ S₃ [0-9] → S₇ 2

...
### Transition-rule vector

<table>
<thead>
<tr>
<th>Rule</th>
<th>State</th>
<th>Input/Class</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₀</td>
<td>*</td>
<td>*</td>
<td>S₀</td>
</tr>
<tr>
<td>R₁</td>
<td>*</td>
<td>a</td>
<td>S₁</td>
</tr>
<tr>
<td>R₂</td>
<td>S₁</td>
<td>[bB]</td>
<td>S₂</td>
</tr>
<tr>
<td>R₃</td>
<td>S₁</td>
<td>c</td>
<td>S₅</td>
</tr>
<tr>
<td>R₄</td>
<td>S₂</td>
<td>[^abBc]</td>
<td>S₇</td>
</tr>
<tr>
<td>R₅</td>
<td>S₂</td>
<td>a</td>
<td>S₆</td>
</tr>
<tr>
<td>R₆</td>
<td>S₂</td>
<td>[bB]</td>
<td>S₃</td>
</tr>
<tr>
<td>R₇</td>
<td>S₂</td>
<td>c</td>
<td>S₅</td>
</tr>
<tr>
<td>R₈</td>
<td>S₃</td>
<td>[0–9]</td>
<td>S₇</td>
</tr>
</tbody>
</table>

...
B-FSM Programmable State Machine

- Transition-rule vector

```
rule state input → state prior.
R₀  *   *   → S₀  0
R₁  *   a   → S₁  1
R₂  S₁  [bB] → S₂  2
R₃  S₁  c   → S₅  2
R₄  S₂  [^abBc] → S₇  2
R₅  S₂  a   → S₆  2
R₆  S₂  [bB] → S₃  
R₇  S₂  c   
R₈  S₃  [0–9] → S₇  
```

**Example 3-bit type encoding**

- 000b case-sensitive exact match
- 001b case-insensitive match
- 010b class match A
- 011b class match B
- 100b negated case-sensitive exact match
- 101b negated case-insensitive match
- 110b negated class match A
- 111b negated class match B

**Test part**

- type: 3 bits
- state: 3 bits
- input/class: 8 bits

**Result part**

- next state: 11 bits
- table address: n bits
- mask: 8 bits
- result: 1 bit
B-FSM Programmable State Machine

- Address generation
B-FSM Programmable State Machine

- Address generation
B-FSM Programmable State Machine

- **Address generation**
  - states and transitions mapped on clusters
  - each cluster stored as compressed transition-rule table

![State transition diagram (transition rules)]
B-FSM Programmable State Machine

- Address generation
- states and transitions mapped on clusters
- each cluster stored as compressed transition-rule table
- Table register points to transition-rule table that stores transition rules of current state

state transition diagram (transition rules)
B-FSM Programmable State Machine

- Address generation
  - states and transitions mapped on clusters
  - each cluster stored as compressed transition-rule table
  - Table register points to transition-rule table that stores transition rules of current state

→ Hash function:

\[ \text{index} = (\text{state'} \, \text{and} \, \text{not} \, \text{mask}) \, \text{or} \, (\text{input} \, \text{and} \, \text{mask}) \]

- \text{and, or, not}: bitwise operators
- \text{state}', mask, input, index: 8 bits
- \text{state}' is least significant part of state
Agenda

1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
B-FSM Compiler Tasks

1) state clustering
   – select for each state a transition-rule table on which its transition rules will be mapped

2) transition-rule table construction
   – perform state encoding
   – determine for each state a mask (hash function selection)

- special features
  – integration of instructions
  – range of storage optimizations (ext. address support, common rules)

=> Objectives
  – very compact data structure
  – fast compilation times
B-FSM Compiler

- States with up to $P$ transition rules can be mapped on a single line
  - mask = 00000000b
  - index = state'

- Hash function
  
  \[ \text{index} = (\text{state'} \ \text{and not} \ \text{mask}) \ \text{or} \ (\text{input} \ \text{and} \ \text{mask}) \]

\[
\begin{array}{cccc}
\text{mask} &=& 00000000b \\
S_1 &=& 000 00000000b \\
\text{input} & \text{ascii} & \text{index} & \text{rule} \\
B & 01000010b & 00h & R_2 \\
b & 01100010b & 00h & R_2 \\
c & 01100011b & 00h & R_3 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{transition-rule table} \\
0h & \text{R}_2 & \text{R}_3 \\
FFh
\end{array}
\]

\[
\begin{array}{cc}
\text{rule} & \text{state} \ \text{input} \ \rightarrow \ \text{state} \ \text{prior.} \\
\text{R}_2 & S_1 & [\text{bB}] \rightarrow S_2 & 2 \\
\text{R}_3 & S_1 & c \rightarrow S_5 & 2 \\
\end{array}
\]
B-FSM Compiler

- States with up to \( P \) transition rules can be mapped on a single line
  - \( \text{mask} = 00000000b \)
  - \( \text{index} = \text{state}' \)

- Hash function
  
  \[
  \text{index} = (\text{state'} \text{ and not mask}) \text{ or } (\text{input and mask})
  \]

\[
\begin{align*}
\text{mask} & = 00000000b \\
S_1 & = 000 00110100b \\
\text{input} & \quad \text{ascii} \quad \text{index} \quad \text{rule} \\
B & \quad 01000010b \quad 34h \quad R_2 \\
b & \quad 01100010b \quad 34h \quad R_2 \\
c & \quad 01100011b \quad 34h \quad R_3
\end{align*}
\]

\[
\begin{array}{c|c|c}
\text{rule} & \text{state} & \text{input} \\
\hline
R_2 & S_1 & [bB] \\
R_3 & S_1 & c \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{state prior.} & \\
S_2 & 2 \\
S_5 & 2 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{rule} & \text{state} & \text{prior} \\
\hline
R_2 & S_1 \rightarrow S_2 & 2 \\
R_3 & S_1 \rightarrow S_5 & 2 \\
\end{array}
\]

transition-rule table
States with up to $P$ transition rules can be mapped on a single line
- mask = 0000000b
- index = state’

Hash function

$$index = (state’ \text{ and not } mask) \text{ or } (input \text{ and } mask)$$

<table>
<thead>
<tr>
<th>mask= 00000000b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$=00 01111000b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>ascii</th>
<th>index</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>01000010b</td>
<td>78h</td>
<td>$R_2$</td>
</tr>
<tr>
<td>b</td>
<td>01100010b</td>
<td>78h</td>
<td>$R_2$</td>
</tr>
<tr>
<td>c</td>
<td>01100011b</td>
<td>78h</td>
<td>$R_3$</td>
</tr>
</tbody>
</table>

rule state input $\rightarrow$ state prior.

- $R_2$ $S_1$ [bB] $\rightarrow$ $S_2$ 2
- $R_3$ $S_1$ c $\rightarrow$ $S_5$ 2

transition-rule table
B-FSM Compiler

- States with up to $P$ transition rules can be mapped on a single line
  - mask = 00000000b
  - index = state’

- Hash function
  $index = (state’ \text{ and not } mask) \text{ or } (input \text{ and } mask)$

<table>
<thead>
<tr>
<th>rule</th>
<th>state</th>
<th>input</th>
<th>index</th>
<th>state prior.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_2$</td>
<td>$S_1$</td>
<td>[bB]</td>
<td></td>
<td>$S_2$</td>
</tr>
<tr>
<td>$R_3$</td>
<td>$S_1$</td>
<td>c</td>
<td></td>
<td>$S_5$</td>
</tr>
</tbody>
</table>

- transition-rule vectors can be “moved” through transition-rule table by varying the state encoding

\[
\begin{array}{|c|c|c|c|}
\hline
\text{FFh} & \text{78h} & \text{R}_2 & \text{R}_3 \\
\hline
\text{On} & \text{rule state input} & \rightarrow & \text{state prior.} \\
\hline
\end{array}
\]
B-FSM Compiler

- States with more than $P$ transition rules need to be distributed over multiple lines
  - requires non-zero mask that maps at most $P$ transitions rules on each line

- Hash function
  
  $\text{index} = (\text{state\ ' and not mask}) \text{ or } (\text{input and mask})$

- Hash function

$$\begin{align*}
\text{mask} &= 00000001_{\text{b}} \\
S_3 &= 000000000_{\text{b}}
\end{align*}$$

- Hash function

<table>
<thead>
<tr>
<th>input</th>
<th>ascii</th>
<th>index</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00110000_{\text{b}}</td>
<td>00h</td>
<td>R_8</td>
</tr>
<tr>
<td>1</td>
<td>00110001_{\text{b}}</td>
<td>01h</td>
<td>R_8</td>
</tr>
<tr>
<td>2</td>
<td>00110010_{\text{b}}</td>
<td>00h</td>
<td>R_8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00111000_{\text{b}}</td>
<td>00h</td>
<td>R_8</td>
</tr>
<tr>
<td>9</td>
<td>00111001_{\text{b}}</td>
<td>01h</td>
<td>R_9</td>
</tr>
<tr>
<td>B</td>
<td>01000001_{\text{b}}</td>
<td>00h</td>
<td>R_{10}</td>
</tr>
<tr>
<td>b</td>
<td>01100010_{\text{b}}</td>
<td>00h</td>
<td>R_{10}</td>
</tr>
<tr>
<td>c</td>
<td>01100011_{\text{b}}</td>
<td>01h</td>
<td>R_{11}</td>
</tr>
</tbody>
</table>

- Hash function

**Transition-rule table**

<table>
<thead>
<tr>
<th>FFh</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Duplicates**
B-FSM Compiler

- States with more than $P$ transition rules need to be distributed over multiple lines
  - requires non-zero mask that maps at most $P$ transitions rules on each line

- Hash function
  
  \[
  \text{index} = (\text{state'} \text{ and not mask}) \text{ or } (\text{input and mask})
  \]

<table>
<thead>
<tr>
<th>mask</th>
<th>00000111b</th>
</tr>
</thead>
</table>

| S_3  | 00000000b |

<table>
<thead>
<tr>
<th>input</th>
<th>ascii</th>
<th>index</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00110000b</td>
<td>00h</td>
<td>R_8</td>
</tr>
<tr>
<td>1</td>
<td>00110001b</td>
<td>01h</td>
<td>R_8</td>
</tr>
<tr>
<td>2</td>
<td>00110010b</td>
<td>02h</td>
<td>R_8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00111000b</td>
<td>00h</td>
<td>R_8</td>
</tr>
<tr>
<td>9</td>
<td>00111001b</td>
<td>01h</td>
<td>R_9</td>
</tr>
<tr>
<td>B</td>
<td>01000100b</td>
<td>02h</td>
<td>R_{10}</td>
</tr>
<tr>
<td>b</td>
<td>01100100b</td>
<td>02h</td>
<td>R_{10}</td>
</tr>
<tr>
<td>c</td>
<td>01100111b</td>
<td>03h</td>
<td>R_{11}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FFh</th>
<th>duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>7h</td>
<td>R_8</td>
</tr>
<tr>
<td>6h</td>
<td>R_8</td>
</tr>
<tr>
<td>5h</td>
<td>R_8</td>
</tr>
<tr>
<td>4h</td>
<td>R_8</td>
</tr>
<tr>
<td>3h</td>
<td>R_8</td>
</tr>
<tr>
<td>2h</td>
<td>R_8</td>
</tr>
<tr>
<td>1h</td>
<td>R_9</td>
</tr>
<tr>
<td>0h</td>
<td>R_8</td>
</tr>
</tbody>
</table>

transition-rule table

\[
\text{rule state input} \rightarrow \text{state prior.}
\]

<table>
<thead>
<tr>
<th>R_8</th>
<th>S_3</th>
<th>[0-9]</th>
<th>\rightarrow</th>
<th>S_7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_9</td>
<td>S_3</td>
<td>9</td>
<td>\rightarrow</td>
<td>S_8</td>
<td>3</td>
</tr>
<tr>
<td>R_{10}</td>
<td>S_3</td>
<td>[bB]</td>
<td>\rightarrow</td>
<td>S_4</td>
<td>2</td>
</tr>
<tr>
<td>R_{11}</td>
<td>S_3</td>
<td>c</td>
<td>\rightarrow</td>
<td>S_5</td>
<td>2</td>
</tr>
</tbody>
</table>
B-FSM Compiler

- States with more than $P$ transition rules need to be distributed over multiple lines
  - requires non-zero mask that maps at most $P$ transitions rules on each line

- Hash function
  
  $index = (state \text{' and not mask}) \text{ or } (input \text{ and mask})$

<table>
<thead>
<tr>
<th>mask=</th>
<th>00010000b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_3=$</td>
<td>000 0000000b</td>
</tr>
<tr>
<td>input</td>
<td>ascii</td>
</tr>
<tr>
<td>0</td>
<td>00110000b</td>
</tr>
<tr>
<td>1</td>
<td>00110001b</td>
</tr>
<tr>
<td>2</td>
<td>00110010b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>00111000b</td>
</tr>
<tr>
<td>9</td>
<td>00111001b</td>
</tr>
<tr>
<td>B</td>
<td>01000010b</td>
</tr>
<tr>
<td>b</td>
<td>01100010b</td>
</tr>
<tr>
<td>c</td>
<td>01100011b</td>
</tr>
</tbody>
</table>

rule state input  →  state prior.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_8$</td>
<td>$S_3$</td>
<td>[0–9]</td>
<td>$S_7$ 2</td>
</tr>
<tr>
<td>$R_9$</td>
<td>$S_3$</td>
<td>9</td>
<td>$S_8$ 3</td>
</tr>
<tr>
<td>$R_{10}$</td>
<td>$S_3$</td>
<td>[bB]</td>
<td>$S_4$ 2</td>
</tr>
<tr>
<td>$R_{11}$</td>
<td>$S_3$</td>
<td>c</td>
<td>$S_5$ 2</td>
</tr>
</tbody>
</table>

transition-rule table
B-FSM Compiler

- States with more than $P$ transition rules need to be distributed over multiple lines
  - requires non-zero mask that maps at most $P$ transitions rules on each line

- Hash function

  \[
  \text{index} = (\text{'state'} \text{ and } \neg \text{mask}) \text{ or } \text{(input and mask)}
  \]

> mask = 00010000b

\[
S_3 = 000 0100001b
\]

<table>
<thead>
<tr>
<th>input</th>
<th>ascii</th>
<th>index</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00110000b</td>
<td>51h</td>
<td>$R_8$</td>
</tr>
<tr>
<td>1</td>
<td>00110001b</td>
<td>51h</td>
<td>$R_8$</td>
</tr>
<tr>
<td>2</td>
<td>00110010b</td>
<td>51h</td>
<td>$R_8$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00111000b</td>
<td>51h</td>
<td>$R_8$</td>
</tr>
<tr>
<td>9</td>
<td>00111001b</td>
<td>51h</td>
<td>$R_9$</td>
</tr>
<tr>
<td>B</td>
<td>01000010b</td>
<td>41h</td>
<td>$R_{10}$</td>
</tr>
<tr>
<td>b</td>
<td>01100010b</td>
<td>41h</td>
<td>$R_{10}$</td>
</tr>
<tr>
<td>c</td>
<td>01100011b</td>
<td>41h</td>
<td>$R_{11}$</td>
</tr>
</tbody>
</table>

**rule state input \rightarrow state prior.**

| $R_8$ | $S_3$ | [0-9] | \rightarrow | $S_7$ | 2 |
| $R_9$ | $S_3$ | 9     | \rightarrow  | $S_8$ | 3 |
| $R_{10}$ | $S_3$ | [bB]  | \rightarrow  | $S_4$ | 2 |
| $R_{11}$ | $S_3$ | c     | \rightarrow  | $S_5$ | 2 |

**transition-rule table**
States with more than $P$ transition rules need to be distributed over multiple lines
  - requires non-zero mask that maps at most $P$ transitions rules on each line

Hash function

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

<table>
<thead>
<tr>
<th>rule state input</th>
<th>→</th>
<th>state prior.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_8$</td>
<td>$S_3$</td>
<td>[0-9] → $S_7$</td>
</tr>
<tr>
<td>$R_9$</td>
<td>$S_3$</td>
<td>9 → $S_8$</td>
</tr>
<tr>
<td>$R_{10}$</td>
<td>$S_3$</td>
<td>[bB] → $S_4$</td>
</tr>
<tr>
<td>$R_{11}$</td>
<td>$S_3$</td>
<td>c → $S_5$</td>
</tr>
</tbody>
</table>

mask= 00010000b

$S_3=0001000011b$

<table>
<thead>
<tr>
<th>input ascii index rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00100000b 97h $R_8$</td>
</tr>
<tr>
<td>1 0010001b 97h $R_8$</td>
</tr>
<tr>
<td>2 0010010b 97h $R_8$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>8 00111000b 97h $R_8$</td>
</tr>
<tr>
<td>9 00111001b 97h $R_9$</td>
</tr>
<tr>
<td>B 01000010b 87h $R_{10}$</td>
</tr>
<tr>
<td>b 01100010b 87h $R_{10}$</td>
</tr>
<tr>
<td>c 01100011b 87h $R_{11}$</td>
</tr>
</tbody>
</table>

Transition-rule table
- States with more than \( P \) transition rules need to be distributed over multiple lines
  - requires non-zero mask that maps at most \( P \) transitions rules on each line

- Hash function

\[
index = (\text{state'} \text{ and not mask}) \text{ or} \ (\text{input and mask})
\]

<table>
<thead>
<tr>
<th>Rule</th>
<th>State</th>
<th>Input</th>
<th>State Prior</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_8 )</td>
<td>( S_3 )</td>
<td>[0-9]</td>
<td>( S_7 ) 2</td>
</tr>
<tr>
<td>( R_9 )</td>
<td>( S_3 )</td>
<td>9</td>
<td>( S_8 ) 3</td>
</tr>
<tr>
<td>( R_{10} )</td>
<td>( S_3 )</td>
<td>[bB]</td>
<td>( S_4 ) 2</td>
</tr>
<tr>
<td>( R_{11} )</td>
<td>( S_3 )</td>
<td>3</td>
<td>( S_5 ) 2</td>
</tr>
</tbody>
</table>

\[ FFh \]

\[ 97h \]

- mask determines “shape” of mapped transition-rule block, including the number of duplicates
- state encoding can be used to “move” mapped transition-rule block through transition-rule table without affecting mapped “block shape”
Maping multiple states/transition rules on a transition-rule table
- compact data structure requires
  - minimize duplicates
  - maximize fill rate of transition-rule table

<table>
<thead>
<tr>
<th>state</th>
<th>vector</th>
<th>mask</th>
<th>rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>000</td>
<td>0000000000b</td>
<td>01h R₂, R₃</td>
</tr>
<tr>
<td>S₂</td>
<td>000</td>
<td>0000000100b</td>
<td>01h R₄-R₇</td>
</tr>
<tr>
<td>S₃</td>
<td>001</td>
<td>0000000000b</td>
<td>10h R₈-R₁₁</td>
</tr>
<tr>
<td>S₄</td>
<td>000</td>
<td>0001000000b</td>
<td>01h R₁₂-R₁₃</td>
</tr>
<tr>
<td>S₅</td>
<td>001</td>
<td>0000001000b</td>
<td>10h R₁₄-R₁₇</td>
</tr>
<tr>
<td>S₆</td>
<td>000</td>
<td>0000010000b</td>
<td>00h R₁₈-R₁₉</td>
</tr>
<tr>
<td>S₇</td>
<td>111</td>
<td>0000000000b</td>
<td>00h</td>
</tr>
<tr>
<td>S₈</td>
<td>111</td>
<td>0000000001b</td>
<td>00h</td>
</tr>
</tbody>
</table>

transition-rule table
B-FSM Compiler

- Mapping multiple states/transition rules
  - compact data structure requires
    - minimize duplicates
    - maximize fill rate of transition-rule table
    - try to do this such that all “rule blocks” nicely fit in the transition-rule table

similar to Tetris™ game

- select mask for each state to create and modify shape of mapped “rule blocks”
- use state encoding to move “rule blocks” at different table offsets

<table>
<thead>
<tr>
<th>state</th>
<th>vector</th>
<th>mask</th>
<th>rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td>000</td>
<td>10000000000b</td>
<td>01h</td>
</tr>
<tr>
<td>S_2</td>
<td>000</td>
<td>00000000010b</td>
<td>01h</td>
</tr>
<tr>
<td>S_3</td>
<td>001</td>
<td>0000000000b</td>
<td>10h</td>
</tr>
<tr>
<td>S_4</td>
<td>000</td>
<td>0000010000b</td>
<td>01h</td>
</tr>
<tr>
<td>S_5</td>
<td>001</td>
<td>000000001b</td>
<td>10h</td>
</tr>
<tr>
<td>S_6</td>
<td>000</td>
<td>000000100b</td>
<td>00h</td>
</tr>
</tbody>
</table>

M
apping multiple states/transition rules

- compact data structure requires
  - minimize duplicates
  - maximize fill rate of transition-rule table
  - try to do this such that all “rule blocks” nicely fit in the transition-rule table

similar to Tetris™ game

- select mask for each state to create and modify shape of mapped “rule blocks”
- use state encoding to move “rule blocks” at different table offsets
- try to do this such that all “rule blocks” nicely fit in the transition-rule table
B-FSM Compiler

Special features

- Integration of instructions
  - transition rule in transition-rule table entry is replaced by instruction vector
  - line type defines existence of instruction and to which transition rule it is attached
  - B-FSM compiler selects masks such that at most $P-1$ transition rules are mapped on lines with instructions
  ➔ instructions only consume storage when actually used

<table>
<thead>
<tr>
<th>line type</th>
<th>transition rule 0</th>
<th>transition rule 1</th>
<th>transition rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

transition-rule table entry with $P=3$ transition rules

<table>
<thead>
<tr>
<th>line type</th>
<th>transition rule 0</th>
<th>transition rule 1</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

transition-rule table entry with $P-1=2$ transition rules and one instruction vector
B-FSM Compiler

B-FSM Compiler

- Not possible to test all possible combinations of state clustering (table address), state encoding (state vector) and hash selection (mask vector) that
  - minimize storage requirements (minimum duplicates, maximum table fill rate)
  - meet all constraints
- Instead the compiler exploits heuristics, parallel processing and efficient bit-vector based data structures that enable fast processing using modern SIMD instructions
- High-level approach
  1. states are processed by decreasing order of their number of transition rules
  2. for each state, masks are selected resulting in valid mappings with minimum duplicates, and corresponding mapped rule-blocks are created
  3. each rule-block is moved through all possible offsets in the current transition-rule table by varying state encoding until a state encoding is found for which no collision occurs (i.e., no transition-rule table entry contains more than $P$ rules)
  4. if no rule-block can be mapped on current transition-rule table, then a new transition-rule table is created (clustering), and the process continues

-can be effectively parallelized and accelerated using a range of support structures and SIMD instructions
1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
Experimental Results

- Storage requirements – regular-expression patterns

- Roughly linear increase of storage size with DFA size (#transition rules)
- Larger values of $P$ (#transition rules per line) result in a more compact structure
### Experimental Results

- **Table fill ratio – regular-expression patterns**

> Larger values of $P$ (#transition rules per line) typically result in a lower number of duplicates
Experimental Results

- Compilation speed
  - ranged from about **25K to over 100K transition rules per second** on a system with Xeon™ E5-2680 processors running at 2.7 GHz
  - specific optimizations for string DFAs resulted in compilation rates of over **1M transition rules per second**
<table>
<thead>
<tr>
<th>Technology</th>
<th>IBM 45nm SOI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Frequency</td>
<td>2.3GHz @ 0.97V (Worst Case Process)</td>
</tr>
<tr>
<td>Chip size</td>
<td>428 mm2 (including kerf)</td>
</tr>
<tr>
<td>Chip Power (4-AT node)</td>
<td>65W @ 2.0GHz, 0.85V Max Single Chip</td>
</tr>
<tr>
<td>Chip Power (1-AT node)</td>
<td>20W @ 1.4GHz, 0.77V Min Single Chip</td>
</tr>
<tr>
<td>Main Voltage (VDD)</td>
<td>0.7V to 1.1V</td>
</tr>
<tr>
<td>Metal Layers</td>
<td>11 Cu (3-1x, 2-1.3x, 3-2x, 1-4x, 2-10x)</td>
</tr>
<tr>
<td>Latch Count</td>
<td>3.2M</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>1.43B</td>
</tr>
<tr>
<td>A2 Cores / Threads</td>
<td>16 / 64</td>
</tr>
<tr>
<td>L1 I &amp; D Cache</td>
<td>16 x (16KB + 16KB) SRAM</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>4 x 2MB eDRAM</td>
</tr>
<tr>
<td>Hardware Accelerators</td>
<td>Crypto, Compression, RegX, XML</td>
</tr>
<tr>
<td>Intelligent Network Interfaces</td>
<td>Host Ethernet Adapter/Packet Processor</td>
</tr>
<tr>
<td></td>
<td>2 Modes: Endpoint &amp; Network</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>2x DDR3 controllers</td>
</tr>
<tr>
<td></td>
<td>4 Channels @ 800-1600MHz</td>
</tr>
<tr>
<td>System I/O Bandwidth</td>
<td>4x 10G Ethernet, 2x PCI Gen2</td>
</tr>
<tr>
<td>Chip-to-Chip Bandwidth</td>
<td>3 Links, 20GB/s per link</td>
</tr>
<tr>
<td>Chip Scaling</td>
<td>4 Chip SMP</td>
</tr>
<tr>
<td>Package</td>
<td>50mm FCPBGA (4 or 6 layers)</td>
</tr>
</tbody>
</table>

Source: Johnson et al., "A wire-speed power™ processor: 2.3GHz 45nm SOI with 16 cores and 64 threads," ISSCC 2010.
1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
Comparison with Related work

B-FSM-based compression

- applies a combination of indexing (input bits related to set mask bits) and parallel testing ($P$ transition rules per memory line) in an adaptive fashion
  - very effective for compressing densely respectively sparsely populated portions of DFA structures
  - supports states with any number of transitions
  - supports integration of instructions
- powerful state and input conditions at transition level combined with priorities
  - exact match, character classes, case insensitive, ternary match, negation
- single memory access per lookup (deterministic performance)
- extremely simple logic enabling efficient hardware implementations operating at high clock frequencies (2-3 GHz)
Comparison with Related work

B-FSM vs. linear and bitmapped encoding

- B-FSM clearly outperforms these types of schemes, which require
  - *multiple accesses* to process each input (testing against stored values)
  - *at least 3-4 bytes* to store next state information for each input value

  ➔ B-FSM uses a single access per input
  ➔ B-FSM needs substantially less storage

B-FSM vs. indirect addressing (CD²FA scheme)

- B-FSM and CD²FA both use a single access per input
- CD²FA can only be applied to states with a *limited number of transitions*
  - limits of two and five transitions were reported for 32-bit and 64-bit state vectors respectively, which correspond to an average storage per input value (transition) of about \(32/2=16\) bits and \(64/5\approx13\) bits respectively
  - larger numbers of transitions per state will result in higher average #bits/input

  ➔ B-FSM can use as little as \(~11\) bits per input value (depends on patterns)
  ➔ construction of CD²FA structure much more complicated, requiring longer compilation times and harder to scale to larger DFAs than B-FSM
Agenda

1. Broader Research Scope and Context
2. Regular-Expression Acceleration
3. B-FSM Programmable State Machine
4. B-FSM Compiler
5. Experimental Results
6. Comparison with Related Work
7. Conclusions
Presented B-FSM compilation scheme

- achieves (one of) the most compact DFA structure(s) existing today
  - which can be executed by a simple programmable hardware engine to process data at rates of multiple tens to hundreds of Gbit/s
  - supports integration of instructions
- can scale to DFAs with multiple tens of millions of transitions while maintaining an approximately linear growth of the storage requirements as a function of the DFA size
- achieves fast compilation times by exploiting a range of heuristics combined with implementation optimizations
- is used for a range of programmable accelerator engines beyond regular-expression matching
B-FSM is also used in a new project on Near-Memory Acceleration.

Three *PhD positions* available as part of European Union Horizon 2020 / Marie Curie ITN-EID program NeMeCo which is aimed at developing power-efficient HPC systems for Big-data processing based on the exploitation of near-memory computing.

- topics:
  - run-time optimization
  - compiler technologies
  - near-memory accelerator architecture