Extending the PCRE Library with Static Backtracking Based Just-in-Time Compilation Support

Zoltán Herczeg
19/02/14
Outline

- Motivation
- PCRE-JIT compiler
- Static backtracking
- Results
  - Raw backtracking performance
  - PCRE-JIT engine
Motivation
History of Regular Expressions

- Kleene: regular sets in 1950s
- Thompson: Deterministic Finite Automaton (DFA) in 1960s
  - A pattern was constructed from character sets, and few operators: star, vertical bar
- Purpose of 'real' regular expressions
  - Defining a pattern which matches a set of words
    - `/c|ab*` matches c, a, ab, abb, abbb, ...
Regular Expressions Today

- Henry Spencer: backtracking engine in 1980s (open source)
  - Nondeterministic Finite Automaton (NFA)
- Improved later by PERL developers
  - Command based pattern matching language
  - Backward compatible
  - Closer to scripting languages
- Context sensitive decisions
  - Reduces complexity
Matches only when fox is outside of <> brackets:
PERL: /<((?:[^<>]*+(?:<(?1)|))++)?>(*SKIP)(*F)|fox/

Matches: <<fox>fox>fox
No match: <fox<<>fox>fox>
No match: <<fox>fox

Multibyte character sequences:
PERL: /a(?>
\n|\r|\n){2}b/
DFA: /a(?:\r(?::\n(?::\r\n|\r|\n)\n|\r\n|\n\r\n\n)?\b/
\[(x|x)^n y/\]

\[/a[^b]^n abc/\]

\begin{center}
\begin{tabular}{c}
\includegraphics[width=0.5\textwidth]{chart1.png} & \includegraphics[width=0.5\textwidth]{chart2.png}
\end{tabular}
\end{center}

\textbf{PCRE}  \hspace{1cm} \textbf{RE2}
Summary

- New approach is needed to accelerate these “regular expressions”
  - DFA is not enough anymore
- Command based languages can be efficiently accelerated by Just-in-time (JIT) compilation
  - 2009 Irregexp
  - 2009 YARR (Yet Another Regex Runtime)
  - 2011 PCRE-JIT
PCRE-JIT compiler
PCRE Overview

- Perl Compatible Regular Expressions
  - Standalone C library, which supports PERL style patterns
- Started by Philip Hazel in 1997
- PCRE-JIT supports all of its features
  - Comparison of regular expression engines in Wikipedia: Only PCRE supports all 19 categories

Main Components of PCRE-JIT

PCRE byte code

Static Backtracking based PCRE-JIT compiler

PCRE-JIT engine

SLJIT compiler

Machine code
SLJIT Compiler

- Low-level assembly like language
  - Translated to machine code
- Platform independent
  - LIR is designed by uniting the common features of CPUs
- Supported architectures
  - x86-32/64, ARMv5/7/Thumb2, PowerPC-32/64, MIPS-32, SPARC-32
  - ARM-64 is close to done
Evolution of Backtracking Algorithms
Interpreting (?:B)+ Pattern

```java
boolean recursive_match() {
  // Infinite loop for tail merging.
  while (TRUE) {
    switch (*current_opcode) {
      // ...

      case GREEDY_PLUS_CLOSE_BRACKET:
        if (recursive_match() == SUCCESS)
          return SUCCESS
        // Tail merge.
        current_opcode += opcode_size(
          GREEDY_PLUS_CLOSE_BRACKET)
        // Back to the while loop.
        break;

      // ...
    }
  }
}
```
Static Backtracking

- JIT compilers replaced recursive function calls to try-catch blocks
  - Unknown destination when backtracking
- Our new approach: Static Backtracking
- Generating code from Abstract Syntax Tree (AST)
  - Each node has exactly one parent node, and it is known at compile time
Nondeterministic Finite Automaton

Abstract Syntax Tree

Continue matching: green lines
Perform backtracking: red lines / texts
Static Backtracking (2)

- Not all NFAs are supported
  - Enough for PERL compatible patterns: they are always compiled to an AST first
- Context dependent backtracking
- No need to set-up catch handlers
- Direct, conditional jumps
  - Indirect jumps are usually not conditional: two jumps are needed for checking a condition and backtrack on fail
Overview of Building Blocks

- Two code paths are generated from each AST node
  - Matching path: action
  - Backtracking path: reverse action
- These two code paths form a single function
  - With two entry, two exit points
  - Optimization: these points can represent passing or returning a boolean value
Basic Building Block

Try a new match. Input position contains a valid starting position.

Find another match. A valid context is provided on the top of the stack.

Matching path

Backtracking path

A match is found. A valid context is stored on the top of the stack. The input position contains the end of the match.

No match is found. Removes the topmost context if appropriate. Input position is undefined.
Control Flow of /A(?::B)+C/
Results
Raw Backtracking Performance

- Measured on pathological cases
  - Exponential runtime, large amount of backtracking.
- Compared to Irregexp (in V8 JavaScript engine)
  - Traditional (indirect jump) based backtracking
  - Highly optimized engine
- These patterns cannot be optimized by backtracking elimination techniques
  - E.g: expected character check
# Pathological Cases

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input</th>
<th>Irregexp (ms)</th>
<th>Irregexp ratio</th>
<th>PCRE JIT (ms)</th>
<th>PCRE JIT ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 /a?26c</td>
<td>$a^{26}bc$</td>
<td>777.0</td>
<td>2.20</td>
<td>352.5</td>
<td>1.00</td>
</tr>
<tr>
<td>2 /a+14c</td>
<td>$a^{26}b^{14}c$</td>
<td>221.6</td>
<td>1.21</td>
<td>182.8</td>
<td>1.00</td>
</tr>
<tr>
<td>3 /(?a+)c</td>
<td>$a^{23}bc$</td>
<td>101.2</td>
<td>1.44</td>
<td>70.5</td>
<td>1.00</td>
</tr>
<tr>
<td>4 /(?a+)c</td>
<td>$a^{23}bc$</td>
<td>102.8</td>
<td>1.20</td>
<td>85.9</td>
<td>1.00</td>
</tr>
<tr>
<td>5 /(?aa+)c</td>
<td>$a^{46}bc$</td>
<td>99.5</td>
<td>1.18</td>
<td>84.2</td>
<td>1.00</td>
</tr>
<tr>
<td>6 /((a+)c</td>
<td>$a^{23}bc$</td>
<td>220.1</td>
<td>1.70</td>
<td>129.2</td>
<td>1.00</td>
</tr>
<tr>
<td>7 /(?:(?:aa)+)+c</td>
<td>$a^{30}bc$</td>
<td>217.7</td>
<td>2.14</td>
<td>101.7</td>
<td>1.00</td>
</tr>
<tr>
<td>8 /((aa)+)+c</td>
<td>$a^{28}bc$</td>
<td>364.6</td>
<td>2.16</td>
<td>168.6</td>
<td>1.00</td>
</tr>
<tr>
<td>9 /(?a[a]26c</td>
<td>$a^{26}bc$</td>
<td>110.5</td>
<td>1.00</td>
<td>400.7</td>
<td>3.63</td>
</tr>
<tr>
<td>10 /(?aa[a]24c</td>
<td>$a^{48}bc$</td>
<td>194.2</td>
<td>1.93</td>
<td>100.4</td>
<td>1.00</td>
</tr>
<tr>
<td>11 /aa[a]24c</td>
<td>$a^{48}bc$</td>
<td>179.4</td>
<td>1.15</td>
<td>155.6</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Overall Speedup

- Benchmark set provided by SNORT Intrusion Detection System (IDS)
  - >1000 HTTP content filtering patterns
- Compared to PCRE Interp., and Irregexp
  - All engine optimizations are enabled (including backtracking eliminations)
- Backtracking is not a rare event
  - About 46% of matching attempts are failed in the PCRE interpreter
# PCRE-JIT vs PCRE Interpreter

<table>
<thead>
<tr>
<th>Target CPU</th>
<th>Average speedup (x as fast)</th>
<th>&lt; 3.0 x as fast</th>
<th>3.0 - ∞ x as fast</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>% of patterns</td>
<td>% of total runtime</td>
</tr>
<tr>
<td>x86/32</td>
<td>6.84</td>
<td>84.99%</td>
<td>3.66%</td>
</tr>
<tr>
<td>x86/64</td>
<td>5.55</td>
<td>82.92%</td>
<td>3.80%</td>
</tr>
<tr>
<td>ARM-V7/32</td>
<td>6.76</td>
<td>82.24%</td>
<td>3.71%</td>
</tr>
<tr>
<td>ARM-THUMB2/32</td>
<td>7.24</td>
<td>84.59%</td>
<td>3.78%</td>
</tr>
<tr>
<td>PowerPC/32</td>
<td>5.48</td>
<td>88.13%</td>
<td>6.47%</td>
</tr>
<tr>
<td>PowerPC/64</td>
<td>5.55</td>
<td>88.42%</td>
<td>6.38%</td>
</tr>
<tr>
<td>SPARC/32</td>
<td>5.85</td>
<td>83.61%</td>
<td>4.08%</td>
</tr>
<tr>
<td>MIPS/32</td>
<td>7.59</td>
<td>81.35%</td>
<td>3.27%</td>
</tr>
<tr>
<td>Average</td>
<td>6.36</td>
<td>84.53%</td>
<td>4.39%</td>
</tr>
<tr>
<td>Std. dev.</td>
<td>0.79</td>
<td>2.42%</td>
<td>1.19%</td>
</tr>
</tbody>
</table>
Overall: PCRE-JIT is 1.63x faster than Irregexp

Number of patterns in each group

- S: 799 (78.56%)
- M: 123 (12.09%)
- L: 95 (9.34%)

Total runtime (s) of each group

S: Patterns with short runtime (0-20 ms)
M: Patterns with medium runtime (20-200 ms)
L: Patterns with long runtime (200-2000 ms)
Patterns(%) Unsupported by DFA

- Most patterns in group L are not supported by DFA based engines!
- DFA is not a viable option to accelerate these long running patterns
Conclusion and Future Work
Conclusion

- Regular expressions are changed
  - Better to optimize them using JIT compiling

- Traditional NFA based backtracking can be improved by AST based backtracking
  - Context dependent backtracking

- Results (SNORT benchmark set)
  - 6.36x faster than interpreter
  - 1.63x faster than Irregexp
Future Work

- Further improve the JIT compiler
  - Integrating optimizations used by other engines
  - A simplified Boyer-Moore string search was already implemented
- Support more CPUs (in SLJIT)
  - MIPS-64, SPARC-64
PCRE-JIT Is Production Ready

- NGINX Web Server
- Qt Framework (since 5.0)
- Suricata Intrusion Detection System
- Apache ModSecurity Firewall
- GNU grep (with -P option)
- HAPProxy Load Balancer
- Sigil E-book Editor
Thanks for listening
Questions?