2008 International Conference on Reconfigurable Computing and FPGAs

Automatic Construction of Large-Scale Regular Expression Matching Engines 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 algorithms for implementing large-scale *regular expression matching* (REM) on FPGA. Based on the proposed algorithms, we develop tools that first transform regular expressions into corresponding non-deterministic finite automata (RE-NFA), then convert the RE-NFA into structural VHDL that utilize both logic slices and block memory (BRAM) available on modern FPGA devices. An n-state m-character input regular expression matching engine (REME) can be constructed in $O(n \times m \log_2 m)$ time using $O(n \times m)$ memory space, resulting in a circuit that occupies no more than $O(n \times m)$ slices on FPGA. A large number of REMEs are placed automatically onto a two-dimensional *staged pipeline*, allowing scalability to hundreds of REMEs with linear area increase, running at over 300 MHz on Xilinx Virtex 4 devices.**

***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, a regular expression defines a regular language over a fixed alphabet for the character sets, and offers three basic operators to bind the character sets together: *concatenation* ($\cdot$), *union* ($|$), and *Kleene closure* ($*$). There are other common operators that also conform to the regular language construct, such as *character classes* ($[\ldots]$), *optionality* ($?$) and *constrained repetitions* ($\{\mathtt{a},\}$, $\{,\mathtt{b}\}$, $\{\mathtt{a},\mathtt{b}\}$). 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], [2], [11], [9], [1], [7], [5]. Since regular languages can be sufficiently and necessarily 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).

In an RE-NFA approach [4], [10], 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 [6], common-prefix extraction [6], [2], multi-character input, and centralized character decoding [3], [2], can be applied to improve throughput and reduce resource requirements of the overall design.

In an RE-DFA approach, several regular expressions are grouped [11] into a DFA by expanding different combinations of active states into new *combined states*. In principle, only *one combined state* in an RE-DFA is active at any time. Various techniques [5], [8], [7], [1] are then applied to improve memory access efficiency and to reduce the total number of states, which usually suffers from quadratic to exponential explosion [11].

Due to the matching power of regular expression and the complexity of the strings being matched, an REM can be the slowest path of a system. A REM 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) [11]. Furthermore, to process K concurrent REM patterns, 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.

In this study we focus on the automatic construction of large-scale REM on FPGA using the RE-NFA approach. We develop programs which translate and compile a large number of regular expression matching engines (REMEs) into an area-efficient, high-performance circuit, described as structural VHDL for FPGA implementation.

The rest of this paper is organized as follows. We discuss the background and prior work of RE-NFA on FPGA in Section 2. In Section 3 we describe our basic RE-NFA architecture and the algorithms we use to construct the large-scale REME on FPGA. Section 4 explains the architectural optimizations used by our REME construction tools, while Section 5 discusses the achieved performance. Section 6 concludes the papers.

II. Related Work

Hardware implementation of regular expression matching (REM) was first studied by Floyd and Ullman [4]. They showed that an n-state RE-NFA can be translated into integrated circuits using no more than $O(n)$ circuit area. Sidhu and Prasanna [10] later proposed an algorithm and strategy to implement REM on FPGA in a similar RE-NFA architecture, which has been used by most other RE-NFA implementations on FPGA ([6], [3], [2], [9]).

Automatic REME construction on FPGA was first proposed in [6] using JHDL for both regular expression parsing and circuit generation. In particular, the HDL *construction* approach used in [6] is in contrast to the *self-configuration* approach done by [10]. Reference [6] also considered large-scale REM construction, where the character input is broadcasted globally to all states in a tree-structured pipeline.

Supported by U.S. National Science Foundation under grant CCR-0702784.

978-0-7695-3474-9/08 $25.00 © 2008 IEEE
DOI 10.1109/ReConFig.2008.47

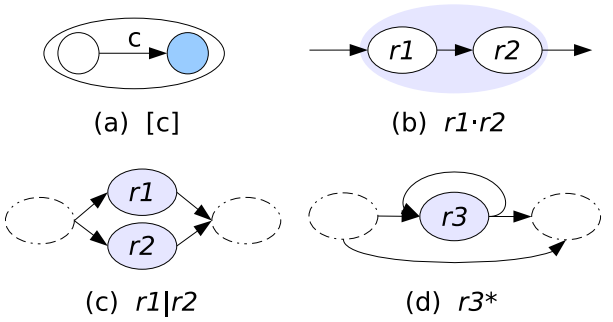

Fig. 1. Modified McNaughton-Yamada construction. Note that in rules (c) and (d), the dashed ellipses are *not* part of the current construction.

Automatic REME constructions in VHDL were proposed in both [2] and [9]. In [2], the regular expression was first tokenized and parsed into a hierarchy, then converted to VHDL in a bottom-up scheme. In [9], a set of scripts were used to compile regular expressions into op-codes, to convert op-codes into NFA, and to construct the NFA circuit in VHDL.

A multi-character decoder was proposed in [3] to improve REM throughput. It was also observed that character matching could be performed more efficiently at a centralized location. The paper, however, fell short to describe an automatic mechanism to translate any general regular expression into a multi-character input circuit.

III. Automatic REME Construction

We implement REM on FPGA in three steps: (1) Parse the regular expressions into tree structures. (2) Use the *modified* McNaughton-Yamada construction (Figure 1, Algorithm 1) to construct the RE-NFAs. (3) Map the RE-NFAs into structural VHDL suitable for FPGA implementation.

Our first step is the same as that described in [4]. Steps (2) and (3) are explained in the following subsections.

A. From Regular Expression to NFA

Unlike in [4], we use a *modified* McNaughton-Yamada construction to construct our RE-NFA state machines. Figure 1 shows the graphical description of the modified construction rules. The modified construction is described in Algorithm 1. Note that the modified McNaughton-Yamada algorithm produces no extra nodes nor ϵ-transitions for the `union` and `closure` operators (see Algorithm 1 or Figure 1 (c) and (d)). This makes the resulting NFA extremely modular and easy to map to HDL codes.

For example, using the modified construction algorithm, the regular expression "$\backslash$x2F(f|s)$\backslash$?[^$\backslash$r$\backslash$n]*si" is converted into a modular NFA with a uniform structure (Figure 2). This conversion is arguably the most complex part of the construction process, taking roughly 350 lines of C code for the automation.

B. From RE-NFA to VHDL

To translate the RE-NFA (like Figure 2) into VHDL, each pair of nodes inside a lightly shaded ellipse is mapped to an

Algorithm 1 The *modified* McNaughton-Yamada construction that converts a regular expression parse tree to a modular RE-NFA.

Global data:

$T_{\texttt{NFA}}$ The resulting state transition table.

Conventions:

$n\,[\texttt{value}]$ Content value of node n.
$n\,[\texttt{left}|\texttt{right}|\texttt{child}]$ 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:

$s \leftarrow$ `CREATE_STATE` (T) :
Create a new state s in the state transition table T.

$p \leftarrow$ `CREATE_PSEUDO`() :
Create a special pseudo-state p for later use.

`ADD_PSEUDO_NEXT` (p, S) :
For every state $s \in S$, add $p\,[\texttt{next}]$ to $s\,[\texttt{next}]$. Pseudo-state p is deleted afterward.

PROCEDURE $S_{\texttt{out}} \Leftarrow \texttt{RE2NFA}\left(n_{\texttt{root}}, S_{\texttt{pre}}\right)$

$n_{\texttt{root}}$ Root node of the parse (sub-)tree.
$S_{\texttt{pre}}$ Set of immediate previous states.
$S_{\texttt{out}}$ Set of states transitioning directly outside of $n_{\texttt{root}}$.

BEGIN

```
n_cur ← n_root;
while n_cur ≠ null
  if n_cur[value] = OP_CONCAT
    S_pre ⇐ RE2NFA(n_cur[left], S_pre);
    n_cur ← n_cur[right];
  else if n_cur[value] = OP_UNION
    S_L ⇐ RE2NFA(n_cur[left], S_pre);
    S_R ⇐ RE2NFA(n_cur[right], S_pre);
    return S_L ∪ S_R;
  else if n_cur[value] = OP_CLOSURE
    p ← CREATE_PSEUDO();
    S_tmp ← S_pre ∪ p;
    S_C ⇐ RE2NFA(n_cur[child], S_tmp);
    ADD_PSEUDO_NEXT(p, S_C);
    return S_C ∪ S_pre;
  else  // n_cur = leaf node
    s_new ← CREATE_STATE(T_NFA);
    s_new[char] ← n_cur[value];
    foreach s in S_pre

      // add epsilon transitions
      s[next] ← s[next] ∪ s_new
    end foreach
    return s_new
  end if
end while
// error: n_cur[right] cannot be null
```

END

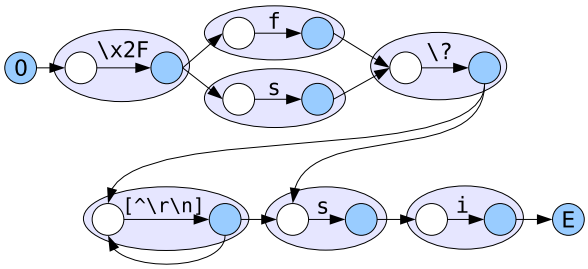

Fig. 2. A modular NFA for "`\x2F(f|s)\?[^\r\n]*si`" constructed using the modified McNaughton-Yamada rules specified in Figure 1.

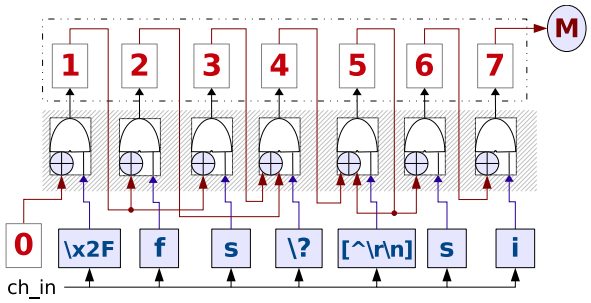

Fig. 3. Circuits for matching "`\x2F(f|s)\?[^\r\n]*si`" constructed by mapping Figure 2 directly to HDL. The $\oplus$ symbol represent the logic `OR` gate.

`entity statebit` in VHDL 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 a logic `AND` and a state value register. A single output port connects the output value of the register to the inputs of the immediate "next states."

The REM circuit for Figure 2 is shown in Figure 3. On FPGA devices with 4-input LUTs, a k-input `OR` followed by the 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* such RE-NFA to its corresponding VHDL.[1]

C. BRAM-based Character Classification

In our BRAM-based character classification, each 8-bit character classification is specified by a 256-bit vector, with every bit member representing the inclusion of one character value. The character matching result $(true|false)$ is sent to the `entity statebit` as a single bit from the 256-bit vector indexed by the value of the input character.

If two states match the same character class, they can share the same character classification output. 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. The aggregation of character classes and their distribution to every REME takes 200 lines of C code.

[1]This includes the instantiation of BRAM-based character classification (bottom part of Figure 3, discussed in Section III-C) *and* the m-character input extension (Section IV-A). The contents of the BRAM required for character classification are specified in separate data files.

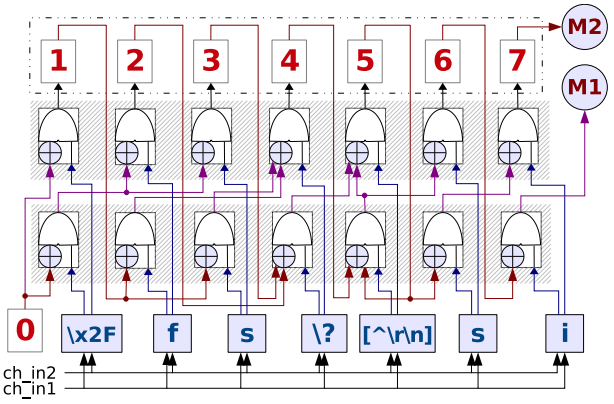

Fig. 4. A 2-input matching circuit for the regular expression "`\x2F(f|s)\?[^\r\n]*si`".

IV. Architectural Optimizations

We apply three optimizations to improve our basic design: (1) *multi-character input matching*, (2) *centralized character classification*, and (3) *staged pipelining*. These architectural enhancements take advantage of the uniform structure of our design and are fully integrated to the automatic translation tools that we developed.

A. Multi-Character Input Matching

We adopt a circuit-level *spatial* approach to construct multi-character input REMEs. Let C_l and C_m be an l-character and m-character input circuit, respectively, for the same RE-NFA. To construct the $(l+m)$-character input circuit C_{l+m}, we perform the following transformations on every state $i \in \{1, 2, \ldots, n-1\}$ of both C_l and C_m:

1) Remove the 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 the 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.

Each application of the procedure requires $O(n)$ time to build an n-state m-character input REME, resulting in a circuit of $O(n \times m)$ area. Recursively, an n-state m-character input REME can be constructed in $O(n \times \log_2 m)$ time, starting from the one-character input REME. A two-character input REME for the circuit of Figure 3 is shown in Figure 4. Note that the `START` and `MATCH` states must be merged and aggregated, respectively.

B. Staged Pipelining with Centralized Character Classification

With a straight forward implementation, the BRAM-based character classification (Section III-C) would incur much memory redundancy and become the resource bottleneck for large

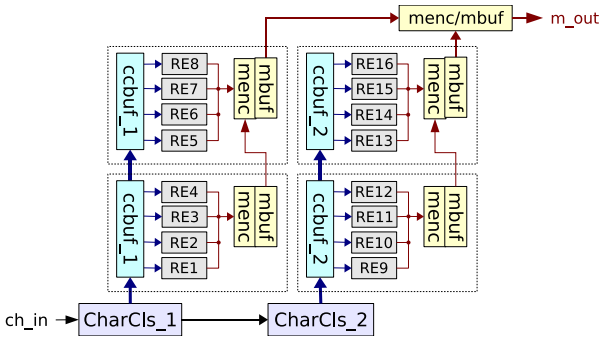

Fig. 5. Structure of a staged pipeline for 16 different REMEs.

numbers of REMEs. We solve this problem by aggregating character classifications of different REMEs into one centralized location.

When constructing the centralized character classification, a function is called to compare the character class of every state to the character classes previously collected in BRAM. Only newly seen character classes are added as new entries (column of 256 bits) in BRAM. For previously seen character classes, proper connections are made from their BRAM outputs to the inputs of the respective states. The time complexity of this procedure is $O(n \times w)$, where n is the total number of states in *all* REMEs and w is the number of *distinct* character classes among the n states. The space complexity is just $256 \times w$. We find that with a prudent grouping of REMEs, w tends to grow much more slowly than $O(\log n)$ in practice.

When grouping a large number of REMEs together on FPGA, the achievable clock frequency tends to decline due to the higher fan-outs and more complicated routing. We design a 2-D *staged pipeline* to improve the scalability of our circuit. An example of 16 REMEs in 2 pipelines of 2 stages per pipeline is shown in Figure 5. Every input character is sent to the first pipeline in one clock cycle, and forwarded to the next pipeline in the next clock cycle. Within a pipeline, all the REMEs share the same centralized character classification, whose output is buffered and forwarded through all stages in the pipeline.

Matching outputs of all REMEs are prioritized, with lower-indexed pipelines and stages having higher priority. Within a stage, matching outputs from different REMEs are priority-encoded, buffered and forwarded to the next stages in the same way as the input character classifications.

V. Experimental Results

A. Performance Benchmark

We developed a regular expression *benchmark generator* to test how different types of regular expressions affect performance of the REMEs constructed in our architecture. The generator produced regular expressions of different *state count* (n), *state fan-in* (w), and variable lengths of *loop-back* (k) and *feed-forward* ($k-p$). A general structure of the generated

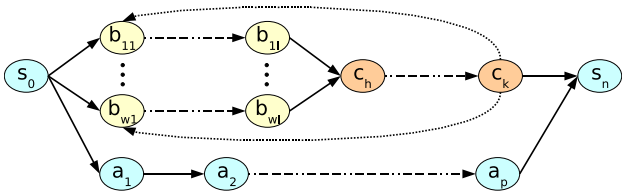

Fig. 6. Structure of the regular expressions from the benchmark generator.

regular expressions is described in Figure 6.[2]

State count represents the total number of states in an RE-NFA. It was used in [4] to describe the area requirement of a REME with respect to the length of the regular expression. We defined *state fan-in* as the *maximum* number of transitions entering any state, since the state machine runs at the speed of the slowest state transition. State transitions *loop-back* and *feed-forward*, caused by `closure` operators, affected routing complexity and delay.

B. Construction of Snort REMEs

Figure 7 shows the performance of 267 REMEs extracted from Snort rules with medium lengths (8-64 states) and complexity. There were total 6551 states, averaged 24.5 per REME. Most of the REMEs had state fan-in around 2 or 3, with a few going as high as 8.

In each test case, a circuit with 267 REMEs was synthesized, placed and routed by Xilinx ISE 10.1 on a Xilinx Virtex 4 LX-40-12 device. The reported achievable clock frequency was used to estimate the throughput. As m increased, resource usage increased almost linearly. However, due to the lower clock frequencies of the circuits, throughput increased sub-linearly. BRAM usage increased linearly with every two increments of m, since every two input characters require a dedicated dual-port BRAM block to produce the character matches. Due to the use of centralized character classification, BRAM was always underutilized and never a resource constraint.

In Figure 8 we examined the relationships between the number of inputs/cycle (m) and the resource usage versus achieved clock frequency. The curve for the clock frequency was well approximated by $440\,\text{MHz}/(1+0.10\log_4 m+0.20m)$. The clock frequency for $m=1$ was limited by the BRAM access and was excluded from the regression. It can be inferred that the "logic only" frequency without BRAM access is roughly $440/(1+0.2)=367\,\text{MHz}$. This value coincides well with the maximum frequency achieved in [2] (362 MHz) on the same device technology (Virtex4-40-12).

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 IV-A. On a 2 GHz Athlon 64 PC, it took 3 to 6 seconds to translate 768

[2]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.

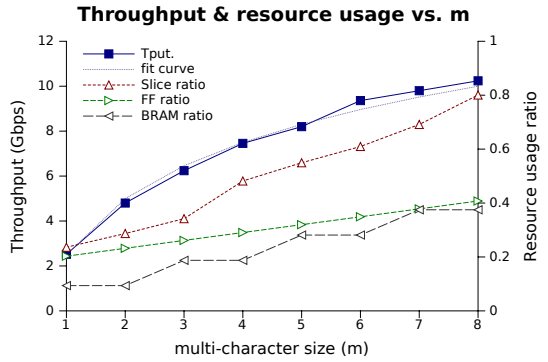

Fig. 7. Throughput of 267 Snort REMEs on Virtex 4 LX-40-12 vs. different m-character input sizes. Squares (left scale) are throughput; triangles (right scale) are resource usage ratios.

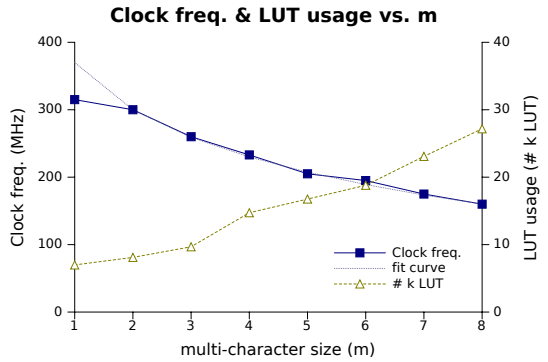

Fig. 8. Clock rate and LUT usage of 267 Snort REMEs on Virtex 4 LX-40-12. Squares (left scale) are clock rate; trangles (right scale) are LUT usage in thousands.

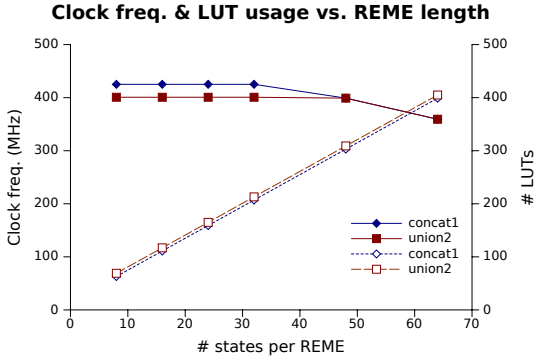

Fig. 9. Clock frequency and LUT usage of group of 6 identical synthetic REMEs versus length of every REME. Solid lines (left scale) are clock frequencies; dashed lines (right scale) are number of LUTs.

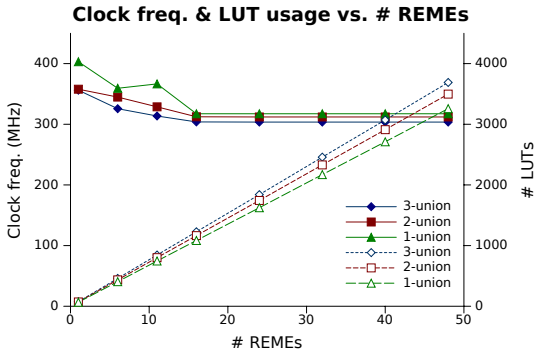

Fig. 10. Clock frequency and LUT usage of group of 64-state synthetic REMEs versus number of REMEs implemented. Solid lines (left scale) are clock frequencies; dashed lines (right scale) are number of LUTs.

Snort REMEs (18715 states) to VHDL, as m increased from 2 to 8. In all cases, about 1/3 of the time was used for file I/O.

These results show that the algorithms proposed in this paper are suitable for large-scale translation of regular expressions into VHDL. With $\sim$ 80% slice usage ($\sim$ 27k 4-LUTs) on the Virtex LX-40, a throughput of more than 10 Gbps was achieved for 267 REMEs in parallel. Furthermore, in practice, it takes only a few seconds for the algorithms to translate hundreds of REMEs into structural VHDL.

C. Construction of Synthetic REMEs

We first used the benchmark generator described in Section V-A to produce synthetic regular expressions of different numbers and complexities, then use our REME construction tools to convert the synthetic regular expressions into 2-character input 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 9 shows clock frequency and LUT usage versus length of REMEs. Series `concat1` was produced by one long string 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.

Series `union2` ran at lower clock frequency than series `concat1` due to the use of the `union` operator, which caused `union2` to have twice the state fan-in as `concat1`. The clock rates of both series started to decline linearly with respect to REME length around 32 to 40 states per REME. This decline was due to the longer routes to access BRAM for centralized character classification. 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 10, we analyzed the effect of the number of REMEs on achieved clock frequency and total LUT usage. In each test case, 64 states were generated for each REME; 30 states were wrapped inside a `closure` operator, which was then `union`-ed with a sequence of 30 other states and concatenated with the last 4 states in sequence. In the $w-$`union` series, $w = 1$, 2, or 3, the 30 states inside the `closure` operator were further wrapped by a `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 complexities.

As shown in Figure 10, clock frequency declined between

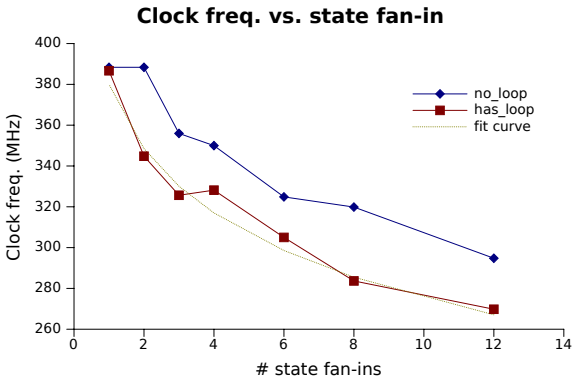

Fig. 11. Clock frequency versus state fan-in of the synthetic REMEs.

15% to 25% when number of REMEs varied from 1 to 16. 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, our staged pipeline came into effect, keeping the clock rates at slightly above 300 MHz. This evidently shows that the staged pipeline we propose was effective in scaling up number of REMEs in a single circuit. LUT usage maintained linear increase with respect to the number of REMEs.

Figure 11 shows clock frequency versus state fan-in. In each test case, REMEs of 52 states were constructed, with 24 states put inside a `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 V-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 others. The loop-back transition around the `union` operator (the `has_loop` series) connected the output of every operand to the input of every operand. This resulted in more complex routing and further impacted the clock frequency.

Overall we observed that the REME construction algorithms proposed here generated FPGA circuits with high clock frequency and high LUT efficiency for large number of regular expressions and high REME complexities.

VI. Conclusions

We presented the architecture and algorithms to construct high-performance *regular expression matching engines* (REMEs) on FPGA. We developed tools that convert a large number of regular expressions automatically into REMEs in VHDL, which could be accepted directly by FPGA synthesis and implementation tools. We also developed a benchmark generator to produce REMEs of different complexities, and used it to test the performance of our proposed architecture.

Our REME construction algorithms resulted in a circuit that utilizes both logic slices and block memory (BRAM) available on modern FPGA devices to achieve high REME density. It can be stacked in a space-extensive fashion to match multiple input characters per clock cycle. Our 2-dimensional *staged pipeline* effectively localized signal routing and achieved a clock rate over 300 MHz, processing hundreds of REMEs in parallel. Extensive studies showed that our tools and proposed algorithms are efficient and effective in realizing large-scale regular expression matching on FPGA.

References

[1] Becchi, M., and Crowley, P. An improved algorithm to accelerate regular expression evaluation. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems* (New York, NY, USA, 2007), ACM, pp. 145–154.

[2] Bispo, J., Sourdis, I., M.P.Cardoso, J., and Vassiliadis, S. Regular expression matching for reconfigurable packet inspection. In *FPT '06: Proceedings of the IEEE International Conference on Field Programmable Technology, 2006.* (Dec. 2006), pp. 119–126.

[3] Clark, C., and Schimmel, D. Scalable pattern matching for high speed networks. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004.* (April 2004), pp. 249–257.

[4] Floyd, R. W., and Ullman, J. D. The Compilation of Regular Expressions into Integrated Circuits. In *J. ACM* (New York, NY, USA, 1982), vol. 29, ACM, pp. 603–622.

[5] Hayes, C. L., and Luo, Y. Dpico: a high speed deep packet inspection engine using compact finite automata. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems* (New York, NY, USA, 2007), ACM, pp. 195–203.

[6] Hutchings, B. L., Franklin, R., and Carver, D. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2002), IEEE Computer Society, p. 111.

[7] Kumar, S., Chandrasekaran, B., Turner, J., and Varghese, G. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems* (New York, NY, USA, 2007), ACM, pp. 155–164.

[8] Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., and Turner, J. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM Computer Communication Review* (New York, NY, USA, 2006), vol. 36, ACM, pp. 339–350.

[9] Mitra, A., Najjar, W., and Bhuyan, L. Compiling PCRE to FPGA for accelerating SNORT IDS. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems* (New York, NY, USA, 2007), ACM, pp. 127–136.

[10] Sidhu, R., and Prasanna, V. Fast Regular Expression Matching Using FPGAs. In *FCCM '01: Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001.* (2001), pp. 227–238.

[11] 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 *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems* (New York, NY, USA, 2006), ACM, pp. 93–102.