Performance Matters—A New Algorithmic Approach for Fast Unicode Regular Expression Processing

Robert D. Cameron

School of Computing Science
Simon Fraser University

October 28, 2015

Overview

The Parabix Regular Expression Project

- Goal: building a modern, Unicode-compliant, ultra-fast, regular expression engine and application framework.
- New algorithmic approach: bitwise data parallel matching.
- Performance target: Gigabyte/second regexp processing.
- Status: icgrep 1.0 is a full Unicode Level 1 grep replacement.
- Current work: Unicode Level 2 Regular Expression Support
- Longer-term: high-level application framework.

UTS #18 - Unicode Regular Expressions

Unicode Level 1

- General category (e.g., `\p{Lu}`), script (e.g. `\p{Greek}`), and other core Unicode properties.
- Simple Unicode line breaks, word boundaries.
- Set operations, e.g., `[\p{Greek}&&\p{Lu}]`
- Simple Unicode case-insensitive matching.

UTS #18 - Unicode Regular Expressions

Unicode Level 1

- General category (e.g., `\p{Lu}`), script (e.g. `\p{Greek}`), and other core Unicode properties.
- Simple Unicode line breaks, word boundaries.
- Set operations, e.g., `[\p{Greek}&&\p{Lu}]`
- Simple Unicode case-insensitive matching.

Unicode Level 2

- Grapheme clusters and grapheme cluster mode.
- Canonical equivalents.
- Name property with regexp values `\p{name=/AIRPLANE/}`
- Broad set of Unicode general, numeric, identifier, case, normalization, shaping, bidirectional, CJK and other properties.

Performance Matters - Big Text

Text Analytics

- Text Analytics applications: very large textual data sets.
- Gigabyte to Terabyte scale.
- Regular expressions are often used in these applications.
- If performance were better, regular expressions could be applied more widely.

Performance Matters - Big Text

Text Analytics

- Text Analytics applications: very large textual data sets.
- Gigabyte to Terabyte scale.
- Regular expressions are often used in these applications.
- If performance were better, regular expressions could be applied more widely.

IBM System T Study

- Five customer workloads studied
- Four of five workloads dominated by regexp processing.
- 60%-75% of total execution time.
- Polig *et al.* Giving text analytics a boost. *IEEE Micro*, **34**(4):6–14, 2014.

Performance Matters - Catastrophic Backtracking

Nondeterministic Regular Expression Features

- Ambiguity: regexes may match strings in multiple ways.
- Ex: match(`^(\p{L})+t(\p{L})+$`, "`statistics`")
- (`s`, `atistics`), (`sta`, `istics`), or (`statis`, `ics`)
- Backtracking engines: potential runaway behaviour.
- Procedural workarounds: `^(\p{L})++t(\p{L})+$` or `^(?>(\p{L})+?)t(\p{L})+$`
- Declarative nature of regexps diminished.

Performance Matters - Catastrophic Backtracking

Nondeterministic Regular Expression Features

- Ambiguity: regexes may match strings in multiple ways.
- Ex: match(`^(\p{L})+t(\p{L})+$`, "`statistics`")
- (`s`, `atistics`), (`sta`, `istics`), or (`statis`, `ics`)
- Backtracking engines: potential runaway behaviour.
- Procedural workarounds: `^(\p{L})++t(\p{L})+$` or `^(?>(\p{L})+?)t(\p{L})+$`
- Declarative nature of regexps diminished.

Web Site Security

- Regular expressions may be used in web services.
- Malicious agents may inject ill-formed regular expressions.
- Web sites may be tied up (denial-of-service) or compromised.

Much Better Performance is Possible

Unicode regular expression matching can be ultra-fast and robust.

Example: email regex with Unicode properties

- `([^\p{Z}<]+@[\p{L}\p{M}\p{N}.-]+\.`
  `(\p{L}\p{M}*){2,6})(>|\p{Z}|$)`
- Compare time in CPU cycles per byte for 4 grep programs.
- Data source: 620 MB Wikibooks document set (15 languages)
- Processor: Intel Core i7-2600 CPU @ 3.40GHz

Much Better Performance is Possible

Unicode regular expression matching can be ultra-fast and robust.

Example: email regex with Unicode properties

- `([^\p{Z}<]+@[\p{L}\p{M}\p{N}.-]+\.`
  `(\p{L}\p{M}*){2,6})(>|\p{Z}|$)`
- Compare time in CPU cycles per byte for 4 grep programs.
- Data source: 620 MB Wikibooks document set (15 languages)
- Processor: Intel Core i7-2600 CPU @ 3.40GHz

Performance Results

Program	Elapsed sec	Cycles/Byte	Megabytes/sec
ugrep561	180.7	1024.0	3.60
pcre2grep	144.0	815.5	4.51
grep210 -P	87.3	498.2	7.45
icgrep4750	1.06	6.05	613.0

Bitwise Data Parallel Regular Expression Matching

Beyond Byte-At-A-Time

- Traditional regular expression technology processes one code unit at a time using DFA, NFA or backtracking implementations.
- Instead consider a bitwise data parallel approach.
- Byte-oriented data is first transformed to 8 parallel bit streams (Parabix transform).
- Bit stream j consists of bit j of each byte.
- Load 128-bit SIMD registers to process 128 positions at a time in bitwise data parallel fashion (SSE2, ARM Neon, ...).
- Or use 256-bit AVX2 registers of newer Intel processors.
- Process using bitwise logic, shifting and addition.
- Parabix methods have previously been used to accelerate Unicode transcoding and XML parsing.

Unbounded Stream Abstraction

- Program operations as if *all positions in the file are to be processed simultaneously*.
- Unbounded bitwise parallelism.
- Pablo compiler technology maps to block-by-block processing.
- Information flows between blocks using carry bits.
- LLVM compiler infrastructure for Just-in-Time compilation.
- Custom LLVM improvements further accelerate processing.

Marker Streams

- Marker stream M_i indicates the positions that are reachable after item i in the regular expression.
- Each marker stream M_i has one bit for every input byte in the input file.
- $M_i[j] = 1$ if and only if a match to the regular expression up to and including item i in the expression occurs at position $j - 1$ in the input stream.
- Conceptually, marker streams are computed in parallel for all positions in the file at once (bitwise data parallelism).
- In practice, marker streams are computed block-by-block, where the block size is the size of a SIMD register in bits.

Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.

input data `a453z--b3z--az--a12949z--ca22z7--`

Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- M_1 marks positions after occurrences of a.

input data	`a453z--b3z--az--a12949z--ca22z7--`
M_1	`.1..........1...1.........1.....`

Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- M_1 marks positions after occurrences of `a`.
- M_2 marks positions after occurrences of `a[0-9]*`.

input data	`a453z--b3z--az--a12949z--ca22z7--`
M_1	`.1..........1...1.........1.....`
M_2	`.1111........1...111111....111...`

Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- M_1 marks positions after occurrences of `a`.
- M_2 marks positions after occurrences of `a[0-9]*`.
- M_3 marks positions after occurrences of `a[0-9]*[z9]`.

```
input data   a453z--b3z--az--a12949z--ca22z7--
   M1        .1..........1...1.........1.....
   M2        .1111.......1...111111....111...
   M3        .....1.......1.....1.11......1..
```

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- CCC(`cc_a = [a]`)
- ```
  temp1 = (bit[1] &~ bit[0])
  temp2 = (bit[2] &~ bit[3])
  temp3 = (temp1 & temp2)
  temp4 = (bit[4] | bit[5])
  temp5 = (bit[7] &~ bit[6])
  temp6 = (temp5 &~ temp4)
  cc_a = (temp3 & temp6)
  ```
  ```

Character Class Ranges

- Ranges of characters are often very simple to compute.
- CCC(`cc_0_9 = [0-9]`)
- ```
  temp7 = (bit[0] | bit[1])
  temp8 = (bit[2] & bit[3])
  temp9 = (temp8 &~ temp7)
  temp10 = (bit[5] | bit[6])
  temp11 = (bit[4] & temp10)
  cc_0_9 = (temp9 &~ temp11)
  ```
  ```

Character Class Common Subexpressions

- Multiple definitions use common subexpressions.
- CCC(`cc_z9 = [z9]`)
- ```
  temp12 = (bit[4] &~ bit[5])
  temp13 = (temp12 & temp5)
  temp14 = (temp9 & temp13)
  temp15 = (temp1 & temp8)
  temp16 = (bit[6] &~ bit[7])
  temp17 = (temp12 & temp16)
  temp18 = (temp15 & temp17)
  cc_z9 = (temp14 | temp18)
  ```
  ```

Matching Character Class Repetitions with MatchStar

Matching Character Class Repetitions with MatchStar

- $\text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) \vee M$

Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 = \text{MatchStar}(M_1, C)$

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1.........1...1........1....
$C = [0\text{-}9]$	.111....1.......11111....11.1..

Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 = \text{MatchStar}(M_1, C)$
- Use addition to scan each marker through the class.

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1.........1...1........1....
$C = [0\text{-}9]$	.111....1.......11111....11.1..
$T_0 = M_1 \wedge C$	.1.............1........1....
$T_1 = T_0 + C$	....1...1............1.....11..

Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 = \text{MatchStar}(M_1, C)$
- Use addition to scan each marker through the class.
- Bits that change represent matches.

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1.........1...1........1....
$C = [0\text{-}9]$	.111....1.......11111....11.1..
$T_0 = M_1 \wedge C$	.1.............1........1....
$T_1 = T_0 + C$	....1...1...........1.....11..
$T_2 = T_1 \oplus C$	.1111...........111111...111...

Matching Character Class Repetitions with MatchStar

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 = \text{MatchStar}(M_1, C)$
- Use addition to scan each marker through the class.
- Bits that change represent matches.
- We also have matches at start positions in M_1.

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1.........1...1........1....
$C = [0\text{-}9]$	.111....1.......11111....11.1..
$T_0 = M_1 \wedge C$	.1.............1........1....
$T_1 = T_0 + C$	....1...1...........1.....11..
$T_2 = T_1 \oplus C$	.1111...........111111...111...
$M_2 = T_2 \vee M_1$	.1111........1...111111...111...

Matching Equations

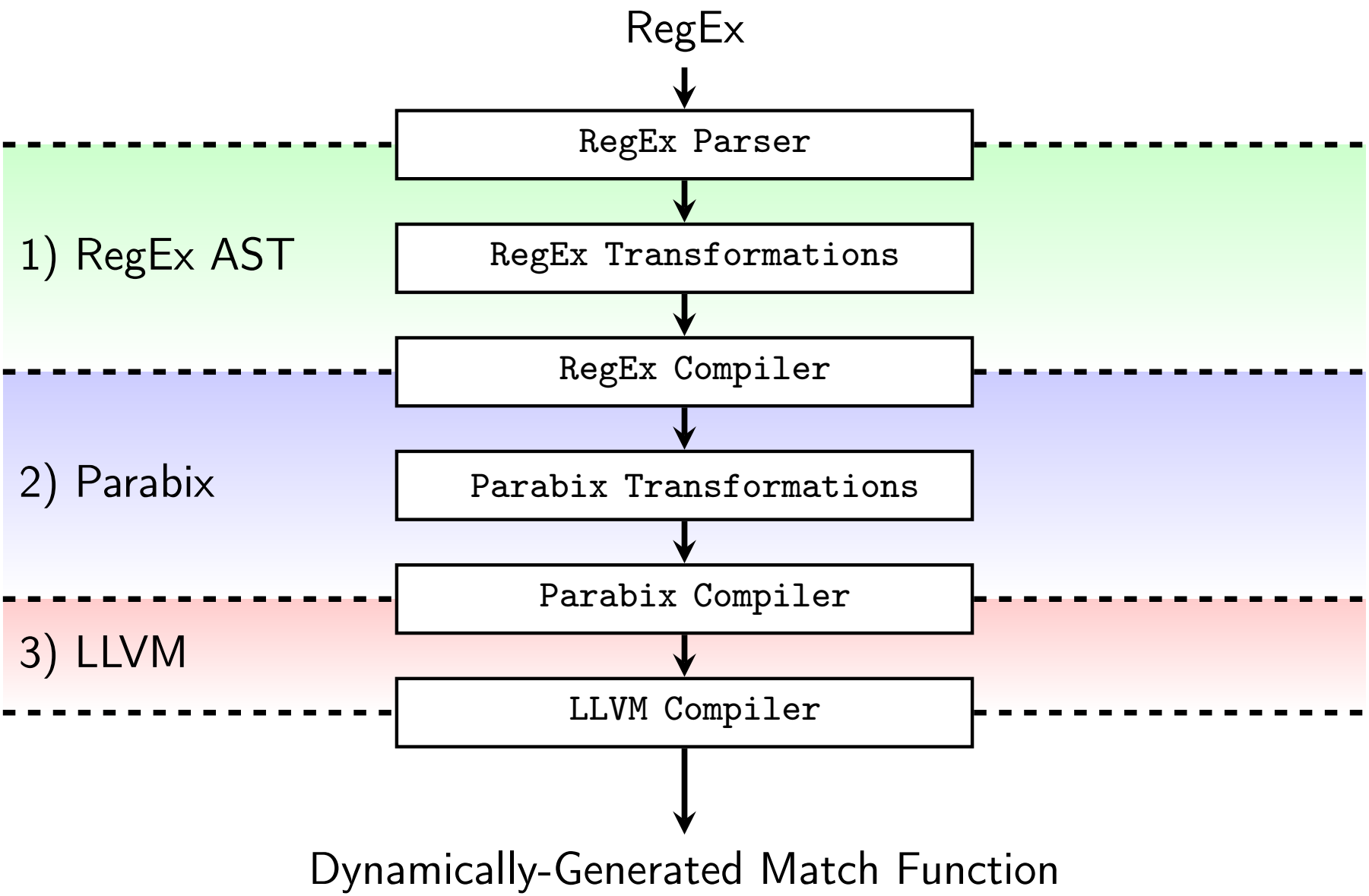
The rules for bitwise data parallel regular expression matching can be summarized by these equations.

$$
\begin{aligned}
\text{Match}(m, C) &= \text{Advance}(\text{CharClass}(C) \wedge m) \\
\text{Match}(m, RS) &= \text{Match}(\text{Match}(m, R), S) \\
\text{Match}(m, R|S) &= \text{Match}(m, R) \vee \text{Match}(m, S)) \\
\text{Match}(m, C*) &= \text{MatchStar}(m, \text{CharClass}(C)) \\
\text{Match}(m, R*) &= m \vee \text{Match}(\text{Match}(m, R), R*) \\
\text{Advance}(m) &= m + m \\
\text{MatchStar}(m, C) &= (((m \wedge C) + C) \oplus C) \vee m
\end{aligned}
$$

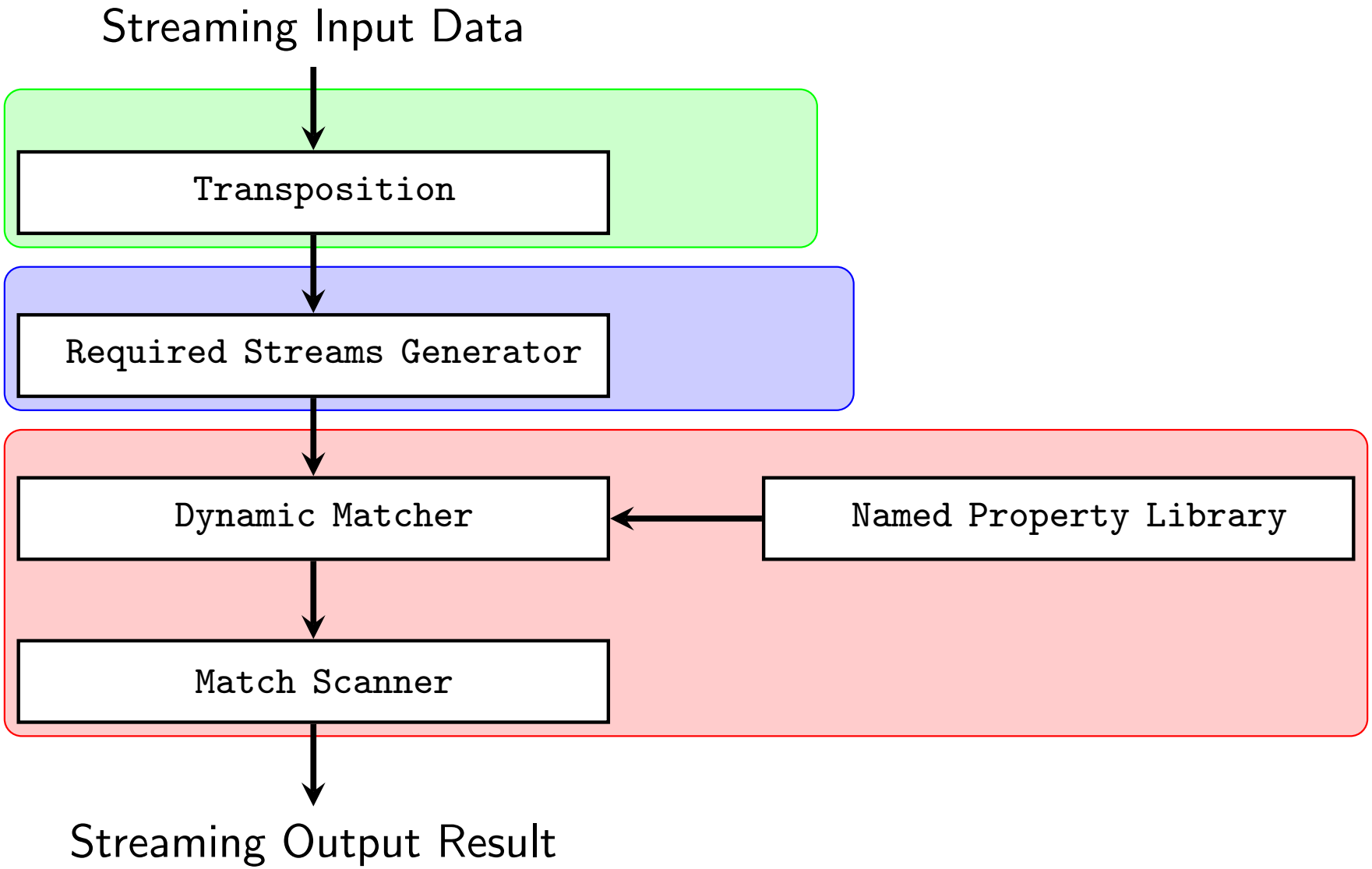
The recursive equation is implemented with a while loop.

Architecture

Compilation Architecture



Execution Architecture



UTF-8 Regular Expression Matching

UTF-8 Byte Classification and Validation

$$
\begin{aligned}
\text{ASCII} &= \text{CharClass}(\texttt{[\textbackslash x00-\textbackslash x7F]}) \\
\text{Prefix} &= \text{CharClass}(\texttt{[\textbackslash xC2-\textbackslash F4]}) \\
\text{Pfx3or4} &= \text{CharClass}(\texttt{[\textbackslash xE0-\textbackslash xF4]}) \\
\text{Prefix4} &= \text{CharClass}(\texttt{[\textbackslash xF0-\textbackslash xF4]}) \\
\text{Suffix} &= \text{CharClass}(\texttt{[\textbackslash x80-\textbackslash xBF]}) \\
\text{Scope} &= \text{Advance}(\text{Prefix}) \vee \text{Advance}(\text{Pfx3or4}, 2) \vee \text{Advance}(\text{Prefix4}, 3) \\
\text{Mismatch} &= \text{Scope} \oplus \text{Suffix} \\
\text{Initial} &= \text{ASCII} \vee \text{Prefix} \\
\text{NonFinal} &= \text{Prefix} \vee \text{Advance}(\text{Pfx3or4}) \vee \text{Advance}(\text{Prefix4}, 2)
\end{aligned}
$$

UTF-8 Matching Operations

Advance and Matchstar can both be adapted for UTF-8 matching. Let U8Class(U) be the stream marking the *final* UTF-8 byte position of occurrences of members of the class.

$$\begin{aligned} \text{Match}(m, U) &= \text{Advance}(\text{ScanThru}(m, \text{NonFinal}) \wedge \text{U8Class}(U)) \\ \text{Match}(m, U*) &= \text{MatchStar}(m, \text{U8Class}(U) \vee \text{NonFinal}) \wedge \text{Initial} \\ \text{ScanThru}(m, C) &= (m + C) \wedge \neg C \end{aligned}$$

Simple UTF-8 Matching Operations

- Search for 你好 in UTF-8 text.
- Let ni3 represent the three byte sequence for 你.
- Let hao represent the three byte sequence for 好。
- Let men represent the three byte sequence for 们。

input data	`ni3hao(Hello),ni3men(You),`
CC_{ni3}	`..1...........1.....`
CC_{hao}	`.....1..............`
$m_0 = \text{Initial}$	`1..1..111111111..1..111111`
NonFinal	`11.11.....11.11.....`
$t_1 = \text{ScanThru}(m_0, \text{NonFinal})$	`..1..111111111..1..1111111`
$m_1 = \text{Advance}(t_1 \wedge CC_{ni3})$	`...1...........1.....`
$t_2 = \text{ScanThru}(m_1, \text{NonFinal})$	`.....1...........1.....`
$m_2 = \text{Advance}(t_2 \wedge CC_{hao})$	`......1..............`

If Hierarchies

- Unicode property classes may contain many thousands of codepoints.
- Evaluating all required equations may be very expensive.
- However, any 128-byte segment of input will involve only a few codepoint ranges.
- Evaluation of property classes is embedded in if-hierarchies of successively finer ranges.
- This techique greatly reduces the number of equations evaluated for each property.

iCgrep 1.0

icGrep 1.0 provides full Unicode Level 1 support.

- RL1.1 Hex Notation
- RL1.2 Properties
  - General_Category
  - Script and Script_Extensions
  - Alphabetic, Uppercase, Lowercase, White_Space, Noncharacter_Code_Point, Default_Ignorable_Code_Point
- RL1.3 Subtraction and Intersection
- RL1.4 Simple Word Boundaries
- RL 1.5 Simple Loose Matches
- RL1.6 Line Boundaries
- RL1.7 Supplementary Code Points

iCgrep-devel r4850

Current status of Unicode Level 2 support in icGrep.

- 2.1 Canonical Equivalents: TO DO (only in Grapheme Cluster Mode)
- 2.2 Extended Grapheme Clusters: `\b{g}` implemented
- 2.2.1 Grapheme Cluster Mode: `(?g)` implemented
- 2.3 Default Word Boundaries: currently Level 1
- 2.4 Default Case Conversion: currently Level 1
- 2.5 Name Properties: implemented
- 2.6 Wildcards in Property Values: Regular Experession in Name Properties
- 2.7 Full Properties: Most Properties implemented

Emoji Regular Expressions!

- Individual Emoji can be entered as characters in regular expressions.
- Or as hex codepoints.
- Even better: search by name pattern:
  - Create a pattern for Emoji character names: `\bSMIL(E|ING)\b`
  - Search documents for any character having the name pattern.
    `icgrep '\N{\bSMIL(E|ING)\b}'`

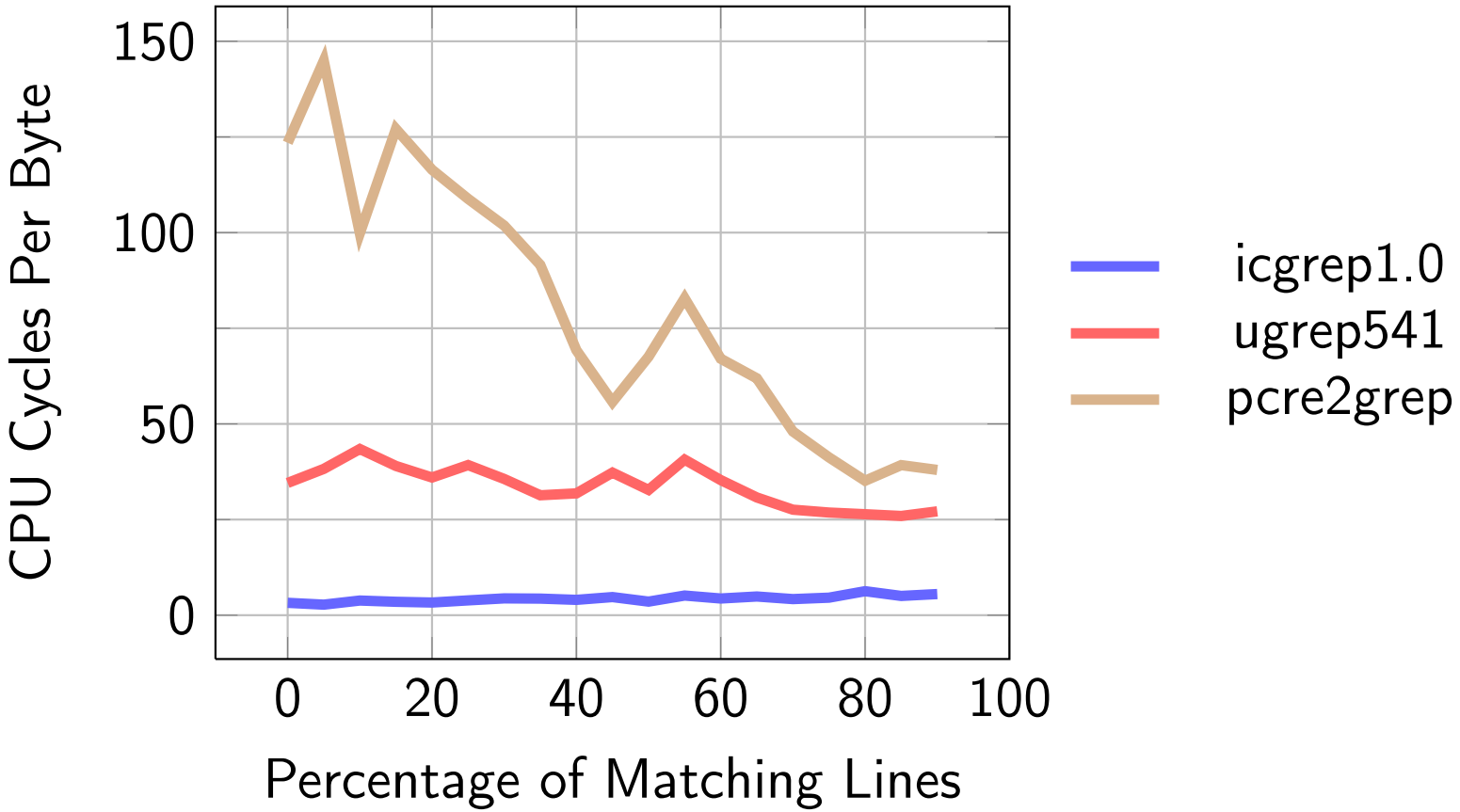
Performance Study: Unicode Set Operations

`[\p{Greek}&&\{lowercase}]`

Performance Study Set-up

- Search for lines containing characters in computed sets.
- Set expressions combine general category and script properties.
- Set intersection, union, difference, negation.
- Cover all general categories and scripts.
- 246 test expressions in all.
- Evaluate icgrep 1.0 vs. ugrep541 and pcre2grep.
- Measure performance against a large collection of Wikimedia documents.

Performance Comparison



Results

- Regular expression matching using bitwise data parallelism is well-suited to Unicode regular expression matching requirements.
- Performance is ultra-fast and robust against all kinds of nondeterminstic regular expressions.
- Performance improves as SIMD processor width increases.
- Unicode level 1 requirements fully met.
- Unicode level 2 requirements partially met:
  - grapheme cluster boundaries `\b{g}`
  - grapheme cluster mode: `(?g)`
  - Unicode name expressions search `\N{LATIN.*\b[A-G]\b}`
  - Broad Unicode property support
- Open source implementation available:
  `http://parabix.costar.sfu.ca/wiki/ICgrep`