High-Performance and Compact Architecture for Regular Expression Matching on FPGA*

Yi-Hua E. Yang and Viktor K. Prasanna
Ming-Hsieh Dept. of Electrical Engineering, University of Southern California
{yeyang and prasanna}@usc.edu

***Abstract*—We present the design, implementation and evaluation of a high-performance architecture for regular expression matching (REM) on field-programmable gate array (FPGA). Each regular expression (regex) is first parsed into a concise token list representation, then compiled to a modular nondeterministic finite automaton (RE-NFA) using a modified version of the *McNaughton-Yamada* algorithm. The RE-NFA can be mapped directly onto a compact register-transistor level (RTL) circuit. A number of optimizations are applied to improve the circuit performance: (1) *spatial stacking* is used to construct an REM circuit processing $m \geq 1$ input characters per clock cycle; (2) single-character constrained repetitions are matched efficiently by *parallel shift-register lookup tables*; (3) complex character classes are matched by a BRAM based classifier shared across regexes; (4) a *multi-pipeline* architecture is used to organize a large number of RE-NFAs into priority groups to limit the I/O size of the circuit. We implemented 2,630 unique PCRE regexes from Snort rules (February 2010) in the proposed REM architecture. Based on the place-and-route results from Xilinx ISE 11.1 targeting Virtex 5 LX-220 FPGAs, the proposed REM architecture achieved up to 11 Gbps *concurrent throughput* for various regex sets and up to 2.67x the *throughput efficiency* of other state-of-the-art designs.**

***Index Terms*—Regular expression, NFA, finite state machine, network intrusion detection, FPGA, BRAM, LUT, SRL**

I. Introduction

Regular expression matching (REM) is an important mechanism used by popular network intrusion detection systems (NIDS) such as Bro [1] and Snort [2] to perform deep packet inspection against potential threats. Due to the large number of patterns to scan for and the increasing bandwidth of network traffic, REM is becoming a performance bottleneck of the network security system; in addition, the software based REM engine itself may be vulnerable to performance-based DoS attack [3].

In language theory, a regular expression (regex) defines a regular language constructed over symbols from a fixed alphabet. Table I list the operators commonly used in a regex. Since regular languages are exactly the class of languages accepted by finite state machines, REM with the above operators can always be performed by either nondeterministic finite automata (NFA) or deterministic finite automata (DFA). In the NFA approach [4, 5, 6, 7, 8], individual regexes and their character matching states are processed in parallel with one another. As a result, more than one state in an NFA can be *active* at any time. Optimizations such as input/output pipelining [5], common-prefix extraction [4, 5], multi-character input [8], and centralized character decoding [4, 9], can be applied to improve throughput and reduce resource requirements of the overall design. The NFA can be further converted to a DFA with various optimization techniques [10, 11, 12, 13]. In principle, the DFA maintains only one active state and, for each input character, performs a single state transition table (STT) lookup to determine the next active state.

Table I: REM operators supported by finite automata.

Op.	Name	Example	Description
-	Concatenation	q_1q_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
^	Start of string	$\hat{}q$	q at start of input
$	End of string	$q\$$	q at end of input
[...]	Character class	`[a-c]`	Either `a`, `b` or `c`
[^...]	Inv. char. class	`[^\r\n]`	Neither `\r` nor `\n`

While many simple regexes can be efficiently matched by DFA-based REM, for some regexes converting the $O(n)$-state NFA to a DFA can generate an excessively large STT with $O(2^n)$ states, a phenomenon known as the (exponential) state explosion [14, 15]. In these cases, DFA-based REM can become infeasible to construct and accelerate. On the other hand, although NFA-based REM makes large numbers of state transitions per input character, the parallel processing nature of the NFA makes it particularly suitable to accelerate on FPGAs, which offer highly parallel logic and wire resources.

In this study, we focus on the design and analysis of NFA-based REM architecture on FPGA. Specifically, our main contributions are the following:

- *Modular RE-NFA construction*: We proposed the modified McNaughton-Yamada (MMY) algorithm for converting arbitrary regex patterns into modular RE-NFAs.
- *Spatially stacked multi-character matching*: We devised a time and space-efficient "spatial stacking" technique to transform any modular RE-NFA to a multi-character matching RTL circuit.
- *Parallel shift-register lookup table*: We designed a parallel shift-register lookup table (pSRL) architecture to match constrained character repetitions efficiently against multiple input characters per clock cycle.

* Supported by U.S. National Science Foundation under grant CCR-1018801.

- *BRAM-based character classification*: We utilized the block memory (BRAM) on FPGAs to match each input character with all complex character classes efficiently in one memory access.
- *Multi-pipeline priority grouping*: We designed a two-dimensional multi-pipeline architecture to match large numbers of RE-NFAs individually on a single FPGA with limited number of I/O pins.
- *Size estimation function*: We produced a simple size estimation function for the proposed RE-NFA, to quickly estimate and compare its relative circuit complexity on FPGA without the lengthy circuit synthesis.
- *Large-scale REM prototype*: We implemented prototype REM circuits for 2,630 unique regexes from Snort-rules on Xilinx Virtex 5 LX-220 FPGAs. The circuit construction and optimization process is fully automated and parametrized. In particular, we processed all regexes as-is without manual selection or simplification. [1]

The rest of this paper is organized as follows. Section II gives the background and prior work of NFA-based REM on FPGA. Section III and IV explain the proposed RE-NFA and REM circuit constructions, respectively. Section V discuss various circuit optimizations. Section VI evaluates prototype performance, and Section VII concludes the paper.

II. Background

A. Challenges of REM on FPGA

There are three challenges to performing REM on FPGA:

- Match large numbers of regex patterns in parallel.
- Handle arbitrarily structured regexes efficiently.
- Obtain high concurrent matching throughput.

Large number of patterns require more hardware resources. In addition, many regexes used by real-world applications (*e.g.* Snort [2]) contain nested unions and Kleene closures, long single-character repetitions, arbitrary character classes and multiple character classes in closures or sub-pattern unions. These regexes are often translated to NFAs with large number (thousands) of states and/or arranged in non-trivial topology. Although modern FPGAs offer high circuit parallelism, efficient use of on-chip resources is still critical to obtaining high regex capacity and achieving maximum clock rate.

To help understand the relation between REM throughput and regex capacity, we make the following definition.

Definition 1: The *concurrent throughput* of a REM circuit on FPGA for n regexes is defined as the throughput of the input stream(s) matched by all n regexes concurrently.

It can be seen that the concurrent throughput is determined by three factors: (1) the size/complexity of the regexes; (2) the amount of resource used by the REM circuit; and (3) the achieved clock frequency. In general, to obtain the same level of concurrent throughput, a *larger* set of regex patterns with *more complex* features require proportionally *more* resources (*i.e.*, number of slices).

[1]To our knowledge, this is the most comprehensive study on FPGA implementation of Snort-rules regexes to date. Our software is available for academic purposes upon request.

We note that the *concurrent* throughput is different from the *aggregated* throughput, which sums up the throughput of matching individual regexes. For example, a REM circuit matching 5 regexes against a 1 Gbps input may have 5 Gbps aggregated throughput but only 1 Gbps concurrent throughput. When evaluating the performance of a REM circuit architecture, the concurrent throughput should be used, and the resource usage must be normalized.

B. Prior Work

Hardware implementation of NFA-based REM was first studied by Floyd and Ullman [16]. They showed that, when translating directly to integrated circuits, an n-state NFA requires no more than $O(n)$ circuit area. Sidhu and Prasanna [7] later proposed a self-configuration algorithm to translate an arbitrary regular expression (regex) directly into its matching circuit on FPGA. Both studies are based on the same NFA architecture with the McNaughton-Yamada construction.

Large-scale REM was first considered in [5], where character inputs are pipelined and broadcast in a tree structure. It also proposed an automatic REM circuit construction using JHDL, in contrast to the self-configuration approach used in [7]. Automatic REM circuit construction in VHDL was proposed in [4] and [6]. In [4], each regex was first tokenized and parsed into a hierarchy of basic NFA blocks, then transformed into VHDL using a bottom-up scheme. In [6], a set of scripts were used to compile regexes into op-codes, convert op-codes into NFAs, and construct the NFA circuits in VHDL.

An algorithm for construction multi-stride NFA was proposed in [8] to transforms an n-state NFA accepting 2^i input characters per cycle into an equivalent NFA accepting 2^{i+1} input characters per cycle, $i \geq 0$, in $O(n^3)$ time. A technique using shift-register lookup tables (SRL) for implementing single-character repetitions was proposed in [4]. Other techniques such as common-prefix extraction and centralized character matching were used to further reduce resource usage. The resulting circuit, however, accepts only one input character per clock cycle. In [9] the authors explored the idea of pattern *infix* sharing to reduce the number of lookup tables (LUTs) required to match similar patterns.

III. Modular RE-NFA Construction

We construct a regex-matching nondeterministic finite automaton (RE-NFA) in two steps, described respectively in the two subsections below.

A. Parsing PCRE to Token List

As discussed earlier, theoretically each regular expression (regex) defines a regular language. Conventionally, the compilation of a regex starts from parsing the regex into a right-leaned (or left-leaned) parse tree representing the right-linear (or left-linear) production for the regular language [17, 16]. When used for real-world regexes in the Perl-Compatible Regular Expression (PCRE) format, however, this approach has several limitations:

1) It is not concise for constrained repetitions. A repetition of $\{n\}$ will be parsed as n identical concatenation nodes.

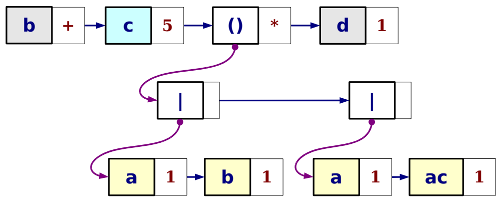


Figure 1: Token list for regex /`b+c{5}(ab|a[ac])+d`/.

A repetition of $\{m, n\}$ will be converted to an $(n-m)$-level nested union nodes in the parse tree.

2) It creates an unnecessarily deep tree, which in turn results in deep levels of recursion during the following McNaughton-Yamada construction. For example, a union of 10 sub-regexes will be parsed into 9 consecutive levels of union nodes.
3) It does not accept the PCRE extensions, such as assertions, lookarounds, conditionals and backreferences.

To overcome these limitations, we designed an efficient, simple and modular parsing approach to capture the full semantics of PCRE. Specifically, we transform each regex into a *token list* data structure. A *token* in the token list consists of the following major fields:

Field	Use and meaning of the field
val	Character class; union-op; parenthesis of sub-regex or various PCRE-extended features
rep	Constrained repetition; Kleene closure
next	Next token to concatenate or to union
child	Nested sub-regex or PCRE-extended feature

A token list is a multi-level linked list of tokens chained by the *next* and *child* fields. Normally, a concatenate operation is implied between a token and its *next* token; except when the token's *val* field is the "union-op" (|), then a union operation is used instead. Depending on its *val* field, a token can be one of four types: (1) single character class, (2) union-op to a sub-regex, (3) parenthesis of a sub-regex and (4) various PCRE-extended features. A type-1 or type-3 token may have a *rep* field representing the constrained (`{m,n}`) or closured (+ or $\star$) repetition; a type-2 or type-3 token always points to a nested token list via the *child* field.

The *val* field of a type-4 token stores a PCRE-extended feature, in which case the *child* field points to the corresponding raw sub-regex. Some of the features (*e.g.*, ^ and \$ in Table I) can be implemented by tweaking the following NFA construction (Section III-B); others cannot be handled by finite automata and will be either partially ignored (*e.g.*, conditional) or used to cut-short the regex (*e.g.*, backreference).

The token list for regex /`b+c{5}(ab|a[ac])+d`/ is shown in Figure 1. The *val* and *rep* fields are stored in the left and right blocks of each token, respectively. The *next* field is represented by the straight right-arrow, where as *child* by the curved down-arrow.

Conceptually, our token list data structure is a combination of the parse tree used in [7, 16] and the *opcodes* used in [6, 18]. The tokens are arranged in a 2-D list structure easy to traverse and manipulate; on the other hand, each token is powerful enough to efficiently represent a wide range of PCRE extensions. Although some of these extensions cannot be supported by finite automata, we believe the token list data structure can still be useful to future REM solutions based on a more powerful automaton.

B. Modified McNaughton-Yamada Algorithm

Instead of using the McNaughton-Yamada (MNY) construction [16, 19], which generates an NFA with many intermediate nodes and redundant ϵ-transitions, we use the *modified McNaughton-Yamada* (MMY) algorithm, outlined in Algorithm 1,[2] to convert a token list to a modular RE-NFA suitable for implementation on FPGA.

Algorithm 1 consists of two mutually recursive subroutines: (1) CONVSINGLE, which handles the conversion of a single token instance; (2) CONVTOK, which handles unions, closures and constrained repetitions which come inside a pair of parentheses. The algorithm starts from a call to CONVSINGLE on the first regex token (see Section III-A) and terminates when after the last token is converted. In particular, to handle the closure operations ('+' and '$\star$') without introducing extra states for the backward ϵ-transitions, a "pseudo" state is created temporarily (step 9) as the virtual source of the (backward) ϵ-transitions caused by the closure operator. Then, after the inner part of the closure has been constructed recursively (step 10), the "pseudo" state is mapped to the real source states and deleted (step 12).

Compared with the original MNY construction, the MMY construction has the following distinct properties:

1) Input ϵ-transitions to a token with the "`(...|...)`" operator are sent to each nested sub-token individually, while output ϵ-transitions from all sub-tokens are collected and sent to the subsequent token(s). (CONVTOK step 3)
2) For the "$\star$" and "+" operators, a *pseudo state* is created temporarily during token conversion to find all the feedback targets. The actual backward ϵ-transitions are made after the closured token has been converted. (CONVTOK steps 9 to 12)
3) For the "$\star$" operator, the input transitions into the token are sent directly to the subsequent token(s) without being aggregated by an extra state. (CONVTOK step 14)
4) Constrained repetitions are converted directly into proper NFA states without first converted into multiple unions. (CONVTOK steps 18 and {23 | 26})

As a result, none of the regex operators produces extra states to propagate ϵ-transitions. Note also that recursion only occurs in the MMY algorithm when there is a nesting of sub-regexes (due to unions or parentheses). In practice, the level recursion rarely exceeds 5. Figure 3 shows the RE-NFA constructed by the MMY algorithm for the token list in Figure 1.

Graphically, the MMY algorithm can be described by the 4 construction rules (a)–(d) in Figure 2. For each construction rule, the left-most (dark) circle corresponds to the S_{pre} in Algorithm 1, whereas the right-most (white) circuit corresponds to the set of states that receive ϵ-transitions from S_{out}. To

[2]For clear presentation error checking and corner cases are omitted.

Algorithm 1 The MMY algorithm.

Notations:

t_{cur}	Token currently being converted.
S_{pre}, S_{out}	Set of states transitioning into & out-of t_{cur}.
$t.val, t.rep$	Token t's *value* and *repetition* contents.
$t.next, t.child$	Token t's *next* and *child* tokens.
$s.foset$	Set of fanout states of a RE-NFA state s.
create_state (c)	Create a new state matching char. class c.
delete_state (p)	Delete state p and its fan-ins & fan-outs.

```
SUBROUTINE $S_{out}$ := CONVSINGLE $(t_{cur}, S_{pre})$
1: if $t_{cur}.val$ = "( )" then
2:     $S_{out} \leftarrow$ CONVTOK $(t_{cur}.child, S_{pre})$
3: else {$t_{cur}.val$ represents a character class}
4:     $s_{new} \leftarrow$ create_state $(t_{cur}.val)$
5:     $\forall s \in S_{pre}, s.foset \leftarrow s.foset \cup \{s_{new}\}$
6:     $S_{out} \leftarrow \{s_{new}\}$
7: end if
```

```
SUBROUTINE $S_{out}$ := CONVTOK $(t_{cur}, S_{pre})$
1: if $t_{cur}.val$ = "|" then
2:     while $t_{cur} \neq null$ do
3:         $S_{out} \leftarrow S_{out} \cup$ CONVTOK $(t_{cur}.child, S_{pre})$
4:         $t_{cur} \leftarrow t_{cur}.next$
5:     end while
6: end if
7: while $t_{cur} \neq null$ do
8:     if $t_{cur}.rep$ = "*" or $t_{cur}.rep$ = "+" then
9:         $p \leftarrow$ create_state $(null)$ {a pseudo-state}
10:        $S_{out} \leftarrow$ CONVSINGLE $(t_{cur}, S_{pre} \cup \{p\})$
11:        $\forall s \in S_{out}, s.foset \leftarrow s.foset \cup p.foset$
12:        delete_state $(p)$
13:        if $t_{cur}.rep$ = "*" then
14:            $S_{out} \leftarrow S_{out} \cup S_{pre}$
15:        end if
16:    else if $t_{cur}.rep$ = "$\{m,n\}$" then
17:        for $(cnt \leftarrow 0 ; cnt < m ; cnt{+}{+})$ do
18:            $S_{pre} \leftarrow$ CONVSINGLE $(t_{cur}, S_{pre})$
19:        end for
20:        $S_{out} \leftarrow S_{pre}$
21:        for $(cnt \leftarrow m ; cnt < n ; cnt{+}{+})$ do
22:            if use fan-in oriented approach then
23:                $S_{pre} \leftarrow$ CONVSINGLE $(t_{cur}, S_{pre})$
24:                $S_{out} \leftarrow S_{out} \cup S_{pre}$
25:            else {use fan-out oriented approach}
26:                $S_{out} \leftarrow$ CONVSINGLE $(t_{cur}, S_{out} \cup S_{pre})$
27:            end if
28:        end for
29:    end if
30:    $t_{cur} \leftarrow t_{cur}.next$
31: end while
```

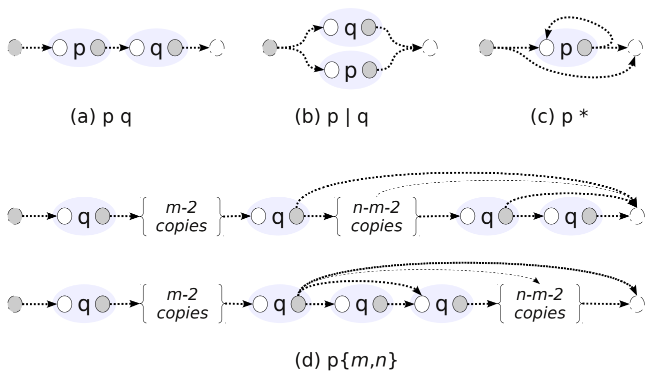


Figure 2: Graphical representation of the MMY construction rules. A letter (p or q) represents a token. Each shaded oval represents the RE-NFA for the specified token. Dashed lines represent ϵ-transitions. White and dark circles represent (sets of) states receiving and sending ϵ-transitions, respectively.

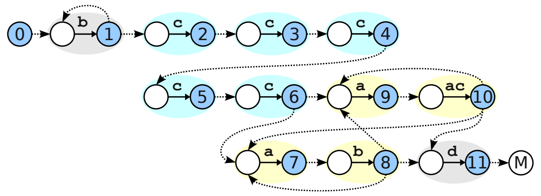


Figure 3: Example RE-NFA for regex `/b+c{5}(ab|a[ac])+d/`.

ensure applicability of the MMY rules to the edge tokens, extra `START` and `MATCH` states must be added to the RE-NFA as the initial "source" and final "sink" of ϵ-transitions, respectively. In the context of Algorithm 1, the `START` state constitutes the first S_{pre} and the `MATCH` state collects the last S_{out} during the recursive token list conversion. These two states are also used to handle the special ˆ and \$ operators, respectively.

Note that there are two approaches to the constrained repetition in Figure 2(d): a *fan-in oriented* (upper) approach and a *fan-out oriented* (lower) approach . The two approaches differ in how feedforward ϵ-transitions are connected in the last $n - m$ part of p$\{m, n\}$ (see also CONVTOK steps 23 and 26 in Algorithm 1, respectively). While functionally equivalent, the two approaches can result in different circuit-level tradeoffs on FPGA. The fan-in oriented approach induces states with at least $(n - m)$ input ϵ-transitions after the constrained repetition. The fan-out oriented approach increases the output load of the m-th repeating state by $(n - m)$ times. We note that, when the value of $n - m$ is high, it may be advantageous to break p$\{m, n\}$ into $k > 0$ segments: p$\{m_1, n_1\}$ p$\{m_2, n_2\} \cdots$ p$\{m_k, n_k\}$ where $m = \sum_{i=1}^{k} m_i$, $n = \sum_{i=1}^{k} n_i$ and $m_i, n_i > 0$, and apply both fan-in and fan-out oriented approaches alternately to various segments. In practice, we did not implement this feature since there are only few constrained repetitions with high $n - m$ value in Snort-rules regexes.

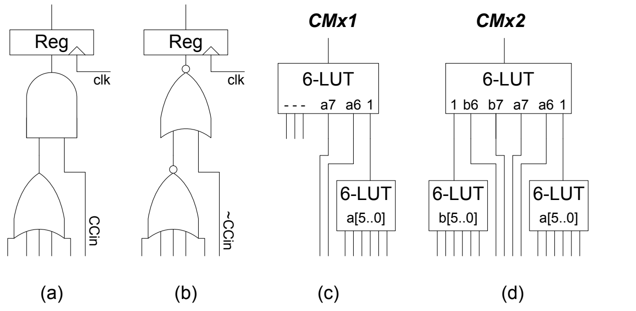


Figure 4: 6-LUT optimized circuit elements: state update modules with (a) normal and (b) inverted character matching inputs; compact logic for matching (c) one-value and (d) two-value simple character classes.

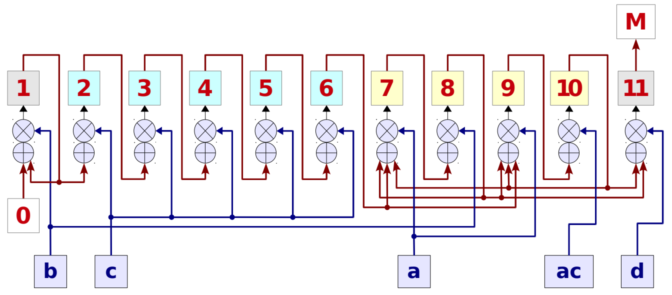


Figure 5: A basic RE-NFA circuit for matching regex /`b+c{5}(ab|a[ac])+d`/. The $\oplus$ and $\otimes$ symbols represent logic `OR` and `AND` gates, respectively.

IV. RE-NFA Circuit Construction

A. State Update Module

As shown in Figure 3, all pairs of nodes inside the lightly shaded ovals have an identical structure. When constructing the RE-NFA circuit on FPGA, each of these pairs constitutes a *state update module* matching a single character class. All we need to do is to connect the inputs and outputs between various state update modules as specified by the ϵ-transitions. Within each node pair, the right node corresponds to a 1-bit state register whereas the left node corresponds to an input ϵ-transition aggregator (an `OR` gate). The matching of the character class associated with the state is received as a 1-bit signal ($true$ or $false$) to the `AND` gate. The state update module does not otherwise care about what the character class is. As a result, state update and transition are inherently uncoupled from character matching and classification in our RE-NFA circuit architecture.

We design two types state update modules, one accepting normal character matching (Figure 4(a)) and the other accepting negated character matching (Figure 4(b)). This allows us to instantiate only one character matching circuit for both a character class and its negation, potentially cut the resource used for character matching by half. In addition, each state update module is also configured with one parameter: the number of input ports, determined by the number of "previous states" that immediately ϵ-transition to the current state.

Figure 5 shows an RE-NFA circuit matching the example

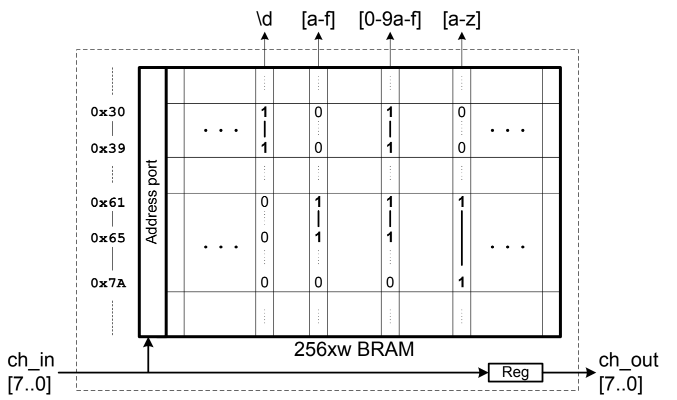


Figure 6: BRAM-based character classifier for w complex character classes.

regex /`b+c{5}(ab|a[ac])*d`/. On modern FPGAs with 6-input LUT, a k-input `OR` followed by a 2-input `AND` can be realized with a single LUT if $k \leq 5$, or two LUTs if $6 \leq k \leq 10$. Since most RE-NFA states ($> 95\%$) constructed from real-world regexes have less than 10 fan-in transitions, our RE-NFA circuit architecture spends only one or two two LUTs per state in the majority of cases.

B. Character Matching and Classification

A complex but important feature of PCRE is the use of character classes. A character class is effectively a union of one or more character symbols from the alphabet. Our RE-NFA architecture fully captures the character class semantics, where any state can match an arbitrary (pre-defined or customized) character class. Efficient implementation of character class matching can significantly reduce the complexity of the RE-NFA circuit. For example, the regex /`[ac][ab]{n-1}`/ can be matched by an n-state NFA with character classes `[ac]` and `[ab]`. In contrast, the equivalent regex /`[ac][ab]{n-1}`/ would have to be matched by $2n$ states with characters `a`, `b` and `c`; the number of inter-state ϵ-transitions is also doubled. In general, expanding a character class of size v to individual characters increases the number of NFA states and state transitions by v times.

We observe that, for 8-bit characters, any character classification can be fully specified by 256 bits, one for the inclusion of each 8-bit value. Thus a w-character classification can be implemented on a block memory (BRAM) of $256 \times w$ bits. Furthermore, if two RE-NFA states accept the same character class, they can share the same character classification output. Figure 6 illustrates an example character classifier where character classes `\d`, `[a-f]`, `[0-9a-f]` and `[a-z]` (among a few unspecified others) are matched in parallel by one BRAM access. In general, BRAM-based character classifier is only used to match *complex* character classes which would otherwise require much logic resource to implement.

While complex character classes are always aggregated and matched collectively by a (per-pipeline) classifier in BRAM, simple character classes consisting of one or two symbol values can be matched logic as shown in Figure 4(c) or

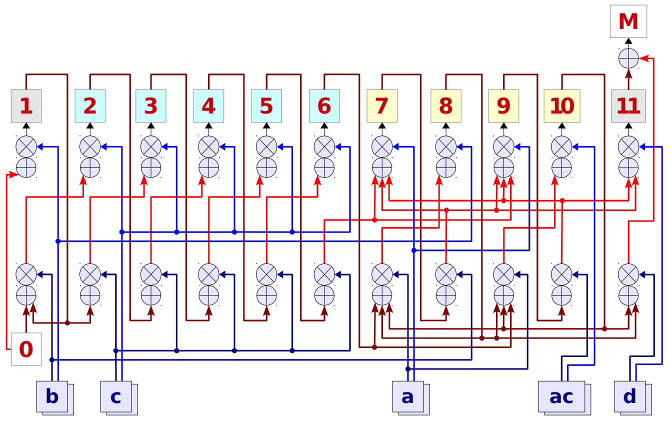


Figure 7: A 2-character matching circuit for regex /`b+c{5}(ab|a[ac])+d`/.

Figure 4(d), respectively. Matching characters in logic significantly reduces the utilization of on-chip BRAM, which can be used instead for buffering or other purposes in a larger system.

C. Differences from Previous Approach

While our RE-NFA circuit architecture is functionally similar to the prior work [4, 5, 6, 7, 8], there are three subtle but important differences:

1) State register occurs **after**, rather than **before**, the character matching. This allows the character matching latency to overlap with the state transition latency, resulting in higher achievable clock rate.
2) The modular RE-NFA can be converted automatically into a modular circuit on FPGA. As is discussed later, individual circuits can be *spatially stacked* to match multiple input characters per clock cycle.
3) We do *not* minimize logic at the HDL level. In our RE-NFA circuit, the same signal can be produced by different state update modules (see Figure 5 states 7 and 9). We leave such minimization to the synthesis software, which will apply its own optimization on the circuit to meet the implementation constraints.

V. REM Circuit Optimizations

A. Spatial Stacking for Multi-Character Matching

Previous designs of multi-character matching [8] adopted *temporal* transformations at the NFA-level, where state transitions are extended forward in time (clock cycle) to construct a new NFA with multi-character state transitions. In contrast, we adopt a circuit-level *spatial stacking* technique to construct multi-character matching circuits. A formal procedure is described in Algorithm 2. An example 2-character matching circuit for the regex /`b+c{5}(ab|a[ac])*d`/ is shown in Figure 7.

For an n-state RE-NFA with max state fan-out d, an application of Algorithm 2 requires $O(n \times d)$ time and produces a circuit taking $O\left(n \times m \times d^2\right)$ area. A single application of the algorithm on two copies of the same m-character matching circuit generates a $2m$-character matching circuit.

Algorithm 2 Multi-character matching circuit construction.

Input: An n-state m_1-character matching circuit C_{m_1} and an n-state m_2-character matching circuit C_{m_2} for some regex.

Output: An n-state m-character matching circuit C_m, $m = m_1 + m_2$, for the same regex.

1) For each $i \in \{1, \ldots, n-1\}$, suppose the output of state i connects to state inputs $\{i_1, i_2, \ldots, i_t\}$:
   a) Remove the state register for state i of C_{m_1}; forward the `AND` gate output of the basic state block directly to the state output.
   b) Disconnect state output i of C_{m_1} from all state inputs of C_{m_1}, and re-connect them to state inputs $\{i_1, i_2, \ldots, i_t\}$ of C_{m_2}.
   c) Disconnect state output i of C_{m_2} from all state inputs of C_{m_2}, and re-connect them to state inputs $\{i_1, i_2, \ldots, i_t\}$ of C_{m_1}.
   d) State i of the combined circuit receives $(m_1 + m_2)$ consecutive character matching signals. The first m_1 signals go to the original C_{m_1} part, while the last m_2 signals go to the original C_{m_2} part.
2) Merge the `START` states of C_{m_1} and C_{m_2} into a single `START` state; connect the output of the merged `START` state to all targets of the original `START` states.
3) Merge the `MATCH` states of C_{m_1} and C_{m_2} into a single `MATCH` state by `OR`-ing the inputs of the original `MATCH` states as the input to the merged `MATCH` state.
4) The resulting circuit is an n-state m-character matching circuit, $m = m_1 + m_2$, for the same regex.

Thus recursively, an n-state m-character matching circuit can be constructed in $O(n \times d \times \log_2 m)$ time, starting from a one-character matching circuit for the n-state RE-NFA.

Lemma 1: The circuit constructed by Algorithm 2 over two copies of a single-character matching circuit is a valid two-character matching circuit.

Proof: First note that a state machine at any moment is completely described by its current state values and the next state transitions. Suppose we perform Algorithm 2 on two identical one-character input circuits, C_A and C_B, for the same regular expression, but *without* removing the state registers of C_A (*i.e.*, do not perform step 1a of Algorithm 2):

1) The algorithm does not add, remove, or change order of the states in either C_A or C_B, nor does it modified the logic used to produce any state value, thus the state values are preserved.
2) From C_A's outputs to C_B's inputs, state transitions are preserved by step 1b of Algorithm 2.
3) From C_B's outputs to C_A's inputs, state transitions are also preserved by step 1c of Algorithm 2.
4) The state transition labels of both C_A and C_B are kept intact by step 1d of Algorithm 2.

The combined circuit, with the same state values and state transitions, would function exactly the same as an original one-character input state machine at every clock cycle. Removing the state registers of C_A simply merges the two-cycle pipeline

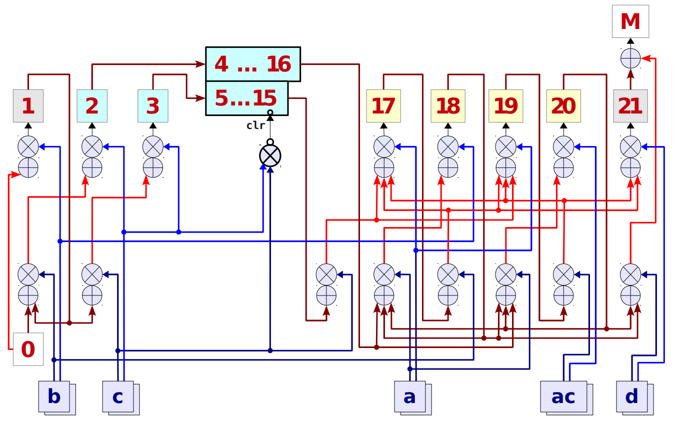


Figure 8: A 2-character matching circuit for regex /b+c{15}(ab|a[ac])+d/ with a 2-way parallel SRL for the single-character repetition.

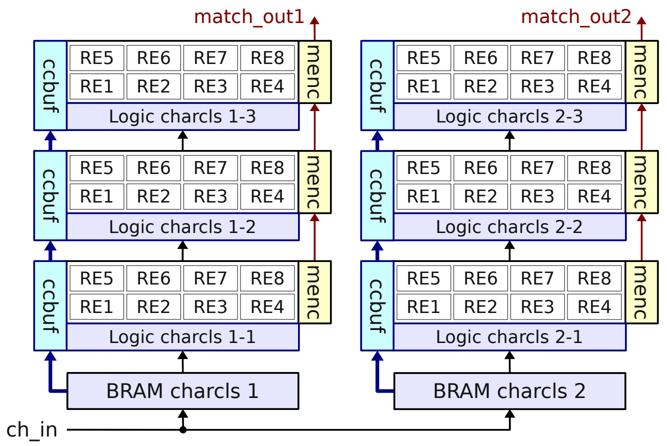


Figure 9: An example REM multi-pipeline with 48 RE-NFA circuits in 6 priority groups and 2 aggregated BRAM-based character classifiers.

of the combined circuit into one cycle, resulting in a circuit of 2-character input per clock cycle. ■

Theorem 1: Algorithm 2 can be used to construct a valid m-character matching circuit for any $m > 1$.

Proof: By the same arguments as for Lemma 1, a circuit constructed by Algorithm 2 over an m-character matching circuit and a single-character matching circuit is a valid $(m+1)$-character matching circuit. By induction, any circuit constructed by Algorithm 2 with an integer $m > 1$ is a valid m-character matching circuit. ■

Compared to the previous approaches, our multi-character construction is simpler, more flexible, and more efficient with the larger number of inputs in the lookup tables (LUTs) in modern FPGAs. The resulting circuit can be optimized automatically and effectively by the FPGA synthesis tools at the circuit level. While a higher value of *m* generally lowers the achievable clock rate, overall throughput still improves dramatically by the ability to match multiple characters per clock cycle (see Table IV).

B. Parallel SRL for Single-character Repetition

A common PCRE feature used by many real-world regexes is single-character repetition, where a character class is repeated for a fixed number of times. In a straight-forward implementation, each repeating instance requires a state register and its associated state update logic, which can occupy much resource on FPGA for a repetition of over a few hundred times. Alternatively, the repetition can be implemented as shift-register lookup tables (SRLs), which are much more resource-efficient than individual registers on FPGA [4]. However, the SRL approach proposed in [4] produces only single-character matching circuits. For example, while states 3 to 6 in Figure 5 can be easily implemented as a shift register of length 4, the same states in Figure 7 involve interleaving state transitions and cannot be implemented by the simple shift register architecture.

We design a *parallel SRL* (pSRL) architecture for single-character repetitions with multi-character matching. A pSRL of *width* $m > 1$ consists of m *tracks*. Each track is implemented as a shift register, has its own shift_in port, state storage and data_out port. All tracks share the same clear signal, but may not have the same length. In every clock cycle, each track accepts one bit from its shift_in port and shifts its internal states by one bit-position towards the data_out port. To construct an m-character matching circuit for a single-character repetition of $r \geq 2m$ times, we connect the output of the first m repeating states to the shift_in ports of an m-track pSRL. The i-th pSRL track, $0 \leq i \leq m-1$, has a state storage of $\left\lceil \frac{r-m-i}{m} \right\rceil$ bits, with its data_out equivalent to the original k-th repeating state, $k = i + m \times \left\lceil \frac{r-m-i}{m} \right\rceil$. The entire pSRL is cleared when any of the m input characters does not match the repeated character class.

Figure 8 shows a 2-character matching circuit with pSRL. To better demonstrate the resource saving, we increase the repetition amount (r) from 5 to 15 in the example. Note that the pSRL architecture is only effective in reducing resource usage on FPGA when $r > 2m$ for an m-character matching circuit. In practice, we have hundreds of instances of large ($>$ 100) single-character repetitions in Snort-rules regexes, some of them repeating over 1,000 times.

C. Multi-Pipeline Priority Grouping

In many real-world regex matching scenarios such as deep packet inspection, a character input stream is usually matched by a small set of regexes. Thus a large-scale regex matching solution consisting of thousands of matched regex patterns can be naturally grouped into several tens to a few hundred regex groups, each matching only a few regexes.

In our large-scale REM design we use a 2-dimensional *staging* and *pipelining* structure as illustrated in Figure 9. Our design supports up to 8 pipelines with up to 16 stages per pipeline and 32 regexes per stage. In total up to 4,096 RE-NFAs can be matched in parallel. Each pipeline matches the input characters with its own BRAM-based complex character classifier. Within a pipeline, all the RE-NFAs are grouped in stages with lower-numbered stage having higher priority. Input characters as well as the complex character classifications are buffered and forwarded through the stages. A match output from a low-number (high-priority) stage overrides those from all higher-number stages at the same clock cycle. Within each stage, match outputs of all RE-NFAs can either be joined (logic-`AND`'d) or prioritized, depending on how the group of regexes are matched.

All RE-NFAs in the a pipeline share the same complex character classifier, which greatly reduces the amount of BRAM required for character classification. When constructing the BRAM-based character classifier, a function is called to compare each state's character class to the character class entries (columns of 256 bits) collected in BRAM so far. If the current character class is the same as or a simple negate of an existing character class, then the output of the existing entry is simply wired to the current state. This procedure has time complexity $O(n \times w)$ with n total number of states and w *distinct* character classes in *all* RE-NFAs. The space complexity is just $256 \times w$. In the worst case, w could be linear in n; in practice, w tends to grow much more slowly than $O(\log n)$. Prudent grouping of RE-NFAs into pipelines can further reduce the growth of w with respect to n.

On the other hand, while using more pipelines increases the number of independent matches that can be output in a clock cycle, doing so also reduces the potential resource sharing in the (complex) character classifier. Subject to the I/O constraint, a "flat" multi-pipeline favors more concurrent match outputs, whereas a "tall" multi-pipeline favors more common characters. The height of the multi-pipeline can be configurable at compile time to tradeoff resource efficiency for multi-match capability.

D. Regex Complexity Metrics and Estimates

The one point that we will emphasize here is that *all regular expressions are **not** created equal*. Thus, when measuring the performance of a regular expression matching (REM) architecture, it is critical to take into account the intrinsic complexity of the regular expressions (regex) being matched. We say a regex is more complex when it is converted to a more complex RE-NFA quantified by the following metrics:

State count Total number of states in the RE-NFAs.

State size The number of states that immediately transition to any state in the RE-NFAs.

Fan size The number of states that any state in the RE-NFAs immediately transitions to.

The *state count* was used in [16] to describe the area requirement of the NFA-based REM circuit; in our RE-NFA architecture, a higher state count translates to more state update modules. The *state size* and *fan size* are affected by the use of union and closure operators. A higher *state size* translates to a larger `OR` gate to aggregate the input transitions, directly affecting the size of the state update module. A higher *fan size* increases the signal routing distance and complexity.

Theoretically the *maximum* state size and fan size would determine the largest state update module and the slowest signal route, respectively; in practice circuit-level optimizations performed by the synthesis tools are often able to compensate the effects of a few states with large numbers of fan-ins or fan-outs. On the other hand, high *average* state size and fan size can make the REM circuit more complex on the whole and thus effectively impact its overall performance.

Due to the modular structure of our RE-NFA, we can fairly accurately estimate these three metrics for every RE-NFA state with a specified multi-character matching value (*m*). We design two types of the *size estimation function* for the i-th state of a RE-NFA, $S(i)$, as follow:

- For ordinary state update module,

$$S(i) = \left\lceil n(i) \times \frac{m}{L} \right\rceil \tag{1}$$

- For single-character repetition in pSRL,

$$S(i) = (m+1) \times \left\lceil \frac{m}{L} \right\rceil + \left\lceil \frac{r(i)}{R} \right\rceil \tag{2}$$

where $n(i)$ is the number of fan-in ϵ-transitions to the i-th state, m is the multi-character matching number, L and R are device-dependent constants specifying the size of each LUT and SRL, respectively, and $r(i)$ is the repeating count of the single-character repetition covering the i-th state. Note that only one size estimate is produced for multiple states in the same single-character repetition.

Note that Equations 1 and 2 are derived directly from the circuit architectures in Section V-A and V-B, respectively; they also do not account for the character matching logic. In practice, for large REM circuits the total number of register-LUT pairs used tend to increase faster than $n(i) \times \lceil \frac{m}{L} \rceil$ due to resources used for signal routing.

Note also that both equations offer only approximate resource estimations; the actual resource usage of each regex matching circuit is highly dependent on the optimizations performed by the synthesis and place-and-route tools. Nevertheless, the size estimate functions can still be useful when we need to quickly compare the relative complexities of a REM solution on FPGA matching different sets of regexes. The number of LUTs required for character matching, on the other hand, can be calculated from the circuit architecture in Figure 4(c) and Figure 4(d).

VI. Performance Evaluation

A. Regex Statistics

We used regexes from Snort-rules (published February 2010) for our RE-NFA evaluation. For fair evaluation, we removed regexes that are too short (< 10 states) and counted identical regexes in different rules as a single one (*i.e.*, all regexes are syntactically unique). We also omit a handful of regexes with $> 2,500$ states, since these extremely long regexes are more appropriate to be handled by some other specialized methodology, which is out of scope of this paper. Many distinct regexes have common prefixes; to reduce the favorable performance bias from these similar regexes, we specifically avoided common-prefix extraction and implemented each regex matching separately. We did not otherwise make manual selection or simplification to the regexes.

The entire set of 2,630 regexes is too large to fit onto a single FPGA. Instead, we partition them into 6 subsets as follows:

Table II: Statistics of the implemented Snort-rules regexes.

name	# pat.	# states	% size-2	% size-3	% size-4+
wax1	399	34,043	14.0	1.46	3.50
wax2	399	35,485	13.8	1.45	3.37
wax3	382	32,944	14.1	1.38	3.71
osp	565	44,341	5.76	0.15	0.10
wsv	332	37,032	7.33	0.86	0.21
bem	553	67,964	3.57	0.21	0.14
Total	**2,630**	**251,809**	8.74	0.79	1.52

wax1	Part 1 of the *web-activex* rule set.
wax2	Part 2 of the *web-activex* rule set.
wax3	Part 3 of the *web-activex* rule set.
osp	From *oracle*, *spyware-put* and a few small sets.
wsv	From *web-**, *smtp* and *voip* rule sets, *etc.*
bem	From *backdoor*, *exploit* and *misc* rule sets, *etc.*

The 6 regex sets can be categorized into two groups. The "*wax*" sets contain more Kleene closure and union operators, but no constrained repetition. The other three sets (*osp*, *wsv* and *bem*) have much fewer closures and unions but a fair amount of constrained repetitions. Within each group, various regex sets have very similar statistics, as shown in Table II. A "size-*k*" state ($k = 2$, 3, ...) accepts k incoming transitions from the same or other states. In general, a size-k state is more complex than a size-h state if $k > h$.

Some Snort rules use "non-regular" regex features such as *backreference* – the ability to refer to a previously matched string as the string pattern to match against the following input. In our implementation, for any regex with backreference, the part of the regex after the first backreference is ignored. Out of over 2,600 *distinct* regexes we took from Snort rules, less than 100 ($< 4\%$) use backreferences, mainly in the *netbios* and *web-client* rule sets.

B. Implementation Results

As evidenced from Section VI-A above, all regexes are *not* created equal. Thus we defined the following metric to fairly evaluate the performance of the proposed REM solution across various different regex sets:

Definition 2: The *throughput efficiency* of a REM circuit on FPGA, in units of Gbps**state*/LUT, is defined as the concurrent throughput of the circuit divided by the average number of LUTs used per state.[3]

Multiple circuits were constructed for each of the 6 regex sets in Table II, matching $m = 2$, 4, 6 and 8 characters per cycle. Each circuit was organized as two pipelines having 5 to 9 stages per pipeline. The targeted device was Virtex 5 LX-220, synthesized (*xst*) and place-and-routed (*par*) using Xilinx ISE 11.1 with either the "maximize speed" or "minimize area" option. Only place-and-route results are reported.

Figure 10 shows the throughput performance and resource usage, respectively, of the circuits matching the *wax1*, *wax2* and *wax3* regex sets. {Fbase, Tbase} plot the {F̲requency, T̲hroughput} of the "base" circuits where regexes are arbitrarily grouped into stages; {Fsort, Tsort} plot those of the "sorted" circuits where regexes are organized into stages of roughly equal size (in number of states). Similarly, {Nbase, Ebase} plot the {FF-LUT pairs, throughput efficiency} of the base circuits, while {Nsort, Esort} the sorted circuits. Because there is no constrained repetitions in these three regex sets, pSRL was not used. An "_s" suffix indicates the circuit is optimized for speed when synthesized; an "_a" suffix indicates optimized for area. Table III gives an overview of the notations used in Figures 10 and 11.

[3]Throughput alone does not account for the length and complexity of the regexes. LUT efficiency alone, on the other hand, does not reflect clock frequency or multi-character matching.

Table III: Legend notations used in Figure 10 and Figure 11.

$\{A\}\{B\}\_\{C\}$

$\{A\}$	
F	Frequency (clock rate)
T	Concurrent Throughput
N	Number of FF-LUT pairs
E	Throughput Efficiency

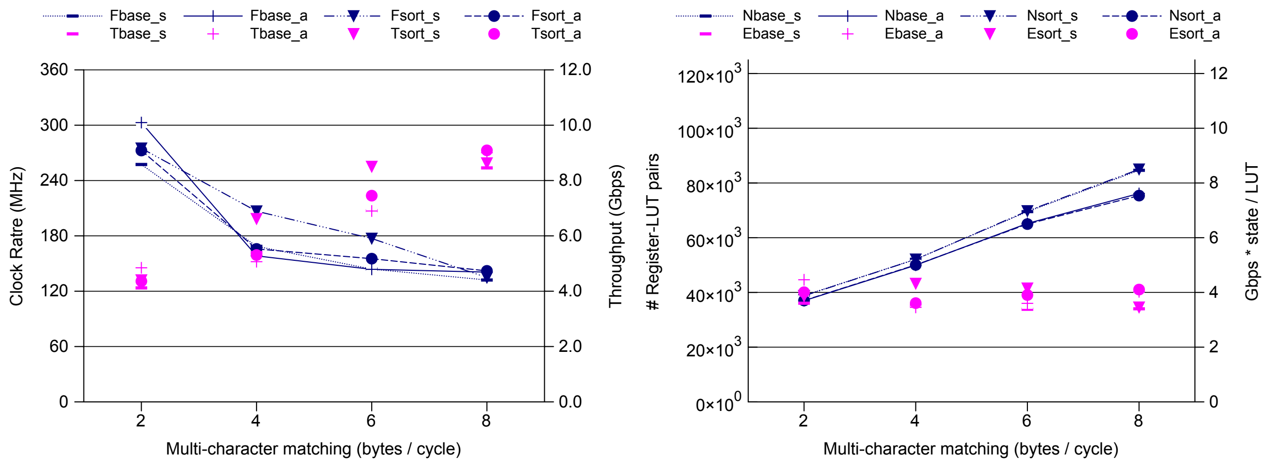
$\{B\}$	
base	Stages contain arbitrarily grouped regexes.
sort	Stages contain roughly equal no. regex states.
psrl	Stages contain pSRL-optimized regex circuits.

$\{C\}$	
_a	Circuit optimized for area.
_s	Circuit optimized for speed.

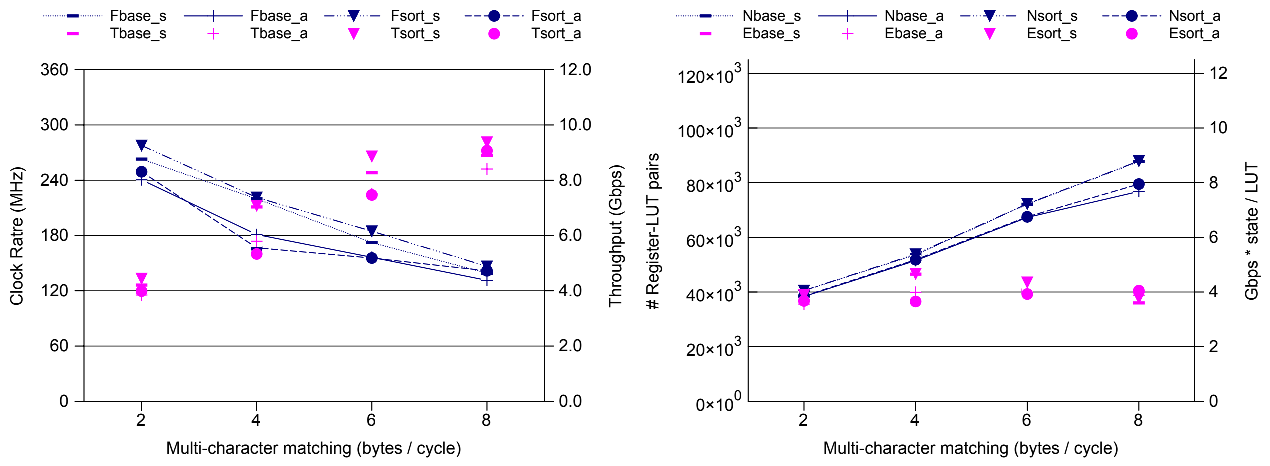
Higher multi-character matching reduced clock rate but increased matching throughput; it also required proportionally more resource on FPGA. As a result, throughput efficiency did not change significantly between $m = 2$ to $m = 8$. Sorting the regexes to balance the number of states across stages had a small but noticeable impact on throughput, since size-balanced stages would theoretically reduce the length of the longest signal path. More interestingly, while optimizing for area (*_a) resulted in lower LUT usage, for $m = 8$ optimizing for area actually resulted in higher throughput than optimizing for speed (*_s) for the *wax1* and *wax3* regex sets, as shown in Figure 10. One explanation is that with $m = 8$, the state update modules became very big, making signal routing a much more dominant factor in determining the clock period. Thus reducing the circuit size, which helps to shorten the signal route, also contributes more significantly to increasing the circuit speed.

Figure 11 shows the throughput performance and resource usage, respectively, of the circuits matching the *osp*, *wsv* and *bem* regex sets. Similar conventions were used for the plots in these figures, except the "Xpsrl" labels which indicate the circuits were optimized with pSRL for matching constrained repetitions (Section V-B). On the other hand, the regexes were *not* sorted in any particular way.

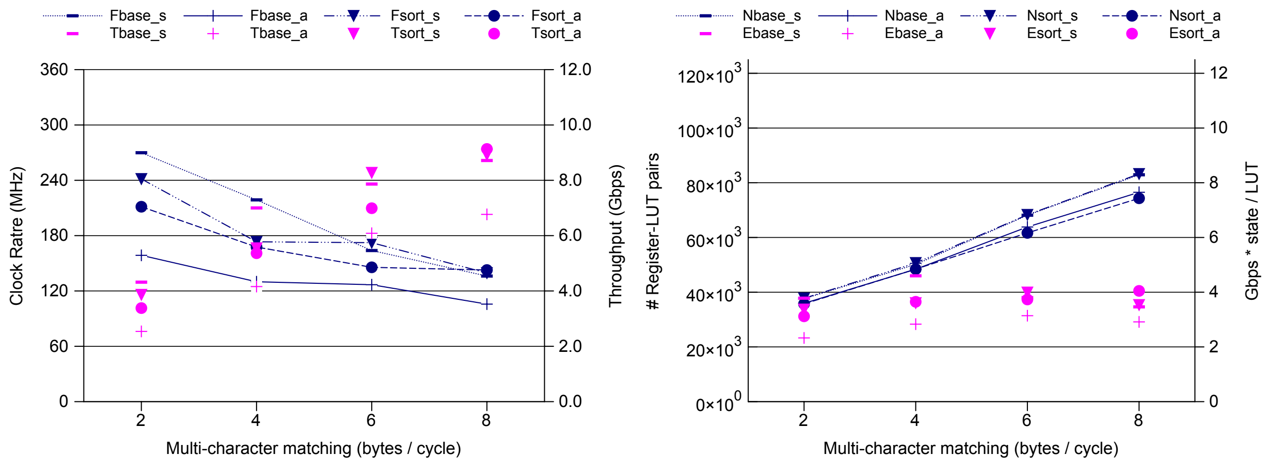
Like before, we observed similar throughput increase with higher multi-character matching, as well as higher throughput optimizing for area than optimizing for speed when $m = 8$. In addition, we find throughput efficiency to slightly increase with higher multi-character values and peak around $m = 6$. This is expected since the targeted FPGA had 6-input LUTs. As shown in Figure 11, applying pSRL to the circuits significantly reduced their resource usage by up to 40%. Interestingly, only when the circuits were optimized for area did pSRL make a big difference. The smaller circuits with pSRL also had visible speed advantage than the base circuits without pSRL.



(a) The “wax1” regex set.

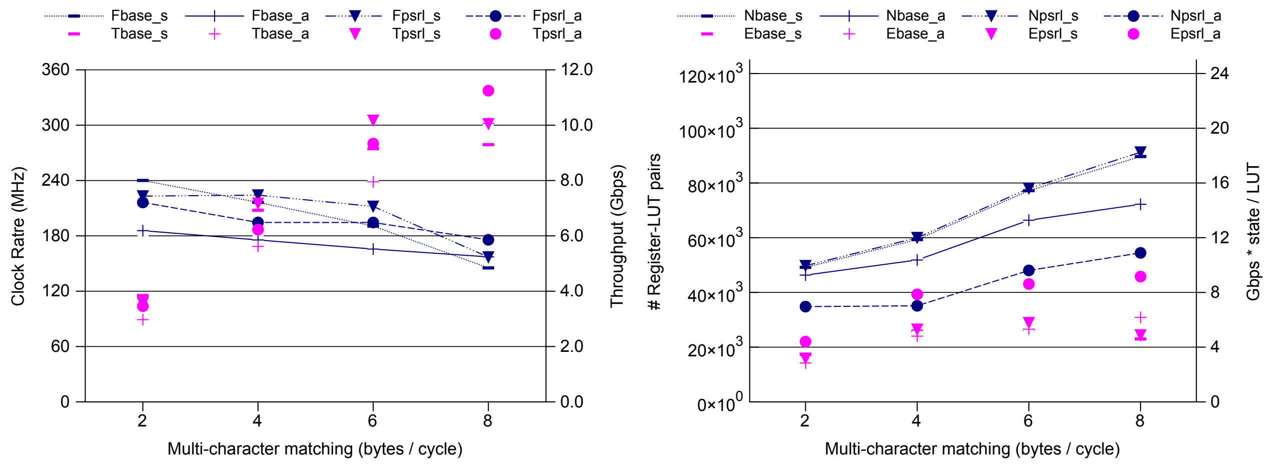


(b) The “wax2” regex set.

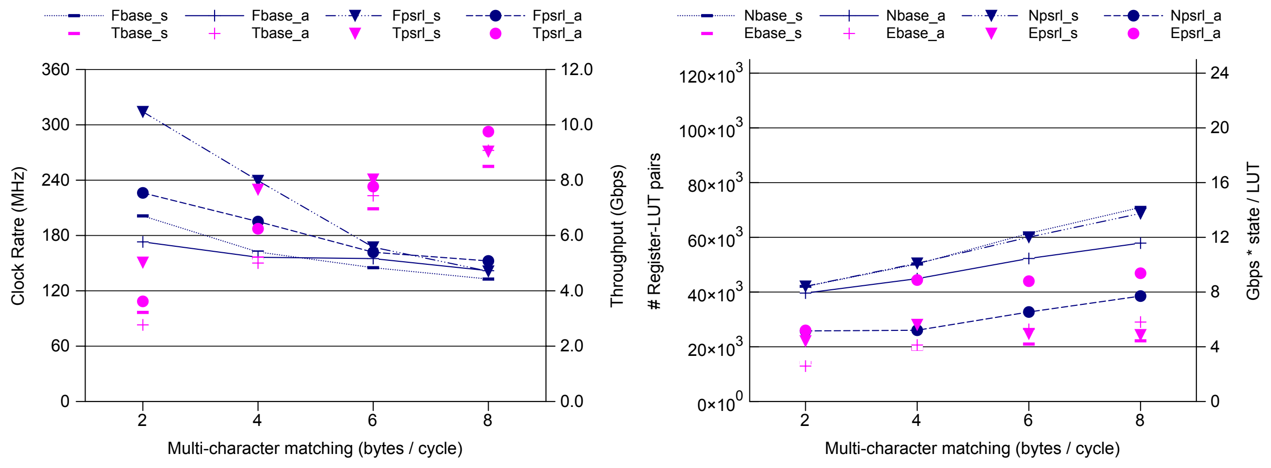


(c) The “wax3” regex set.

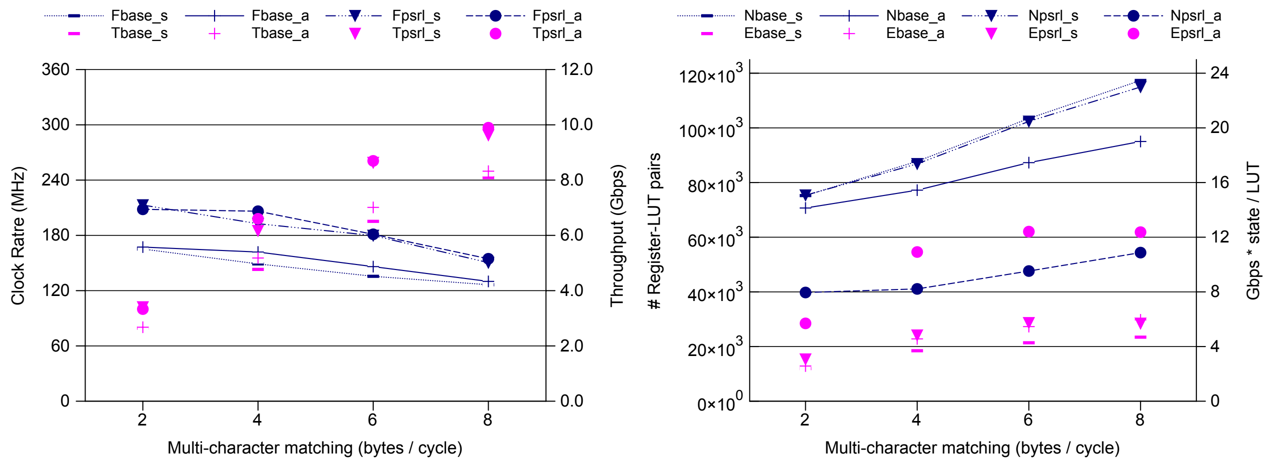
Figure 10: Performance of our REM circuits for regexes with no constrained repetition (see Table III for the meaning of the label notations).



(a) The “osp” regex set.



(b) The “wsv” regex set.



(c) The “bem” regex set.

Figure 11: Performance of our REM circuits for regexes with some constrained repetitions (see Table III for the meaning of the label notations).

Table IV: Performance comparison of FPGA based REM implementations.

	# non-meta chars (states)	# chars per regex	Multi-char. (chars/cycle)	Clock rate (MHz)	**Throughput (Gbps)**	# (A)LUT per state	**Tput efficiency** ($\frac{\text{Gbps}*\text{states}}{\#\text{LUT}}$)
Our design w/ pSRL (Figure 11 average)	> 120k states	> 82 chars	8	160.9	**10.3**	1.02	**10.1**
			6	179.2	**8.60**	0.89	**9.66**
			4	198.6	6.36	0.70	9.09
			2	216.9	3.47	0.69	5.03
Our design w/o pSRL (Figure 10 average)	> 100k states	> 84 chars	8	142.1	**9.10**	2.24	**4.06**
			6	152.2	**7.30**	1.89	**3.86**
			4	166.7	5.33	1.47	3.63
			2	244.4	3.91	1.09	3.59
Yamagaki *et al.* [8]	40,896	∼ 15	4	113.4	3.63	0.94*	3.86
Sourdis *et al.* [20]	69,127	∼ 46	1	302.5	2.42	0.66	3.67
Bispo *et al.* [4]	19,580	∼ 38	1	362.5	2.9	1.28	2.3
Mitra *et al.* [6]	N.A.	–	1	100	0.8	∼ 2.3	∼ 0.35

* The REM circuits in [8] are implemented on a different family of FPGA (Altera Stratix II EP2S180).

C. Performance Comparison

Table IV shows the comparison of our results with other state-of-the-art NFA-based REM on FPGA. Most prior work report the total number of *non-meta characters* in the regexes. However, this value was of no significance to us due to our use of the BRAM-based complex character classifiers (Section IV-B). Instead, what most significantly affected our resource usage was the number of states update modules (Section IV-A), which were up to 20% lower than the number of non-meta characters for complex regexes (for example, `/b+([ab]|[ac])*/` has 5 non-meta characters but only 3 states). On the other hand, in some cases a single non-meta character (*e.g.*, "`a{100}`") in previous studies (*e.g.*, [4]) counts as multiple states in our approach. In Table IV, we assumed every state in our architecture equals to only one non-meta character reported by others.

Our design achieved higher throughput *and* throughput efficiency than previous state-of-the-art results. In [8], REM circuits matching 8 characters per cycle were also designed, but only at much smaller scale (512 regexes), probably due to the complex multi-character circuit construction based on temporal extension. In [20], aggressive SRL constrained repetition and common-prefix sharing were used to achieve very low resource usage per state, although the resource saving was perhaps due more to the *pattern* properties of the regexes. Although [6] achieved lower performance than other solutions, it is worth mentioning that it utilized a generic software-to-FPGA compilation and had performance numbers experimentally measured (rather than reported by the synthesis and place-and-route tools). It was also the only FPGA-based REM able to handle true backreferences.

In our REM architecture, the ability to perform scalable multi-character matching and the application of pSRL for constrained repetition made the greatest performance differences. Most prior work implemented a small set of regexes and did not describe how their regexes were selected. In contrast, we implemented regex sets covering comprehensive and up-to-date (as of February 2010) Snort rules. Other than removing the non-regular features (such as backreferences), we made no manual selection of the regexes. As a result, our regex sets are significantly larger and more complex than those used by the prior work. To implement the wide variety of regexes, our circuits are optimized by the synthesis tools directly and automatically from the RTL architecture, and do not achieve the highest clock rates among other state-of-the-art REM implementations on FPGA. However, the architectural improvements in our design more than offset the effect of the slower clock rates, resulting in overall higher matching throughput and throughput efficiency.

VII. Conclusion and Future work

We presented a compact architecture for high-performance REM on FPGA. We described automatic regex-to-NFA compilation for large-scale REM, as well as optimizations such as multi-character matching, parallel SRL, shared BRAM-based character classifier and multi-pipeline priority grouping. Our designs achieved significantly higher matching throughput and throughput efficiency than previous state-of-the-art over comprehensive sets of regexes from Snort rules.

We plan to extend our REM circuits with the ability to handle multiple packet streams, which will help to better integrate the REM with the rest of the rule-based NIDS. We also plan to enhance our REM architecture to handle PCRE extensions such as lookarounds and conditionals. Based on the modular RE-NFA, we plan to research efficient architectures for push-down automata to match more complicated patterns.

References

[1] "Bro Intrusion Detection System," http://bro-ids.org. [Online]. Available: http://bro-ids.org/

[2] "SNORT," http://www.snort.org/.

[3] R. Smith, C. Estan, and S. Jha, "Backtracking Algorithmic Complexity Attacks against a NIDS," in *Proc. 22nd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2006, pp. 89–98.

[4] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *Proc. IEEE Intl. Conf. on Field Programmable Technology (FPT)*, December 2006, pp. 119–126.

[5] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable

Hardware," in *Proc. IEEE Sym. on Field-Programmable Custom Computing Machines (FCCM)*, 2002, p. 111.

[6] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in *Proc. ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS)*, New York, NY, USA, 2007, pp. 127–136.

[7] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proc. IEEE Sym. on Field-Programmable Custom Computing Machines (FCCM)*, 2001, pp. 227–238.

[8] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-Speed Regular Expression Matching Engine Using Multi-Character NFA," in *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, Aug. 2008, pp. 697–701.

[9] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of Regular Expression Pattern Matching Circuits on FPGA," in *Proc. Conf. on Design, Automation and Test in Europe (DATE)*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 12–17.

[10] M. Becchi and P. Crowley, "An Improved Algorithm to Accelerate Regular Expression Evaluation," in *Proc. ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS)*, 2007, pp. 145–154.

[11] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia," in *Proc. ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS)*, 2007, pp. 155–164.

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," *SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 339–350, 2006.

[13] R. Smith, C. Estan, and S. J. S. Kong, "Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata," in *ACM SIGCOMM*, August 2008.

[14] A. R. Meyer and M. J. Fischer, "Economy of description by automata, grammars, and formal systems," in *Proc. 12th Sym. on Switching and Automata Theory (SWAT '71)*. Washington, DC, USA: IEEE Computer Society, 1971, pp. 188–191.

[15] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection," in *Proc. ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS)*, 2006, pp. 93–102.

[16] R. W. Floyd and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," *Journal of ACM*, vol. 29, no. 3, pp. 603–622, 1982.

[17] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., 1972.

[18] "PCRE: Perl Compatible Regular Expression." http://www.pcre.org/. [Online]. Available: http://www.pcre.org

[19] R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Automata," *IEEE Trans. on Comput.*, vol. 9, no. 1, pp. 39–47, March 1960.

[20] I. Sourdis, S. Vassiliadis, J. Bispo, and J. M.P.Cardoso, "Regular Expression Matching in Reconfigurable Hardware," *Journal of Signal Processing Systems*, vol. 51, pp. 99–121, 2008.