Automation Framework for Large-Scale Regular Expression Matching on FPGA

Thilan Ganegedara, Yi-Hua E. Yang, Viktor K. Prasanna

Ming-Hsieh Department of Electrical Engineering
University of Southern California
• Regular Expression Matching (REM)
  • Applications & requirements

• Existing Solutions for REM
  • Approaches & platforms
  • REM circuits & architecture

• Automation Framework for REM
  • Goals
  • High-level approach
  • Frontend processing
  • Backend processing

• Results
  • Performance
  • Resource utilization
Deep Packet Inspection (DPI)
- Continuously scan packet payload for malicious patterns
- String matching: “|BA 49 FE FF FF F7 D2 B9 BF FF FF FF F7 D1|”
- Regular expression: /\^(ymsg|ypns|yho0).{0-7}[lwt]\.*\xc0\x80/ 

REM for DPI challenges
- Overlapping matches on 10 Gbps input streams
- Over 2k unique patterns or 50k characters
- Complex regex with nested unions & closures
- Arbitrary character classes
- Automated solution to support continual regex updates

REM in popular security systems
- SNORT
- Bro IDS
- Clam AntiVirus
- Cisco Security Appliance
- Citrix Application Firewall
Existing Solutions for REM

- **Software Solutions**
  - POSIX/Unix (grep, egrep, sed, awk)
  - Libraries (PCRE, GnuLib)
  - Features:
    - Optimized for sequential matching
    - Support for backtracking

- **Hardware Solutions**
  - McNaughton-Yamada construction for NFA construction
  - Multi-character matching per cycle [Yamagaki et al. FPL’08]
  - Platforms:
    - ASIC [Floyd & Ullman JACM’82]
    - FPGA [Sidhu & Prasanna FCCM’01, …] ← Our interest
  - Features:
    - High throughput & large capacity
    - Complicated optimization process ← Our targeted problem
    - Non trivial solution construction ← Our targeted problem
REM by Finite Automata

- **RE-NFA**
  - Multiple active states
  - Multiple targets per input

- **NFA-based approach**
  - Logic based
  - One-hot encoded states
  - *Reconfigurable hardware (FPGA)*

- **RE-DFA**
  - Single active state
  - Single target per transition

- **DFA-based approach**
  - Memory based
  - Binary encoded state values

- Processor core(s)
Our Goals

- Automation of large-scale REM circuits construction on FPGA

  Regular Expressions $\rightarrow$ Automation Framework $\rightarrow$ RTL REM Circuit

- Pattern-level and circuit-level optimizations
  - Improve throughput
  - Reduce resource usage (LUT, BRAM, ...)
  - Reduce wiring and processing complexity

- Enhanced multi-pipeline architecture
  - Report multiple matches
  - Better resource aggregation

- Extensibility
  - Modular framework
  - Customized optimization plug-ins
Automation Overview

- Fully automated
  - Regexes to RE-NFAs
  - Pattern-level optimizations
  - RE-NFAs to REM circuit
  - Circuit-level optimizations

- Split the processing into two
  - Frontend → RE-NFA Construction
  - Backend → Circuit Generation

- Frontend processing
  - Group similar regexes
  - Extract complex character classes (for implementation in BRAM)
  - Balance group size

- Backend processing
  - Multi-character matching
  - Efficient character matching circuit (for implementation in logic)
Automation Steps for Regex

- Example regex: `/a(b|c)*(a|b)[a-z]/`

- Convert regex to RE-NFA (right)
  - Modular state structure
  - One character matching per state

- Map RE-NFA to RTL circuit (below)
  - Modular circuit architecture
  - Overlap character matching and state transition
  - Further Optimized by synthesis & PAR
Automation Steps for large-scale REM

• Organize large number of RE-NFAs
  • Collect complex character classes
  • Group and sort RE-NFAs
  • Balance group size

• Construct 2D multi-pipeline
  1. Distribute input characters
  2. Instantiate RE-NFA circuits
  3. Instantiate character matching
  4. Collect matching outputs

• Desired properties
  • Localized routing
  • Modular & extensible
  • Resource efficiency
### Implemented REM Features

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>$r_1r_2$</td>
<td>$r_1$ followed by $r_2$ (concatenation)</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>r_1</td>
</tr>
<tr>
<td></td>
<td>$r^+$</td>
<td>Repeat $r$ one or more times (repetition)</td>
</tr>
<tr>
<td></td>
<td>$r^*$</td>
<td>Repeat $r$ zero or more times (closure)</td>
</tr>
<tr>
<td></td>
<td>${m,n}$</td>
<td>Repeat $r$ at least $m$ and at most $n$ times</td>
</tr>
<tr>
<td>?</td>
<td>$r_1?r_2$</td>
<td>$r_1$ zero or one time followed by $r_2$</td>
</tr>
<tr>
<td>[...]</td>
<td>$[r_1-r_2]$</td>
<td>Any character between $r_1$ and $r_2$ (in Hardware)</td>
</tr>
<tr>
<td>[...]</td>
<td>$[^r_1-r_2]$</td>
<td>Any character but anything between $r_1$ and $r_2$ (in Hardware)</td>
</tr>
</tbody>
</table>

- **Support for up to 3 levels of nested parenthesis**
- **Kleene closure of two parenthesized sub-regexes is not supported**
Modified McNaughton-Yamada Construction

- Properties of MMY construction
  - Multiple input/output state transitions
  - No extra node added
  - Easy to implement in circuits

- Example
  - \( / a (b|c)^*(a|b) [a-z] / \)

- Modified McNaughton-Yamada
  - Merge at label-entry nodes (white)
  - Buffer at label-exit nodes (blue)
Faster and efficient operator implementation using Extended MMY

Native support for:
- Optionality operator
- Repetition Operator
- Constrained repetition
Frontend: Overview

- RE-NFA processing
  - Regex parsing
  - RE-NFA construction
  - Gather statistics
  - Character class & RE-NFA Categorization

- Stage processing
  - Balance stage size
  - Common prefix grouping
  - Character class aggregation
  - Other grouping techniques
Frontend: Parsing Regex to RE-NFA

- Consider regex
  `/hacker[0-9]*\(\s*tcp|udp\)+/`

- Build RE-NFAs for each sub-regex

- Combine sub-NFAs to form the full RE-NFA

- Internal representation format for RE-NFA

<table>
<thead>
<tr>
<th>State</th>
<th>Char./Char. class</th>
<th>Next state(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>2</td>
</tr>
</tbody>
</table>

- Gather statistics
  - Total states required
  - Operator complexity
  - Character class complexity
A Character class can possibly match multiple characters
- Ex: \([0-9a-f<>()]\) can match any characters in
  - Decimal digit ‘0’ through ‘9’
  - Hexadecimal digit ‘a’ through ‘f’
  - ‘<’ or ‘>’ or ‘(’ or ‘)’
- Expensive to match in logic

Categorize character classes as ‘simple’ and ‘complex’

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td># of characters</td>
<td>≤2</td>
<td>&gt;2</td>
</tr>
<tr>
<td>Example</td>
<td>[\r\n], [^\r\n]...,</td>
<td>[0-9], [a-z],...</td>
</tr>
</tbody>
</table>

Many character classes are simple; fewer are complex

Matched in the backend:
- Simple classes as efficient logic circuit
- Complex classes as memory (BRAM) access
Frontend: Optimization Plug-ins

- Framework allows extension through plugins

- RE-NFA grouping according to
  - Common prefix property => *Reduce overall resource usage*
  - Similar character classes => *Reduce character matching complexity*
  - Balanced group sizes => *Help simplify stage placement*

- Other grouping opportunities
  - *Purpose-oriented:* group regexes for the same stream
    - Enable/disable regex matching of the entire group
  - *Size-oriented:* group regexes with similar size
    - Simplify placement of RE-NFA circuits
  - Etc.
Backend: Overview

- Architecture Mapping
  - Multi-pipeline architecture
  - Map RE-NFA groups onto pipeline stages

- Circuit generation
  - Modular circuit construction
  - Efficient logic-based character matching

- Other features/Optimizations
  - Ability to report multiple matches
  - Multi-character matching per clock cycle
**Backend: LUT-based circuits**

- **Two types of state update logic**
  - Accept character matching or its negation
  - Handle negated character classes efficiently (e.g., `[^\r\n]`)

- **Simple character class matched by two or three 6-LUTs**
  - Results also propagate through stages to be reused

- **Results in significant BRAM usage reduction**
  - Minor LUT usage increase
Multi-Character Matching by Spatial Stacking

- MCM by spatial stacking
  - Reconnect STL1_outs to STL2_ins
  - Reconnect STL2_outs to STL1_ins
  - Remove state registers of STL1
  - Use same matching labels
  - OR both matching outputs

- Benefits
  - Low complexity: $O((n+e) \times m)$
  - Flexible: Any natural number $m$
  - Localized: only one RE-NFA
  - Simple: ~150 lines of C code

- Tradeoffs
  - Multiple BRAM usage (e.g. chA, chB) ⇒ Alleviated by aggregated & LUT-based character matching
  - More registers to buffer character matching results ⇒ Extra registers can also help increase clock rate
A 2-Character Input RE-NFA Circuit
Results: Performance

- **Frontend processing components**
  - RE-NFA construction
  - Character class categorization
  - Character class-based grouping
  - Up to 16k regexes (non-unique)

- **Backend processing components**
  - Multi-pipeline mapping
  - Character class aggregation
  - Circuit construction
  - For 760 unique regexes
  - Varying multi-character matching
  - 4-pipeline architecture

- On 2.3 GHz AMD Opteron 1356
### Results: Resource Utilization & Clock Rate

<table>
<thead>
<tr>
<th>m</th>
<th>LUTs</th>
<th>BRAM (Kb)</th>
<th>Clock Rate (MHz)</th>
<th>Compile Time (min)</th>
<th>Throughput (Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2*</td>
<td>31 k</td>
<td>216</td>
<td>303.2</td>
<td>-</td>
<td>4.8</td>
</tr>
<tr>
<td>2</td>
<td>30 k</td>
<td>69</td>
<td>276.3</td>
<td>41</td>
<td>4.4</td>
</tr>
<tr>
<td>4</td>
<td>47 k</td>
<td>138</td>
<td>202.9</td>
<td>54</td>
<td>6.4</td>
</tr>
<tr>
<td>8</td>
<td>84 k</td>
<td>345</td>
<td>178.3</td>
<td>112</td>
<td>11.4</td>
</tr>
</tbody>
</table>

- Significant BRAM reduction compared to non-optimized case
- Higher throughput with increased ‘m’ values
- Compilation time includes synthesis and place-and-route

*Results from [YeYang et al. ANCS’08]*
Thank you!
Regular Expressions

- **RE in formal language theory**
  - Language $L(r)$ over alphabet $\Sigma$
  - Base cases
    - $r = \emptyset$: empty set
    - $r = \varepsilon$: empty string
    - $r = a \in \Sigma$: character $a \in \Sigma$
  - Concat.: $L(r \cdot s) = L(r) \cdot L(s)$
  - Union: $L(r+s) = L(r) \cup L(s)$
  - Closure: $L(r^*) = L(r)^*$

- **For any regular pattern $r$**, exists
  - $N_1$ is an NFA with $\varepsilon$-transitions
  - $N_2$ is an NFA without $\varepsilon$-transitions
  - $M$ is a DFA
  - Such that $L(r) \equiv L(N_1) \equiv L(N_2) \equiv L(M)$
Modular Circuits for REM

- Example: \( a(b | c) \ast (a | b) [a \text{–} z] / \)
  - Two nodes per state
  - One character matching per state
  - Higher state fan-in & fan-out
  - Parallel character matching and state transition

- Directly mapped to VHDL circuits
  - Optimized by synthesis & PAR
  - ~300 lines of C code
Character Classification using BRAM

- **Character classification**
  - **Input**
    - Single input character
    - 8-bit binary value
  - **Output**
    - Membership to character classes
    - Single bit vectors

- **Character classifier table**
  - One row per bit value
  - One column per character class
  - Simple circuit construction
    - Set membership to `1`
  - 256 bits for any complex class
  - States share column output

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>a-z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>`'.'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>'a'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>'b'</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>'c'</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>'z'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

9/2/2010