Software Toolchain for Large-Scale RE-NFA Construction on FPGA

Yi-Hua E. Yang, Viktor K. Prasanna

Dept. of Electrical Engineering, University of Southern California

yeyang@usc.edu, prasanna@usc.edu

Abstract—We present a software toolchain for constructing large-scale *regular expression matching* (REM) on FPGA. The software automates the conversion of regular expressions into compact and high-performance non-deterministic finite automata (RE-NFA) [17]. Assuming a fixed number of fan-out transitions per state, an n-state m-bytes-per-cycle regular expression matching engine (REME) can be constructed in $O(n \times m)$ time and $O(n \times m)$ memory by our software. The resulting circuit occupies no more than $O(n \times m)$ slices on FPGA. Based on the proposed algorithms, we develop prototype software that converts arbitrary regular expressions into RTL codes in VHDL, utilizing both logic slices and block memory (BRAM) available on FPGA devices. A large number of RE-NFAs are placed onto a two-dimensional *staged pipeline*, allowing scalability to thousands of RE-NFAs with linear area increase and little clock rate penalty due to scaling. On a PC with a 2 GHz Athlon64 processor and 2 GB memory, our prototype software converts hundreds of regular expressions from Snort [2] into VHDL in less than 10 seconds. We also designed a benchmark generator which can produce regular expressions with configurable pattern complexity parameters, including state count, state fan-in, loop-back and feed-forward distances. Several regular expressions with various complexities are used to test the performance of our RE-NFA construction software.

Index Terms—Regular expression, FPGA, BRAM, finite state machine, NFA

I. Introduction

Regular expression matching (REM) has many applications ranging from text processing to packet filtering. In the narrow sense, each regular expression defines a regular language over the alphabet of input characters. A regular language applies three basic operators on the alphabet: *concatenation* ($\cdot$), *union* ($|$), and *Kleene closure* ($*$), which allow the construction of complex expressions. There are other common operators that also conform to the regular language construct, such as *character classes* ([...]), *optionality* (?) and *constrained repetitions* ({a, }, {, b}, {a, b}). All of these operators can be realized by proper arrangements of the three basic ones.

Improving large-scale REM performance has been a research focus in recent years [3], [5], [6], [9], [11], [10], [13], [14], [16], [17], [18]. Since regular languages can be necessarily and sufficiently accepted by finite state automata, a regular expression matching engine (REME) supporting *concatenation*, *union*, *closure*, *repetition*, and *optionality* can always be implemented as either a non-deterministic finite

automaton (RE-NFA) or a deterministic finite automaton (RE-DFA). Figure 1 compares side-by-side the architectures of the two types of automata.

In an RE-NFA approach [5], [9], [13], [14], [16], [17], individual regular expressions and their character matching states are processed in parallel with one another. As a result, more than one state in an RE-NFA can be *active* at any time. Optimizations such as input/output pipelining [9], common-prefix extraction [5], [9], multi-character input [16], [17], and centralized character decoding [5], [12], can be applied to improve throughput and reduce resource requirements of the overall design.

In an RE-DFA approach, several regular expressions are grouped (`union`'d) into a DFA by expanding different combinations of active states into additional *combined states*. In principle, only one combined state in an RE-DFA is active at any time. Various techniques [4], [10], [11], [15] are then applied to improve memory access efficiency and to reduce the total number of states, which usually suffers from quadratic to exponential explosion [18].

Due to the matching power of regular expressions and the complexity of the strings being matched, the REM process can be the slowest bottleneck of a system. To match a regular expression of length n over an alphabet of size Σ can take up to $O(n^2)$ time to process each character (for RE-NFA) or $O(\Sigma^n)$ memory space to store the state transition table (for RE-DFA) [18]. Furthermore, to match K concurrent regular expressions, the overall throughput could be K times slower (for RE-NFA) or take $O(\Sigma^K)$ more memory space (for RE-DFA) in the worst case.

Modern FPGAs offer large amount of reconfigurable logic (LUTs) and on-chip memory (BRAM). We developed a compact and high-performance RE-NFA architecture for REM which utilizes both on-chip logic and memory resources on modern FPGAs [17]. In this study, we focus on the automatic parsing, translation and construction of regular expressions matching engine (REME) using our RE-NFA architecture for fully automated FPGA implementation. More specifically, we develop a REME construction software with the following components:

1) Automatic conversion from regular expression parse tree [8] to a uniform and modular RE-NFA structure.
2) Automatic generation of RTL code in VHDL for each RE-NFA. The resulting circuit is spatially stacked a configurable number of times for multi-character matching.
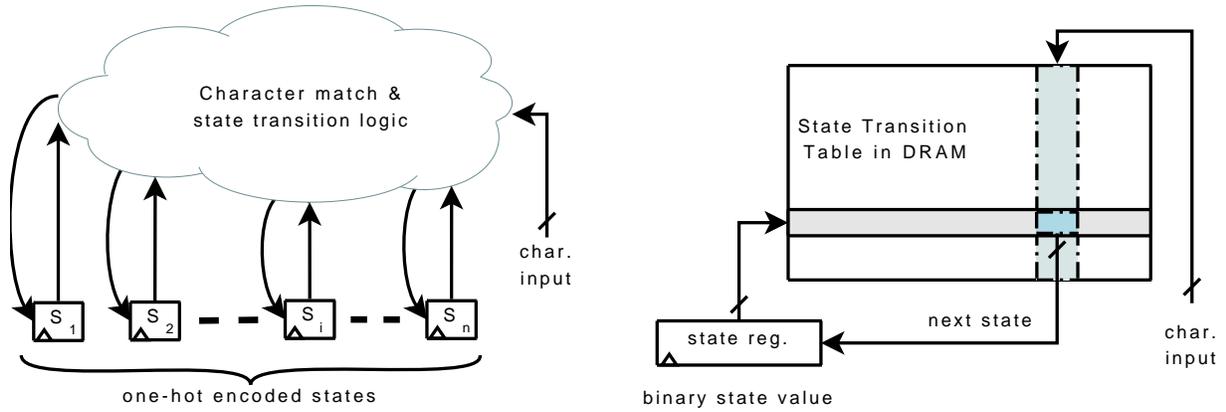3) Allocation of centralized character classification in

Figure 1. Basic architectures of RE-NFA (left) and RE-DFA (right). Our work focus on regular expression matching using the RE-NFA architecture.

BRAM for up to 256 REMEs using a simple heuristics.

4) Automatic construction of up to 16 pipelines in a two-dimensional structure.

5) A benchmark generator of regular expressions with configurable pattern complexity parameters (*state count*, *state fan-in*, *loop-back* and *feed-forward* distances).

The rest of this paper is organized as follows. The background and prior work of RE-NFA on FPGA are discussed in Section II. An overview of our software toolchain is given in Section III. Section IV describes REME construction, while Section V covers architectural optimization. Section VI introduces a REME benchmark generator and uses it to evaluate the performance of the REME construction and optimization software. Section VII concludes the paper.

II. BACKGROUND AND RELATED WORK

Hardware implementation of regular expression matching (REM) was first studied by Floyd and Ullman [8], where an n-state RE-NFA is translated into integrated circuits using no more than $O(n)$ circuit area. Sidhu and Prasanna [14] later proposed an algorithm to implement REM on FPGA in a similar RE-NFA architecture, which has been used by most other RE-NFA implementations on FPGAs ([5], [9], [13], [16]). Yang and Prasanna [17] adopted a different approach to translate arbitrary regular expressions to corresponding RE-NFAs with a more modular and uniform circuit structure.

Automatic REME construction on FPGAs was first proposed in [9] using JHDL for both regular expression parsing and circuit generation. In particular, the (J)HDL construction approach used in [9] is in contrast to the *self-configuration* approach done by [14]. Reference [9] also considered large-scale REME construction, where the character input is broadcasted globally to all states in a tree-structured pipeline. Automatic REME construction in VHDL was proposed in [5] and [13]. In [5], the regular expression was first tokenized and parsed into a hierarchy of basic NFA blocks, then constructed in VHDL using a bottom-up scheme. In [13], a set of scripts were used to compile regular expressions into op-codes, to convert op-codes into NFA, and to construct the NFA circuits in VHDL.

A multi-character decoder was proposed in [7] to improve pattern matching throughput. While the technique was claimed
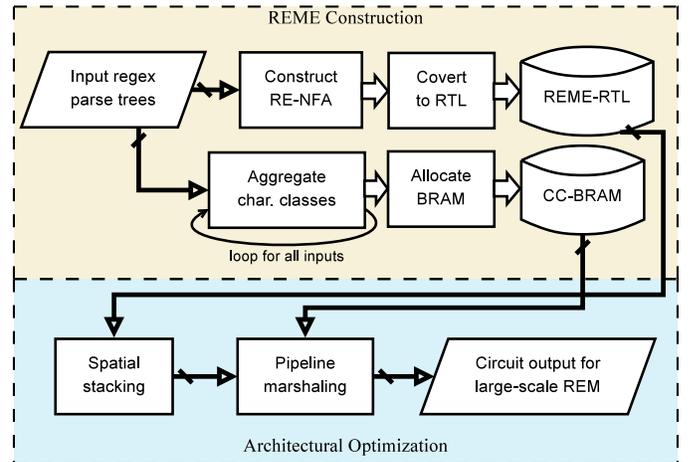

Figure 2. Overview of our toolchain for large-scale REME construction.

to be applicable to REM, only the construction of a fixed-string matching circuit was explained. The paper, however, did not describe an automatic mechanism to translate any general pattern into a multi-character matching circuit. An algorithm that extends any single-character matching REME *temporally* into a multi-character matching REME was proposed in [16]. In contrast, the uniform structure of the RE-NFA in [17] allows its circuit to be stacked *spatially* and automatically to process multiple characters per clock cycle.

III. OVERVIEW OF THE SOFTWARE TOOLCHAIN

The main purpose of our software toolchain is to automate the construction and optimization of large-scale RE-NFA circuits on FPGA. The toolchain allows us to generate the whole RTL circuit matching thousands of regular expressions in orders of seconds using a single command. Such a toolchain can help us not only to avoid the tedious and error-prone circuit construction, but also to generate a large-scale regular expression matching engine (REME) for implementation in a small amount of time.

Figure 2 gives an overview of the toolchain. The toolchain consists of two main parts: *REME Construction* and *Architectural Optimization*, briefly described as follows:

Table I
REM OPERATORS SUPPORT BY OUR SOFTWARE.

Op.	Name	Example	Description
-	Concatenation	$q_1 q_2$	q_2 right after q_1
|	Union	$q_1 \| q_2$	Either q_1 or q_2
*	Kleene closure	$q*$	q zero or more times
+	Repetition	$q+$	q one or more times
?	Optionality	$q?$	q zero or one times
$\{m, n\}$	Constrained rep.	$q\{m, n\}$	q in m to n times
[...]	Character class	$[a - c]$	Either a, b or c
[^...]	Inv. char. class	$[\wedge\backslash r\backslash n]$	Neither \r nor \n
^	Match beginning	$\hat{}q$	q at beginning of input
$	Match ending	$q\$$	q at ending of input

1) *REME Construction*: Converts each regular expression into a RE-NFA circuit and collects unique character classes in BRAM across all regular expressions.
2) *Architectural Optimization*: Applies spatial stacking to the individual RE-NFA circuits; marshals RE-NFAs into a 2-D staged pipeline to form the final circuit.

In practice, the two paths of REME Construction in Figure *2* are written as a single module interleaving the two tasks for each input regular expression. Conceptually, however, they are independent of each other and can be executed in parallel. In contrast, the two tasks in Architectural Optimization, *spatial stacking* and *pipeline marshaling* must be performed in serial. The details of the REME Construction part are presented in Section IV, while that of the Architectural Optimization part are in Section V.

In addition to the basic operators of concatenation, union (|) and Kleene closure (∗) used to define a regular language, our software also handles most frequently-used operators by the Snort IDS [13] such as the repetition (+), optionality (?), constrained repetition ($\{a, b\}$), and any character class ([. . .]). Table I lists the operators supported by our software. The syntax and semantics of these operators are compatible with the Perl-Compatible Regular Expression [1]. For example, the expression "$[0 − 9]\{1, 3\}(\backslash.[0 − 9]\{1, 3\})\{3\}[\hat{}0 − 9]?$" specifies any IP address followed by an optional non-numerical characters.

IV. AUTOMATIC REME CONSTRUCTION

The REME Construction is performed in three steps: (1) Parse the regular expressions into tree structures. (2) Use the *modified* McNaughton-Yamada (MMY) construction (Figure 4, Algorithm 1) to construct the RE-NFAs. (3) Map the RE-NFAs into structural VHDL suitable for FPGA implementation.

A. From Regular Expression to Parse Tree

The first step is to represent each regular expression as a corresponding parse tree using a standard compiler technique. This step is the same as that described in [8]. Figure 3 shows a parse-tree representation of a regular expression "$\backslash$x2F(fn|s)$\backslash$x3F[$\hat{}\backslash$r$\backslash$n] ∗ si." This is simplified for the value of illustration from an actual Snort pattern. In particular, a union of any number of single characters is parsed as a single
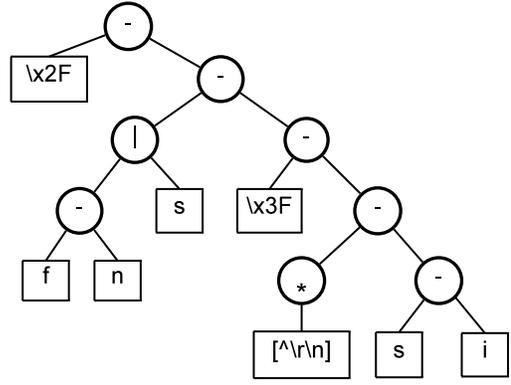

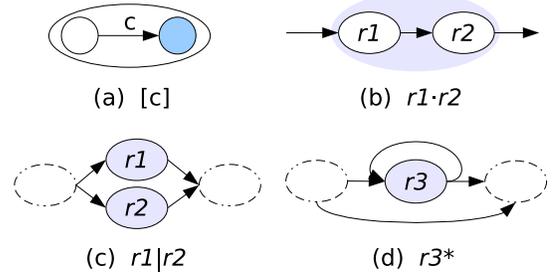Figure 3. Parse-tree representation of "\x2F(fn|s)\x3F[^\r\n] ∗ si."

Figure 4. Graphical representation of the modified McNaughton-Yamada (MMY) construction. Note that unlike the original construction, no ϵ-transition-only node is introduced in rules (c) and (d), where the dashed ellipses are *not* part of the current construction.

character class (*e.g.*, the [^\r\n] in Figure 3), which can be matched very efficiently in our REM architecture [17].

The resulting parse tree always consists of three types of internal nodes, op_concat, op_union and op_closure, and a number of leaf nodes equal to the number of individual (and possibly non-unique) character classes in the regular expression.

B. From Regular Expression Parse Tree to NFA

Unlike previous work in [8] and later in [14] which use the McNaughton-Yamada (MNY) construction to convert regular expressions into RE-NFAs, we proposed the *modified McNaughton-Yamada (MMY) construction* in [17] to perform the conversion. Figure 4 gives a graphical description of the modified construction rules.

A formal definition of the construction mechanism in given in Algorithm 1. The algorithm takes the regular expression parse tree generated from the previous subsection as input. It is in general a recursive algorithm, where the sub-trees of each internal node is processed recursively before the operator of the current node is handled. The only exception is the right child of an op_concat node, where for performance reason the tail recursion is performed iteratively. This avoids excessive recursion for a long sequence of op_concat operators (which is predominantly the case in real-world patterns).

Two special entities are used in Algorithm 1 for the MMY construction. The first is the set of immediate previous states $S_{\texttt{pre}}$, which contains the source states of all fan-in transitions

Algorithm 1 Modified McNaughton-Yamada construction (MMY): converting a regular expression parse-tree to a RE-NFA with a modular and uniform structure.

Notations:

$n\,[\texttt{value}]$ Content value of node n.

$n\,[\texttt{left|right|child}]$ Left, right, or only child of node n.

$s\,[\texttt{next}]$ Set of next-state transitions of state s.

$s\,[\texttt{char}]$ Set of matching characters of state s.

Macros:

$s \leftarrow \texttt{CREATE\_STATE}\,(T)$:

Create a new state s in the state transition table T.

$p \leftarrow \texttt{CREATE\_PSEUDO}()$:

Create a special pseudo-state p for later use.

$\texttt{ADD\_PSEUDO\_NEXT}\,(p, S)$:

For every state $s \in S$, add the state set $p\,[\texttt{next}]$ to $s\,[\texttt{next}]$. Pseudo-state p is deleted afterward.

PROCEDURE $S_{\text{out}} \Leftarrow \text{RE2NFA}\,(n_{\text{root}}, S_{\text{pre}}, T_{\text{NFA}})$

n_{root} Root node of the parse (sub-)tree.

S_{pre} Set of immediate previous states.

S_{out} Set of states transitioning directly outside of n_{root}.

T_{NFA} The resulting state transition table.

BEGIN

 $n_{\text{cur}} \leftarrow n_{\text{root}}$;

 while $n_{\text{cur}} \neq null$

  if $n_{\text{cur}}\,[\texttt{value}] = \texttt{OP\_CONCAT}$

   $S_{\text{pre}} \leftarrow$
   $\text{RE2NFA}\,(n_{\text{cur}}\,[\texttt{left}], S_{\text{pre}}, T_{\text{NFA}})$;

   $n_{\text{cur}} \leftarrow n_{\text{cur}}\,[\texttt{right}]$;

  else if $n_{\text{cur}}\,[\texttt{value}] = \texttt{OP\_UNION}$

   $S_L \leftarrow$
   $\text{RE2NFA}\,(n_{\text{cur}}\,[\texttt{left}], S_{\text{pre}}, T_{\text{NFA}})$;

   $S_R \leftarrow$
   $\text{RE2NFA}\,(n_{\text{cur}}\,[\texttt{right}], S_{\text{pre}}, T_{\text{NFA}})$;

   return $S_L \cup S_R$;

  else if $n_{\text{cur}}\,[\texttt{value}] = \texttt{OP\_CLOSURE}$

   $p \leftarrow \texttt{CREATE\_PSEUDO}()$;

   $S_{tmp} \leftarrow S_{\text{pre}} \cup p$;

   $S_C \leftarrow$
   $\text{RE2NFA}\,(n_{\text{cur}}\,[\texttt{child}], S_{tmp}, T_{\text{NFA}})$;

   $\texttt{ADD\_PSEUDO\_NEXT}\,(p, S_C)$;

   return $S_C \cup S_{\text{pre}}$;

  else   // $n_{\text{cur}} =$ leaf node

   $s_{\text{new}} \leftarrow \texttt{CREATE\_STATE}\,(T_{\text{NFA}})$;

   $s_{\text{new}}\,[\texttt{char}] \leftarrow n_{\text{cur}}\,[\texttt{value}]$;

   foreach s in S_{pre}

    // add ϵ-transitions

    $s\,[\texttt{next}] \leftarrow s\,[\texttt{next}] \cup s_{\text{new}}$;

   end foreach

   return s_{new};

  end if

 end while

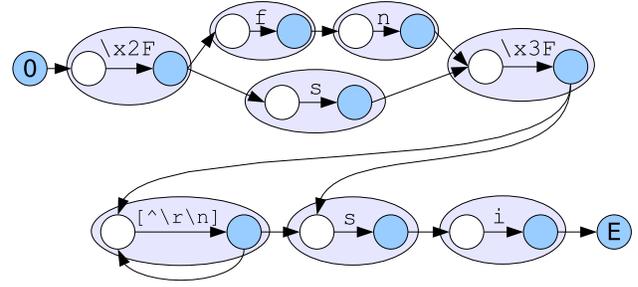 // error: $n_{\text{cur}}\,[\texttt{right}]$ cannot be $null$

END

Figure 5. A modular NFA for "\x2F(fn|s)\x3F[^\r\n]∗si" constructed using the MMY rules specified in Figure 4.
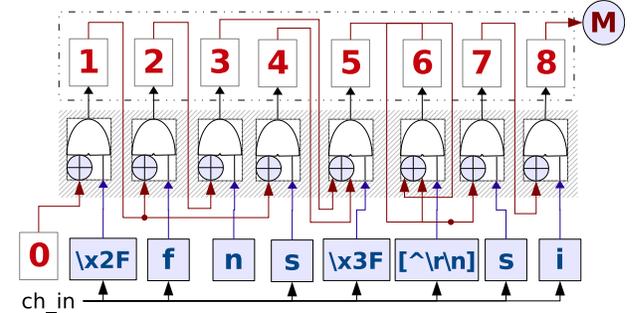

Figure 6. REM circuits constructed by mapping Figure 5 directly to HDL. The ⊕ symbols represent logic OR gates.

to the part of RE-NFA currently under construction. This entity corresponds to the dashed ellipses on the left of Figure 4 (c) and (d). It allows a long sequence of ϵ-transitions in the original MNY construction to be collapsed into a single ϵ-transition in the MMY construction.

The second entity is the pseudo state p, which works as a placeholder for the source states of an op_closure's feedback loop before the op_closure is converted to be part of the RE-NFA. This temporary placeholder is needed to break the circular dependence of an op_closure construction on the resulting fan-out states of the very op_closure construction.

The MMY construction algorithm produces an NFA extremely modular and easy to map to HDL codes. For example, using the modified construction algorithm, the regular expression "\x2F(fn|s)\x3F[^\r\n]∗si" is converted into a modular NFA with a uniform structure (Figure 5). This conversion is arguably the most complex part of the construction process, taking roughly 350 lines of C code for the automation.

C. From RE-NFA to VHDL

To translate the RE-NFA (like Figure 5) into VHDL, each pair of nodes inside a lightly shaded ellipse is mapped to an entity statebit with one parameter: the number of input ports, determined by the number of "previous states" that immediately transition to the current state. Inside the entity statebit, all inputs aggregate to a single OR gate, followed by a character matching via logic AND and a state value register. The single-bit output value of the register is connected to the inputs of the immediate "next states."

The REM circuit for Figure 5 is shown in Figure 6. On FPGA devices with 4-input LUTs, a k-input `OR` followed by a 2-input `AND` can be efficiently implemented on a single LUT if $k \leq 3$, or on a single slice of 2 LUTs if $4 \leq k \leq 7$. The mapping takes only about 300 lines of C code to convert any RE-NFA to its RTL structural VHDL description.

D. BRAM-based Character Classification

Our REM architecture in [17] used a 256-bit column of BRAM to match any character class of 8-bit characters. Each bit of the column represents the inclusion of a 8-bit character in the character set. The value of every input characters is used to as a row index to BRAM to retrieve the matching result ($true|false$) of that character against all character classes (one for each column). Each single-bit result is routed from BRAM to its corresponding correct `entity statebit` as the input to the `AND` gate. As a result, character classification of an n-state RE-NFA can be implemented on a block memory (BRAM) of no more than $256 \times n$ bits.

Furthermore, if two states (either within the same regular expression or across different regular expressions) match the same character class, then they can share the same BRAM column output. We use a two-phase procedure to aggregate the matching outputs of identical character classes:

- In phase 1, the software collects the set of unique character classes from a regular expression. Each unique character class is associated with a floating-point *sorting key*:
  - If the character class appears only once in the regular expression, then the sorting key is its (only) position index within the regular expression.
  - If the charactter class appears multiple times in the regular expression, then the sorting key is the average of all its position indexes within the regular expression.
- In phase 2, the unique character classes are sorted according to their sorting keys and instantiated as BRAM columns. Each BRAM column is also associated with the identifier of the instantiated character class. The output of each BRAM column is then connected to the character matching inputs with the same identifier.

The two-phase procedure allows our software to use the minimum number of BRAM columns for character class matching. It also minimizes routing distance by exploiting the natural ordering (the sorting keys) of the character classes within the regular expressions. The aggregation of character classes and their distribution to the RE-NFA states take ~250 lines of C code.

V. AUTOMATED ARCHITECTURAL OPTIMIZATIONS

After constructing REMEs individually for all regular expressions, the software applies two architectural optimizations [17]: (1) The REMEs are *stacked* to form multi-character matching (MCM) circuits which trade off minimum resource usage for higher performance. (2) The MCM REMEs are grouped into clusters of 16 and marshaled onto a two-dimensional *staged pipeline* structure.
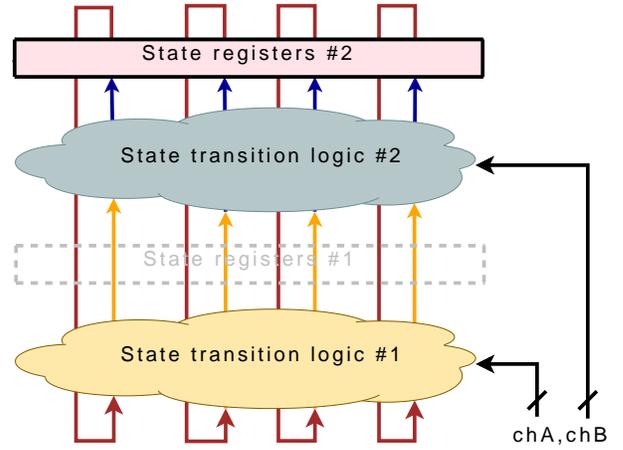

Figure 7. The construction of a 2-character matching circuit.

A. Circuit Stacking for Multi-Character Matching

In contrast to the NFA-level *temporal extension* used in [16], we adopted a circuit-level *spatial stacking* to construct multi-character matching (MCM) REMEs. Figure 6 shows the basic construction concept of a 2-character matching circuit from two copies of a single-character matching circuit. An algorithm for this spatial stacking approach and the proof of correctness were given in [17]. Benefits of the spatial stacking approach include the following:

Simplicity   The time complexity to construct an n-state, m-character matching REME using spatial stacking is $O(n \times m)$ [17]. In contrast, the time complexity of temporal extension is $O(n^3 \log m)$ [16].

Flexibility   The spatial stacking approach can generate an MCM REME of any natural number m, while the temporal extension approach only generates RE-NFAs with $m = 2^i$.

In practice, n is usually a few tens while m between 2 to 8, making the spatial stacking approach hundreds of times faster than the temporal extension approach. As discussed in Section VI-B, our software can construct thousands of MCM REMEs in ~10 seconds. Also, the optimal value of m with respect to *performance efficiency* (defined in [17]) is usually not a power of 2. For example, the REMEs from Snort rules achieve optimal performance efficiency at $m = 6$ [17].

The program code to construct any m-character matching REME using spatial stacking is simple. Let C be a single-character matching circuit. The program first makes m copies of C, $\left\{C^{(1)}, \ldots, C^{(m)}\right\}$, each receiving one of the m consecutive input characters. Then, instead of routing the state outputs back to the state inputs of the same circuit, it removes the state registers of $C^{(k)}$ and connects the (non-registered) state outputs of $C^{(k)}$ to the state inputs of $C^{(k+1)}$ for $k = 1, \ldots m - 1$. Finally, it connects the (registered) state outputs of $C^{(m)}$ to the state inputs of $C^{(1)}$. The result is an m-character matching circuit for C.

In general, to construct an $(l + m)$-character matching circuit C_{l+m}, we perform the following transformations on every state $i \in \{1, 2, \ldots, n-1\}$ of C_l and C_m:
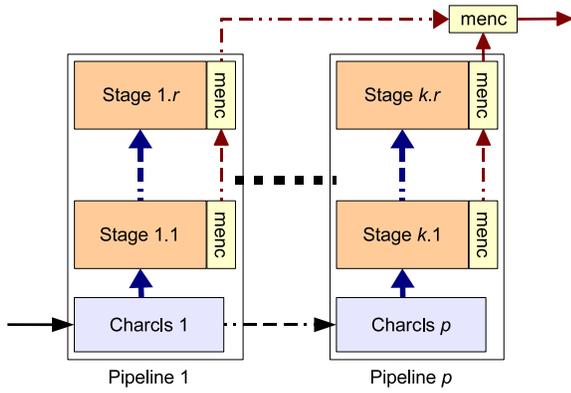
Figure 8. Structure of a 2-D staged pipeline with total p pipelines and r stages per pipeline.

Figure 9. Structure of the regular expressions from the benchmark generator.

1) Remove state register i of C_l; forward the `AND` gate output to its state output.
2) Disconnect state output i of C_l from the state inputs of C_l, and re-connect it to the corresponding state inputs of C_m.
3) Disconnect state output i of C_m from the state inputs of C_l, and re-connect it to the corresponding state inputs of C_l.
4) The combined circuit receives $(l + m)$ character matching signals per cycle. The first l signals are sent to the C_l part; the last m signals are sent to the C_p part.

B. REME Clustering for Staged Pipelining

With a straight-forward implementation, the BRAM-based character classifier (Section IV-D) uses 256 bits per state. To implement thousands of REMEs with tens of thousands states, the character classifier would require tens of megabits of BRAM and become the resource bottleneck on FPGA. A second issue in implementing large number of REMEs on FPGA is signal routing. The character matching results from the centralized character classifier in BRAM must be distributed to all REMEs, while the pattern matching result from every REME must be collected and aggregated to the final output. The potentially long routing makes the circuit hard to scale to large number of REMEs.

A *2-D staged pipeline* design was proposed in [17] to solve both problems. Figure 8 shows the basic structure of such a staged pipeline. Each stage may contain a cluster of up to 16 REMEs. The horizontal arrows between the pipelines are the signal paths of the input characters. The vertical arrows between pipeline stages are the character matching signals and the pattern matching results. A priority encoder is used at every stage and pipeline to aggregate the pattern matching results.

Marshaling REMEs into this staged pipeline structure, however, is painstaking and error-prone when done manually. This is mainly due to the buffering and distribution of the character matching signals (the thick vertical arrows in Figure 8). Additionally, different REME grouping can result in different resou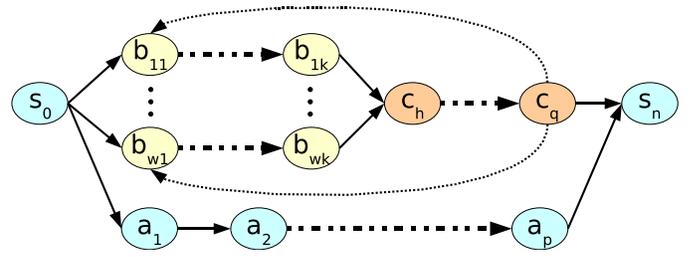rce usage and routing complexity and give rise to performance variation among REME clusters. To solve these problems, our software use the following heuristic to marshal k REMEs with total N states into p pipelines:

1) First calculate the average number of states per pipeline, $v = N/p$.
2) Add any of the k REMEs into a new pipeline. Compute the *compatibility* between the resulting (single-REME) pipeline and each of the rest $k - 1$ REMEs. The *compatibility* between a pipeline and an REME is defined as the number identical character classes in both divided by the number of unique character classes in the REME.
3) Add the most compatible REME to the pipeline. Re-compute the compatibility of all remaining REMEs.
4) Repeat Step 3 until the total number of states in the pipeline is greater than $v - \sigma$, where $\sigma \ll v$ is a design constant.
5) Go back to Step 2 to work on a new pipeline until all REMEs are exhausted.

After marshaling the REMEs into different pipelines, the REMEs within each pipeline are marshaled into different stages in a similar manner. When adding a REME to a pipeline, a function is called to compare each of the character class in the REME to the character classes previously collected in BRAM. If an identical character class is found, then proper connections are made from the BRAM output to the inputs of the respective states.

The time complexity of this procedure is $O(k \times N \times w)$, where w is the number of *distinct* character classes among the N states in the k REMEs. The space complexity is $O(256 \times w)$. In real applications, w grows almost linearly with respect to N for small N, but quickly flats out and grows much slower than $O(\log N)$ when N is moderately large (a few hundred).

Matching outputs from all REMEs are prioritized. Currently, the software assigns higher priority to lower-indexed pipelines and stages, although the priority can be programmed in any other way with little additional complexity.

VI. EXPERIMENTAL RESULTS

A. Design of Benchmark Generator

We developed a regular expression *benchmark generator* to test how different types of regular expressions affect the performance of the REMEs constructed by our software. The benchmark generator produced regular expressions of different
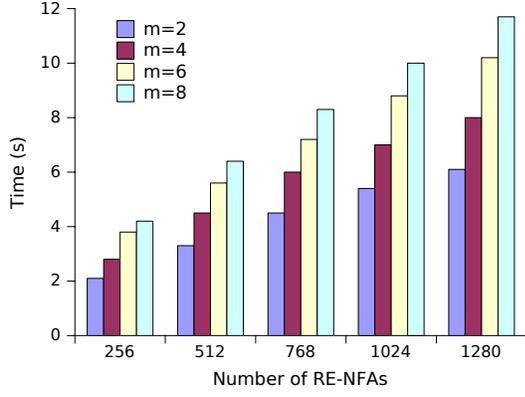
Figure 10. REME construction time for various number of regular expressions and multi-character matching parameters.
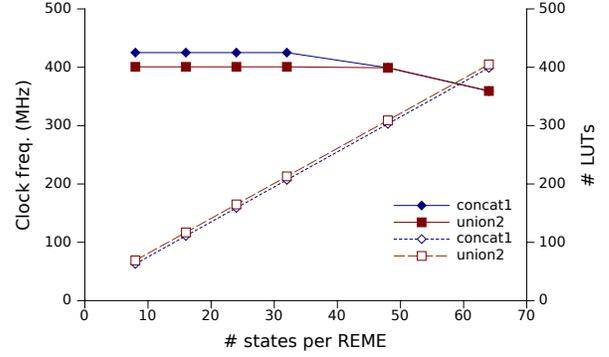

Figure 11. Clock frequency and LUT usage of group of 6 identical synthetic REMEs versus length of every REME. Solid lines (left scale) are clock frequencies; dashed lines (right scale) are number of LUTs.

state count (n), *state fan-in* (w), and variable lengths of *loop-back* (q) and *feed-forward* ($q - p$). A general structure of the generated regular expressions is described in Figure 9.[1]

State count represents the total number of states in an RE-NFA. It was used by most related work as the primary metric for REME complexity [5], [9], [13], [16]. We further defined *state fan-in* as the *maximum* number of transitions entering any state [17], since the state machine runs at the speed of the slowest state transition. Both op_union and op_closure can increase state fan-in, which is the secondary metric for REME complexity.

A state transition *loop-back* is always caused by an op_closure, while a state transition *feed-forward* can be caused by unbalanced alternative paths within an op_union. Both properties are high-order metrics describing the routing lengths of a REME. According to our experimental experience, however, the actual routing complexity of the REME circuit on FPGA is highly subject to the optimizations done by the place and route software and may not reflect these two metrics closely.

B. Performance Evaluation of the Software Toolchain

The time taken to translate a set of parsed regular expressions to VHDL was roughly proportional to the product of the *number of states* (n) and the *size of multi-character input* (m), an observation agreeing with our analysis in Section V-A. On a 2 GHz Athlon 64 PC, it took between 6 and 12 seconds to translate 1280 Snort REMEs (~28k states) to VHDL, as m increased from 2 to 8. In all cases, about 30% of the time was used for file I/O. Figure 10 illustrates the construction time of various cases in more detail.[2]

---

[1]Due to our use of BRAM for character classification, every character class, no matter how simple or complicated it is, takes exactly 256 BRAM bits and is matched by one BRAM access. Since the complexity of character classes does not affect performance, our benchmark generator assigns arbitrary values to the character classes without loss of generality.

[2]Due to the relatively large I/O overhead and the short overall runtime, there is high variance (~15%) among different runs of the same construction. The construction time is also greatly affected by the complexity of regular expressions, especially the *state count* and the *state fan-in* discussed in Section VI-A.
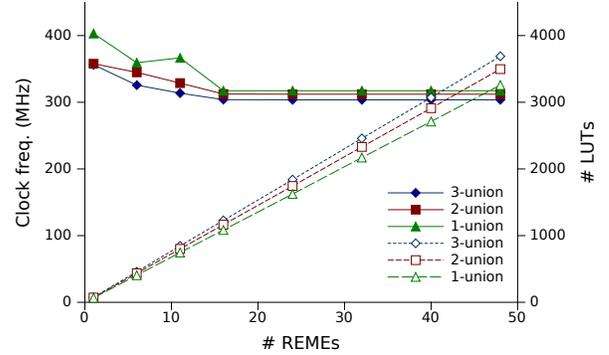

Figure 12. Clock frequency and LUT usage of group of 64-state synthetic REMEs versus number of REMEs implemented. Solid lines (left scale) are clock frequencies; dashed lines (right scale) are number of LUTs.

These results show that the software proposed in this paper is suitable for large-scale REME construction. Since it takes only a few seconds to translate a thousand regular expressions into structural VHDL, the software can be used to reconstruct a large-scale REME quickly in response to dictionary changes. Due to the large number of logic resource used, however, the synthesis and place & route times are in the order of several tens minutes.

C. Performance Evaluation of the Constructed REMEs

We first used the benchmark generator described in Section VI-A to produce synthetic regular expressions of different numbers and complexities, then use our REME construction software to convert the synthetic regular expressions into 2-character matching REME circuits in VHDL. We synthesized the VHDL into Xilinx NGC targeting the Virtex 4 LX device family, and extracted the estimated clock frequency from the timing analysis.

Figure 11 shows clock frequency and LUT usage versus length of REMEs. Series concat1 was produced by one long sequence of concatenations. Series union2 was produced by a union of two equal-length concatenations. In each test case, 6 identical REMEs were placed into a single stage.
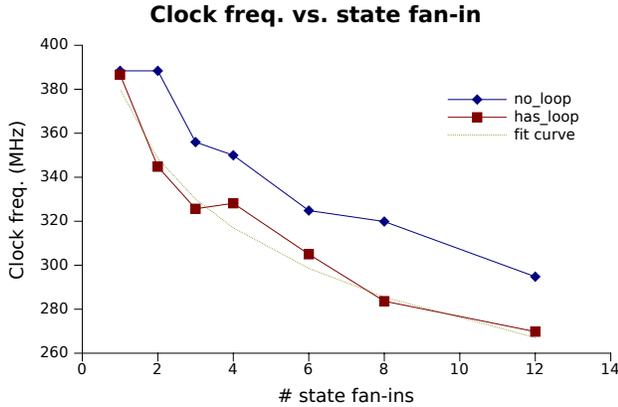
Figure 13. Clock frequency versus state fan-in of the synthetic REMEs.

Series `union2` ran at lower clock frequency than series `concat1` due to the use of the `op_union` operator, which caused series `union2` to have twice the (maximum) state fan-in as `concat1`. The clock rates of both series started to decline gradually with respect to REME length around 32 to 40 states per REME. This decline was due to the longer paths to access the centralized character classification signals from BRAM. This is evidenced by the fact that both `concat1` and `union2` ran at about the same clock rates beyond the length of 40 states, showing a bottleneck elsewhere from the state transitions within the logic slices of FPGA.

In Figure 12, we analyzed the effect of the number of REMEs on achievable clock frequency and total LUT usage. In each test case, 64 states were generated for each REME; 30 states were wrapped inside an `op_closure` ($q = 30$), which was then `op_union`-ed with a sequence of 30 other states ($p = 30$) and concatenated with the last 4 states in sequence. In the $w-$union series, $w = 1$, 2, or 3, the 30 states inside the `op_closure` were further wrapped by an `op_union` of w operands, each $30/w$ states in length. The purpose was to see how clock rate scaled with respect to number of REMEs for different REME structures and complexities.

As shown in Figure 12, clock frequency declined between 15% to 25% when number of REMEs varied from 1 to 16. All these 16 REMEs are put inside a single stage by our software. Since the added regular expressions were all identical, this decline was again due to longer BRAM access, caused by both longer routes and larger fan-out.

Above 16 REMEs, however, the staged pipeline came into effect, keeping the clock rates at slightly above 300 MHz. This evidently shows that the staged pipeline proposed in [17] was effective in scaling up number of REMEs in a single circuit. LUT usage maintained linear increase with respect to the number of REMEs.

As expected, a higher w value results in a slightly lower achievable clock frequency due to the higher state fan-in of the REMEs.

Figure 13 examines clock frequency versus state fan-in more thoroughly. In each test case, REMEs of 52 states were constructed, with 24 states put inside an `op_union` of w operands, w varying from 1 (single 24-state sequence) to 12 (union of 2-state sequences). For the `has_loop` series, there

was also a loop-back transition from the outputs of the 24-state `union` back to the inputs of the `union` itself. There was no such loop-back for the `no_loop` series.

The clock frequency was found to decline sub-linearly with respect to the state fan-in, at a rate consistent with the findings in Section VI-B. The decline however was not completely smooth because the logic gates on the FPGA device were organized as 4-input LUTs - fan-ins of size multiples of 4 tend to perform better than the overall trend. The loop-back transition around the `op_union` (in the `has_loop` series) connected every state output of the union operator to every input state of that operator. This resulted in more complex routing and further impacted the clock frequency.

Overall our experiments show that the REME construction algorithms proposed in [17] generated FPGA circuits with high clock frequency and high LUT efficiency for large number of highly complex regular expressions.

VII. CONCLUSIONS

We presented a software toolchain which automates the construction and optimizations of regular expression matching engines (REMEs) on FPGA. The software accepts a potentially large number of regular expressions as input and generates RTL codes in VHDL as output, which could be accepted directly by FPGA synthesis and implementation tools. The automated REME optimizations include centralized character classifications, multi-character matching, and staged pipelining. We also developed a benchmark generator to produce REMEs of configurable pattern complexities to evaluate the performance of the software.

On a 2 GHz Athlon 64 PC, our software generates a compact and high-performance REME circuit matching over a thousand regular expressions in just a few seconds. Extensive studies showed that the two-dimensional staged pipeline effectively localized signal routing and achieved a clock rate over 300 MHz while processing hundreds of REMEs in parallel.

REFERENCES

[1] PCRE: Perl Compatible Regular Expression. http://www.pcre.org/.

[2] Snort network instrusion detection. http://www.snort.org.

[3] BECCHI, M., AND CROWLEY, P. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *ACM CoNEXT* (2007), pp. 1–12.

[4] BECCHI, M., AND CROWLEY, P. An Improved Algorithm to Accelerate Regular Expression Evaluation. In *Proc. of 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)* (2007), pp. 145–154.

[5] BISPO, J., SOURDIS, I., CARDOSO, J. M. P., AND VASSILIADIS, S. Regular expression matching for reconfigurable packet inspection. In *Proc. of IEEE International Conference on Field Programmable Technology (FPT)* (December 2006), pp. 119–126.

[6] BRODIE, B. C., TAYLOR, D. E., AND CYTRON, R. K. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. *SIGARCH Computer Architecture News 34*, 2 (2006), 191–202.

[7] CLARK, C., AND SCHIMMEL, D. Scalable pattern matching for high speed networks. In *Proc. of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (April 2004), pp. 249–257.

[8] FLOYD, R. W., AND ULLMAN, J. D. The Compilation of Regular Expressions into Integrated Circuits. *Journal of ACM 29*, 3 (1982), 603–622.

[9] HUTCHINGS, B. L., FRANKLIN, R., AND CARVER, D. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proc. of 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2002), p. 111.

[10] KUMAR, S., CHANDRASEKARAN, B., TURNER, J., AND VARGHESE, G. Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In *Proc. of 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)* (2007), pp. 155–164.

[11] KUMAR, S., DHARMAPURIKAR, S., YU, F., CROWLEY, P., AND TURNER, J. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. *SIGCOMM Computer Communication Review 36*, 4 (2006), 339–350.

[12] LIN, C.-H., HUANG, C.-T., JIANG, C.-P., AND CHANG, S.-C. Optimization of Regular Expression Pattern Matching Circuits on FPGA. In *Proc. of Conference on Design, Automation and Test in Europe (DATE)* (3001 Leuven, Belgium, Belgium, 2006), European Design and Automation Association, pp. 12–17.

[13] MITRA, A., NAJJAR, W., AND BHUYAN, L. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proc. of 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)* (New York, NY, USA, 2007), pp. 127–136.

[14] SIDHU, R., AND PRASANNA, V. Fast Regular Expression Matching Using FPGAs. In *Proc. of 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (2001), pp. 227–238.

[15] SMITH, R., ESTAN, C., AND KONG, S. J. S. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *ACM SIGCOMM* (August 2008).

[16] YAMAGAKI, N., SIDHU, R., AND KAMIYA, S. High-Speed Regular Expression Matching Engine Using Multi-Character NFA. In *Proc. of International Conference on Field Programmable Logic and Applications (FPL)* (Aug. 2008), pp. 697–701.

[17] YANG, Y.-H. E., JIANG, W., AND PRASANNA, V. K. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *Proc. of 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (November 2008).

[18] YU, F., CHEN, Z., DIAO, Y., LAKSHMAN, T. V., AND KATZ, R. H. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *Proc. of 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)* (2006), pp. 93–102.