Compact Architecture for High-Throughput Regular Expression Matching on FPGA[*]

Yi-Hua E. Yang, Weirong Jiang and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
{yeyang, weirongj, prasanna}@usc.edu

ABSTRACT

In this paper we present a novel architecture for high-speed and high-capacity *regular expression matching* (REM) on FPGA. The proposed REM architecture, based on nondeterministic finite automaton (RE-NFA), efficiently constructs *regular expression matching engines* (REME) of arbitrary regular patterns and character classes in a uniform structure, utilizing both logic slices and block memory (BRAM) available on modern FPGA devices. The resulting circuits take advantage of synthesis and routing optimizations to achieve high operating speed and area efficiency. The uniform structure of our RE-NFA design can be stacked in a simple way to produce multi-character input circuits to scale up throughput further. An n-state m-character input REME takes only $O(n \times \log_2 m)$ time to construct and occupies no more than $O(n \times m)$ logic units. The REMEs can be *staged* and *pipelined* in large numbers to achieve high parallelism without sacrificing clock frequency.

Using the proposed RE-NFA architecture, we are able to implement 3 copies of two-character input REMEs, each with 760 regular expressions, 18715 states and 371 character classes, onto a single Xilinx Virtex 4 LX-100-12 device. Each copy processes 2 characters per clock cycle at 300 MHz, resulting in a *concurrent throughput* of 14.4 Gbps for 760 REMEs. Compared with the automatic NFA-to-VHDL REME compilation [13], our approach achieves over 9x *throughput efficiency* (Gbps*state/LUT). Compared with state-of-the-art REMEs on FPGA, our approach also indicates up to 70% better throughput efficiency.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.2.0 [**Computer Communication Networks**]: General—*Security and Protection*

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

General Terms

Algorithms, Design, Performance, Security

Keywords

Regular expression, FPGA, BRAM, intrusion detection, finite state machine, NFA

1. INTRODUCTION

Regular expression matching (REM) is an important mechanism used by popular intrusion detection systems such as Snort [2] and Bro [1], *et al.*, 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, regular expression matching is becoming not just a bottleneck, but itself a vulnerability of the network security systems that use it [15].

Basic regular expressions are *regular languages* constructed with *character classes* over a fixed alphabet. A regular language offers three basic operators on the character classes: *concatenation* $(\cdot)$, *union* $(|)$, and *Kleene closure* $(*)$. Other common operators, such as *optionality* (?) and *quantified repetitions* ($\{\texttt{a},\}$, $\{,\texttt{b}\}$, $\{\texttt{a},\texttt{b}\}$), can be constructed by proper arrangements of the three basic operators. Since regular languages are exactly the class of languages that can be accepted by finite state automata, a basic regular expression matching engine (REME) can always be implemented as a finite state machine, either a non-deterministic finite automaton (RE-NFA) or a deterministic finite automaton (RE-DFA).

In a RE-NFA approach [7, 14], individual regular expressions are processed in parallel independently from one another. Each input character is sent to every state in every REME, while matching outputs are collected from all REMEs running in parallel. 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 [9, 5], or multi-character input and centralized character decoding [6, 5], *etc.*, can be applied to improve throughput and reduce the resource requirement of the overall design.

In an RE-DFA approach, several regular expressions are grouped ([17]) into a single 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 [8, 11, 10, 4, 3] are then applied to improve memory access efficiency and to reduce the total number of states, which usually suffer from quadratic to exponential explosion [17].

While both RE-NFA and RE-DFA approaches have their respective advantages and drawbacks, and could share the same performance enhancement techniques (such as multi-character input matching), in this study we focus only on improving the RE-NFA approach on FPGA. Specifically, our main contributions are the following:

1. We slightly modify the original RE-NFA architecture used in [7] and [14], resulting in a highly modular structure, easy to translate into FPGA circuits.
2. We propose an elegant, spatial and circuit-level approach to perform multi-character input matching, taking advantage of the simple and uniform structure of our RE-NFA architecture.
3. We propose a simple way to utilize block memory available on modern FPGA devices to perform centralized character classification. This greatly improves the area efficiency of our design.
4. We design a two-dimensional staging and pipelining organization of the REMEs. This allows us to localize signal propagation and to obtain better clock frequency.
5. We define a set of metrics to quantify the complexity of a general regular expression. This allows us to more accurately assess the complexity of the REMEs being implemented.

The rest of this paper is organized as follows. We discuss the background and prior work of RE-NFA on FPGA in Section 2. Then in Section 3 we describe our basic RE-NFA architecture on FPGA, and the algorithms we use to construct it. Section 4 explains the optimizations we apply to our design, in particular the *multi-character input extension*, *centralized character classification*, and *staging and pipelining* techniques. Section 5 discusses performance figures reported by FPGA implementation tools on patterns extracted from Snort rules. Finally Section 6 discusses the conclusions and future work.

2. BACKGROUND

2.1 Challenges of Hardware REM

There are two main challenges to performing REM on hardware:

1. Processing large numbers of patterns in parallel.
2. Obtaining high *concurrent throughput*.

Large number of patterns require more hardware resources, which in the case of RE-NFA are the amount of logic, registers, and block memory on-chip. As discussed in Section 1, in an RE-NFA approach the state transitions of different REMEs are parallel in nature, making RE-NFA particularly suitable to implement on FPGAs, which offers high circuit parallelism. Nevertheless, efficient use of logic resources on FPGA is still crucial to obtaining optimal REME capacity and achieving maximum clock rate.

Definition 1. *The* concurrent throughput T *of a circuit of N REMEs is defined as the throughput of the input stream processed concurrently by all N REMEs. The concurrent throughput of a REM system is the total volume of inputs* fully *processed per unit time by the system.*

The concurrent throughput of any REM circuit is determined by three factors: (1) the size/complexity of the regular expressions, (2) the amount of available resources, and (3) the achieved clock frequency. In general, to obtain the same level of concurrent throughput, a *larger* number of REMEs with *more complex* patterns require proportionally *more* resources (*i.e.*, number of slices). Similarly, for the same set of REMEs, it is always possible to increase the concurrent throughput linearly with respect to resource usage by replicating the REM circuit multiple times.

We note that the concurrent throughput is different from the *aggregated* throughput, which sums up the throughputs of individual REMEs in a REM system. For example, a REM system consisting of five REM circuits, each processing 1 Gbps input data, may have 5 Gbps aggregated throughput but only 1 Gbps concurrent throughput. When evaluating the performance of a REM architecture, the concurrent throughput must be used, and the resource usage must be normalized.

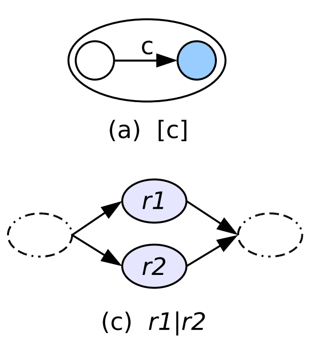
2.2 Prior Work

Floyd and Ullman [7] first studied the implementation of RE-NFA on hardware. They showed that, when translating directly from RE-NFA to integrated circuits, an N-state REME requires no more than $O(N)$ circuit area to implement. Sidhu and Prasanna [14] later proposed an algorithm and strategy to translate a regular expression directly into its matching circuit on FPGA. Both studies use the same RE-NFA structure at the circuit level, which is later used by most other RE-NFA implementations [9, 6, 5, 13].

Large-scale regular expression matching is first considered in [9], where the global distribution of character inputs is pipelined and broadcasted in a tree structure to reduce input stream fan-outs and to increase clock rate. The paper also proposed common prefix extraction and discussed the minimization of state transition logic. A multi-character decoder is proposed in [6] to processes multiple characters per clock cycle. It is also noted that character matching could be performed more efficiently at a centralized location; only a single bit of matching result must be sent to every NFA state. In [16] the authors describe a 2-phase procedure to generate NFA-based multi-character input REMEs. The algorithm transforms an n-state NFA with 2^i-character ($i \geq 0$) state transitions into an n-state NFA with 2^{i+1}-character state transitions in $O(n^3)$ time. This procedure is performed repeatedly to produce a 2^k-character input REME with any $k > 0$.

In [5] the use of shift registers and counters is proposed to implement efficiently different types of single-character repetition blocks (*Exactly*, *AtLeast*, and *Between*) on FPGA. The paper also utilizes other techniques, such as common prefix extraction and centralized character matching, to reduce resource usage to as low as 1.28 (4-input) logic cells per state. Although the implementation of backreference was mentioned in [5], its approach, which uses the *regular expression pattern* rather then the *input string* for backreference matching, actually negates the effect of the backreference in the first place. In [12] the authors explore the idea of pattern *infix* sharing to reduce the number of slices per REME across many similar patterns.

The first real backreference implementation on FPGA that we know of is done by [13], where block memory (BRAM) of the FPGA device is utilized to store the referenced strings.



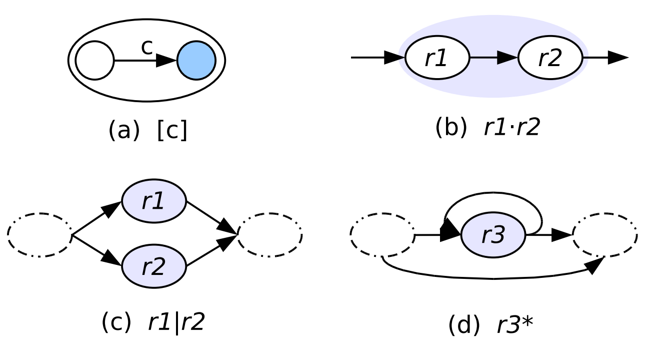


Figure 1: The McNaughton-Yamada construction. Extra nodes and ϵ-transitions (shown as dashed lines) are added for rules (c) and (d).

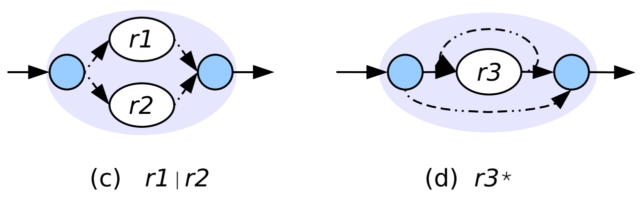


Figure 2: Modified McNaughton-Yamada construction. Note that in rules (c) and (d), the dashed ellipses are *not* part of the current construction.

The number of characters in all backreferences is limited by the amount of BRAM available, and backtracking is not supported due to the excessive memory size and logic complexity required by the feature.[1] The paper also features efficient use of counters for matching fixed-length repetitions.

3. BASIC ARCHITECTURE ON FPGA

We implement a regular expression matching engine (REME) on FPGA in three steps: (1) Parse the regular expression into a tree structure. (2) Use a *modified* version of the McNaughton-Yamada construction to construct an NFA with highly modular structure. (3) Map the resulting NFA into structural HDL suitable for FPGA implementation.

Our first step, converting a regular expression into a parse tree, is the same as that described in [7]. Steps (2) and (3) are unique from previous approaches, and are explained in the following subsections.

3.1 From Regular Expression to NFA

Instead of using the McNaughton-Yamada construction (see Figure 1), which generates an NFA with many intermediate nodes and redundant ϵ-transitions, we use a *modified* version of the construction, shown in Figure 2, to construct our RE-NFA state machine:

1. The union (|) operator does not conceptually combine its operands into a single unit (ellipse). Instead, inputs to the union are sent to each operand individually, while every operand sends its output independently to the output destinations of the union.

[1]Private communication with author, 28 April 2008.

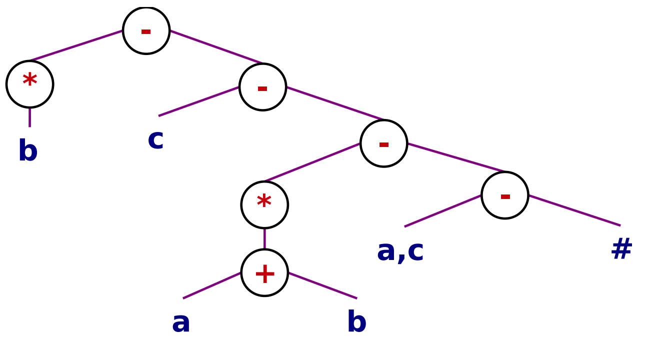


Figure 3: The parse tree for regular expression "b $*$ c (a|b) $*$ [ac]#".

2. The closure ($*$) operator neither prepends nor appends extra nodes for the pass-by transition. Instead, a pass-by signal is connected from the outputs of all immediate previous nodes to inputs of all immediate subsequent nodes.

As a result, neither the union nor the closure operator in the modified rules produces any extra intermediate nodes or redundant ϵ-transitions. Applicability of the modified rules (c) and (d) in the edge cases is ensured by adding extra `START` and `MATCH` states to the NFA when translating the root and the right-most leaf of the parse tree, respectively.

For example, using the original McNaughton-Yamada construction, the regular expression "b $*$ c (a|b) $*$ [ac]#" with the parse tree in Figure 3 is converted into an NFA in Figure 4. Using our modified construction, the parse tree is converted into a much more modular NFA in Figure 5.

3.2 From RE-NFA to HDL

As shown in Figure 5, all pairs of nodes inside the lightly shaded ellipses have an identical structure. Each of these pairs constitutes a "basic state block" for a single matching character. To translate the NFA (Figure 5) to a circuit on FPGA, all we need to do is to connect the inputs and outputs between different basic state blocks as specified by the state transitions.

As it appears, all basic state blocks can be instantiated from a single type of module (*e.g.*, an `entity` in VHDL) with only one parameter: the number of input ports, which

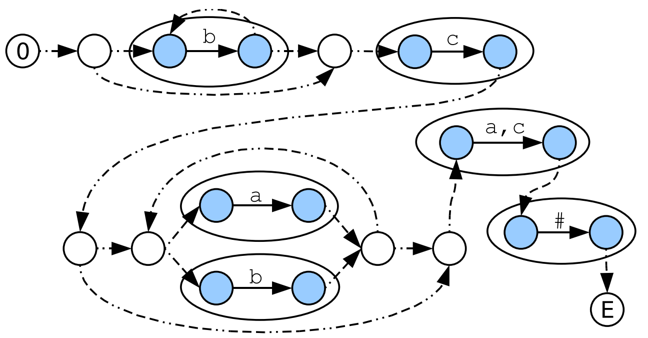


Figure 4: The NFA for "b $*$ c (a|b) $*$ [ac]#" constructed by the original McNaughton-Yamada rules specified in Figure 1.

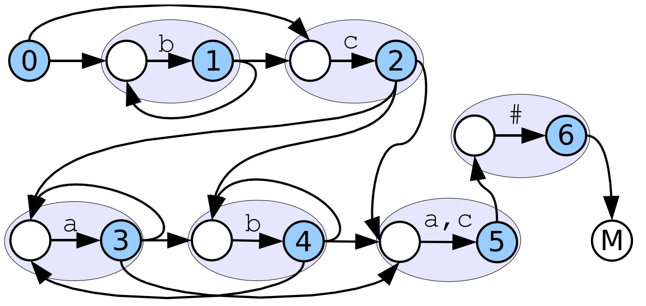


Figure 5: A "modular" NFA for "$\mathtt{b} * \mathtt{c}\,(\mathtt{a}|\mathtt{b}) * [\mathtt{ac}]\#$" constructed using the *modified* McNaughton-Yamada rules.

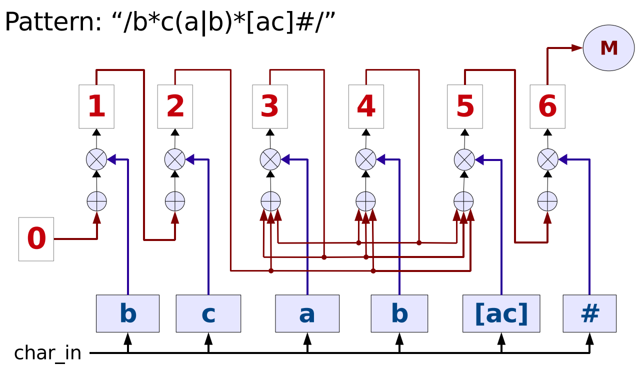


Figure 6: Circuits for matching "$\mathtt{b} * \mathtt{c}\,(\mathtt{a}|\mathtt{b}) * [\mathtt{ac}]\#$" constructed by mapping Figure 5 directly to HDL. The $\oplus$ and $\otimes$ symbols represent logic `OR` and `AND` gates, respectively.

is determined by the number of "previous states" that immediately transition to the current state. A single output port connects the output value of the current state to the inputs of the immediate "next states." Inside a basic state block, all inputs aggregate to a single `OR` gate, followed by a character matching via logic `AND` and a state value register.

The circuit for the regular expression "$\mathtt{b} * \mathtt{c}\,(\mathtt{a}|\mathtt{b}) * [\mathtt{ac}]\#$" is shown in Figure 6. On FPGA devices with 4-input LUTs and 2 LUTs per slice, 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 if $4 \leq k \leq 7$. This makes our RE-NFA circuits use no more than two LUTs per state. In practice, the number of LUTs we use per state is much closer to 1, and we rarely need an `OR` gate with $k > 7$ (less than 10 in 760 regular expression extracted from Snort rules).

3.3 BRAM-based Character Classification

In the basic state block module described above, the matching result of character input is received as a 1-bit signal (*true* or *false*) to the `AND` gate. The state transition logic does not otherwise care about what the matching character class is or what the matched input characters are; as a result, we do not distinguish between single character matching and character range or general character set matching, either. All of these are referred to as *character classification*, which takes one character as an input, and generates one bit of matching result.

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 character classification of an n-state RE-NFA can be completely implemented on a block memory of no more than $256 \times n$ bits. Furthermore, if two states use the same character class for matching inputs, they can share the same character classification output.

A drawback of this approach is that it would result in much redundancy in the block memory, especially for encoding simple character classes. With a straight forward implementation, the block memory would become the resource bottleneck when implementing a large number of REMEs even on modern FPGA devices (*e.g.*, Xilinx Virtex 4 series). Memory compression or further BRAM output encoding could be applied but with adverse effects on logic complexity and achievable clock rate. This problem is handled by centralized character classification described in Section 4.2.

3.4 Differences from Previous Approach

While our REM architecture is functionally similar to that of [14], 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 uniform structure is easy to construct, place and route on FPGA devices. As is discussed later in Section 4.1, individual circuits can be naturally *stacked* to form a circuit that matches multiple input characters per clock cycle.

3. Compared with [14], our REME construction does not attempt to minimize circuit logic at the HDL level. The *same* transition signal can be produced by *different* `OR` gates for different destination states, as shown in Figure 5 at the inputs of states 3 and 4.

Note that the last point above does not actually make our circuits any less efficient. Since HDL syntheses generally perform their own circuit-level optimizations, such redundancy will be evaluated and reduced if necessary to meet the implementation constraints.

4. ARCHITECTURAL OPTIMIZATIONS

We apply three optimizations to improve our basic design: (1) multi-character input matching, (2) centralized character classification, and (3) staging and pipelining. While these concepts have been proposed in previous research [9, 6, 5], the techniques used here are unique to our design and take advantage of the modularity of the proposed architecture.

4.1 Multi-Character Input Matching

Previous designs of multi-character input matching [6, 16] 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* approach to construct multi-character input REMEs. A formal procedure is described in Algorithm 1. An example two-character input circuit for the regular expression "$\mathtt{b} * \mathtt{c}\,(\mathtt{a}|\mathtt{b}) * [\mathtt{ac}]\#$" is shown in Figure 7.

Algorithm 1 Construction of an $(m+p)$-character input regular expression matching engine.

```
INPUT: An $n$-state $m$-character input REME circuit $C_m$
    and an $n$-state $p$-character input REME circuit $C_p$ for
    the same regular expression.

OUTPUT: An $n$-state $(m+p)$-character input REME circuit $C_{m+p}$
    for the same regular expression.

BEGIN

  1. For each $i \in \{1, 2, \ldots, n-1\}$, 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 $C_m$; forward
           the AND gate output of the basic state block directly to
           the state output.
       (b) Disconnect the state output $i$ of $C_m$ from all state
           inputs of $C_m$, and re-connecting them to the state
           inputs $\{i_1, i_2, \ldots, i_t\}$ of $C_p$.
       (c) Disconnect the state output $i$ of $C_p$from all state
           inputs of $C_p$, and re-connecting them to the state
           inputs $\{i_1, i_2, \ldots, i_t\}$ of $C_m$.
       (d) The state $i$ of the combined circuit receives
           $(m+p)$ matching signals of $(m+p)$ consecutive
           characters. The first $m$ signals go to the original
           $C_m$ part, while the last $p$ signals go to the original
           $C_p$ part.

  2. Merge the START states of the original $C_m$ and $C_p$ into
     a single START state; connect the output of the merged
     START state to all destinations of the original START
     states.

  3. Merge the MATCH states of the original $C_m$ and $C_p$ into
     a single MATCH state; take the inputs of the original
     MATCH states and OR them as the input of the merged
     MATCH state.

  4. The resulting circuit is an $n$-state $(m+p)$-character
     input REME for the same regular expression.

END
```

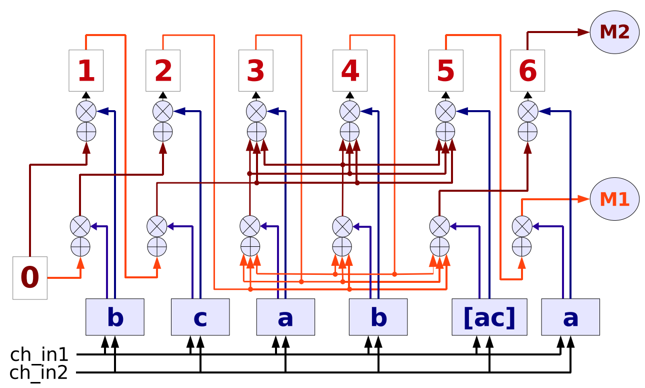


Figure 7: A 2-input matching circuit for the regular expression "b ∗ c (a|b) ∗ [ac]#".

Each application of the algorithm requires $O(n)$ time to build an n-state m-character input REME (as in step 1d of Algorithm 1), resulting in a circuit taking $O(n \times m)$ area. A single application of the algorithm on two copies of the same m-character input REME generates a $2m$-character input REME. Thus 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.

Lemma 1. *The circuit constructed by Algorithm 1 over two copies of a one-character input REME is a valid two-character input REME.*

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 1 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 1):

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 1.

3. From C_B's outputs to C_A's inputs, state transitions are also preserved by step 1c of Algorithm 1.

4. The state transition labels of both C_A and C_B are kept intact by step 1d of Algorithm 1.

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

Theorem 1. *Algorithm 1 constructs valid m-character input REME circuits.*

Proof. By the same arguments as for Lemma 1, a circuit constructed by Algorithm 1 over an m-character input REME and a one-character input REME is a valid $(m+1)$-character input REME. By induction, any circuit constructed by Algorithm 1 with an integer $m > 1$ is a valid m-character input REME. □

Compared to the previous approaches, our multi-character input construction is simpler, faster, and more flexible. It takes advantage of the synthesis and implementation optimizations performed automatically at the circuit level to generate an optimal multi-character input REME.

4.2 Centralized Character Classification

As discussed at the end of Section 3.3, when implementing character classification on the block memory of FPGA devices, the size of the memory can be a limiting factor on the number of REMEs that can be implemented. While there are many techniques to compress memory, we found that simply aggregating the character classifications from several REMEs into one place could by itself effectively solve the problem.

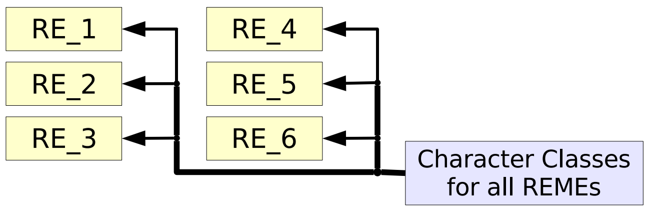


Figure 8: Centralized character classification for 6 different REMEs.

When constructing the centralized character classification, a function is called to examine and compare each state's character class to the character class entries collected in BRAM so far. If the character class of the current state is new, then a new entry (column of 256 bits) is added to the character classification BRAM; if it was old, then a proper connection is made from the BRAM output of the previous character class entry to the input of the current state.

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 $256 \times w$. In the worst case, w could be linear in n; in practice, however, we find w tends to grow much more slowly than $O(\log n)$. Note that prudently grouping REMEs can greatly help (slow down) the growth of w with respect to n.

An illustration of the technique is shown in Figure 8. Here a centralized block memory of character classes is shared by 6 different REMEs, preferably with many states matching the same character classes.

4.3 Staging and Pipelining

A common issue of most RE-NFA implementations on FPGA is the decline in achievable clock frequency with larger numbers of REMEs, supposedly due to the more complicated routing the synthesis and implementation software have to perform. This is especially true with techniques such as centralized character classification where a single character matching output can be used in many disparate states.

In our implementation we use an aggressive staging and pipelining structure to improve the clock rate. An example structure for 16 different REMEs is shown in Figure 9. As shown in the figure, the 16 REMEs are first divided into 2 pipelines; each pipeline is further divided into 2 stages. Every input character goes to a pipeline in the first clock cycle, then 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 at every stage in the pipeline.

Matching outputs of all REMEs are prioritized, with lower-indexed pipelines and lower-indexed stages having higher priority. Within a stage, matching outputs from different REMEs are priority-encoded into a single value of $\log N$ bits, N being the number of REMEs in the stage (2-bit in the example of Figure 9). The encoded matching output is buffered to the next stages and pipelines in the same way as the input character classifications. This allows for a single matching output from all 16 of REMEs at any clock cycle.

One drawback of this aggressive staging and pipelining is that there is a latency from the clock cycle when a last

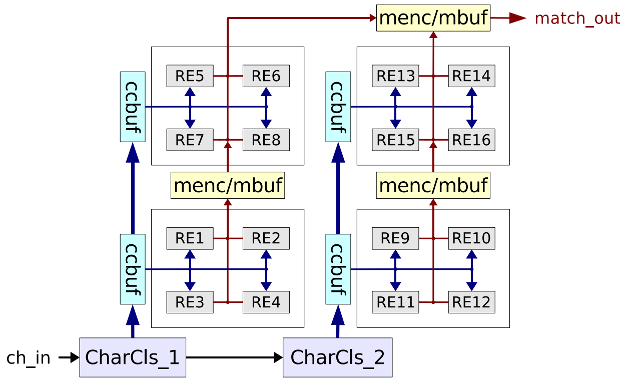


Figure 9: Structure of a staged pipeline for 16 different REMEs.

"matching character" is sent to the chip to the cycle when the "matching found" signal is asserted by the chip. For a structure of p pipelines and s stages per pipeline, the latency is a fixed $p+s$ clock cycles. We argue that, since this latency is fixed, it does not impact our ability to find exactly on which character the matching occurs.

5. PERFORMANCE EVALUATION

In this section we evaluate the performance of the proposed architecture. We first define a set of metrics in Section 5.1, to describe the complexity of an REME more accurately in Section 5.2. We then study the proposed architecture in Section 5.3 and compare our results with the state-of-the-art in Section 5.4.

5.1 Hardware Complexity of Regular Expressions

The one point that we must stress on here is that *all regular expressions are **not** created equal* when implemented on hardware. Thus, when measuring the efficiency of a RE-NFA architecture, it is very important to keep in mind the types and complexities of the regular expressions being implemented. We define the following metrics to quantify the complexity of regular expressions when implemented as hardware matching engines:[2]

State count - Total number of states needed by the regular expression matching engine (REME).

State fan-in - maximum number of states that can immediately transition to any state in the REME.

State fan-out - maximum number of states to which any state in the REME can immediately transition .

Loop size - total number of transitions within a loop of state transitions.

Branch-size delta - difference in number of transitions between two state transition paths with the same first and final states.

[2]The character sets to be matched at each state can also affect logic and routing complexity. Circuits for complex character sets take more logic to implement, while character sets matched at disparate states require longer signal routes. This problem is eliminated in our architecture by prudent use of block memory and pipelining.

Table 1: Snort rule categories whose regular expression patterns are (partially) implemented.

Snort rule cat. implemented	# of pat.	# of states	# of states/pat.
`backdoor`	154	3648	23.7
`chat/ftp`	18	163	9.1
`deleted`	23	607	26.4
`smtp/pop2/pop3`	22	306	13.9
`spyware-put`	446	11720	26.3
`web-misc/web-php`	51	1082	21.2
(others)	46	1052	22.9
760-REME	**760**	**18578**	**24.4**
523-REME	**523**	**13379**	**25.6**
267-REME	**267**	**6551**	**24.5**

The *state count* was used in [7] to describe the (linear) area requirement of a REME, implemented on IC, with respect to length of the regular expression. *State fan-in* and *state fan-out*, created by the use of union and Kleene closure operators, also affect logic complexity. We define state fan-in and state fan-out as the *maximum* number of signals entering and exiting any state, respectively, because the state machine runs at the speed of its slowest state transition. Note that in general, state fan-in and state fan-out could be unequal to each other. For example, for regular expression "`a(b|(c|d)e)f`", state fan-in is 2 (into states `e` and `f`), but state fan-out is 3 (out of state `a`); on the other hand, for "`a(b|c(d|e))f`", state fan-in is 3 (into state `f`) but state fan-out is 2 (out of states `a` and `b`).

The last two metrics, *loop size* and *branch-size delta*, would affect routing complexity when long and complicated regular expressions are implemented. In our experiments, however, most regular expressions from Snort rules have neither long loop nor widely varying branch sizes. Thus the precise quantization of these two metrics is beyond the scope of this paper.

In addition to the union and Kleene closure operators, most software implementations (*flavors*) of regular expression also add a number of features that are outside the regular language construction. The most important feature added by these software flavors is *backreference* – the ability to refer to a previously matched string as the string pattern to match against later inputs. Since backreference makes a pattern *context-sensitive*, it also makes the regular expression *non-regular*. Other extensions include software-based processing directives such as *anchors*, *lookahead* and *lookbehind*, and *conditionals*.

In this paper we focus only on the matching for expressions of *regular languages*, as opposed to the regular expressions extended by any software flavor. We thus avoid the use of backreferences and ignore the software-based directives in the patterns we use. In Snort, the majority of the rules can be described or reduced to regular languages without loss of correctness or authenticity.

5.2 Implemented Regular Expressions

We used patterns from Snort rules for our RE-NFA evaluation. The set of rules implemented are described in Table 1. The goal of this study, however, was not to implement all of Snort rules, but to offer a generic architecture for implementing *any* regular expression. Therefore, we avoided pattern-level optimizations (*i.e.*, patterns that are too short or too simple were avoided; patterns that are identical were counted as a single one even though they appear in different Snort rules). The rule selection guidelines are as follows:[3]

Table 2: Distribution of regular expressions versus state fan-in and state fan-out values.

value v	1	2	3	4	5	6	7	8	9
# of state fan-in = v	52	656	23	5	14	3	3	4	0
# of state fan-out = v	45	663	31	7	7	2	0	2	3

1. Select patterns that are of average lengths. Avoid patterns that are too simple, such as repetition of one or a few characters.
2. Count identical regular expression in different rules as a single one.
3. Avoid long repetitions, such as regular expressions containing "`[^\n]{256}`".
4. Avoid regular expressions that require backreference.

Guidelines 1 and 2 remove those regular expressions which would inflate the number of REMEs. Guideline 3 is chosen because long repetitions in our REME circuits are transformed automatically by the FPGA synthesis into shift-register structures (similar to [5]), which would inflate the number of states in our REMEs without increasing the circuit areas proportionally. Guideline 4 requires capability beyond regular languages and is beyond the scope of this paper; it also affects relatively few patterns in the Snort rule sets.[4]

We implemented 3 sets of REME circuits from the 760 regular expressions. The "760-REME" set contains all implemented regular expressions, and was the biggest circuit we produced (for the same m-character input). All regular expressions of rule numbers beginning with 1 or 5 went into the "267-REME" set; all regular expressions of rules numbers beginning with 1, 2, 3, 5, and 6 went into the "523-REME" set. The purpose here was to form 3 groups of regular expressions - each a super set of another - and see how adding more regular expressions affected scalability of our architecture. As shown below, our architecture achieved almost perfect scaling with respect to number of REMEs implemented on-chip.

The distribution of regular expressions with respect to state fan-in and state fan-out are shown in Table 2. Most regular expressions had values below 3, although a few reached as high as 8 or 9. Note, however, that when implementing in the same clock domain, the slowest REME determines the overall clock frequency.

[3]Most regular expressions in the `netbios` category either are too simple or contain backreferences. Many patterns in `deleted` are identical and too simple for our use. The `oracle` and `web-client` categories have many patterns with long repetitions.

[4]Out of nearly 2000 *distinct* regular expressions in Snort rules (Feb.2008), slightly more than 110 use backreferences, mainly in the `netbios` and `web-client` categories. In particular, the `netbios` category reuses a single regular expression with backreference in 174 different rules.

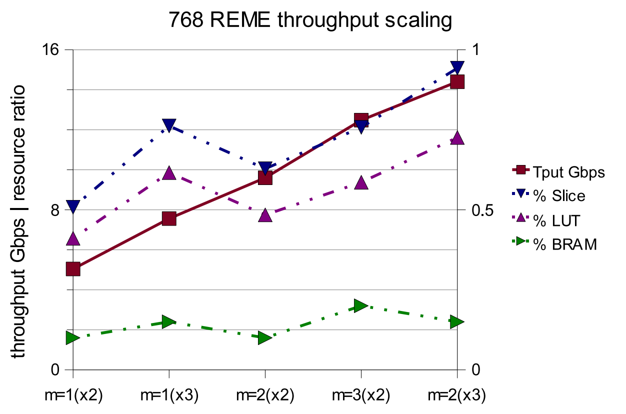


Figure 10: Throughput scaling of 760 REMEs on Virtex 4 LX-100-12. Squares (left scale) are throughput; triangles (right scale) are resource usage ratios.

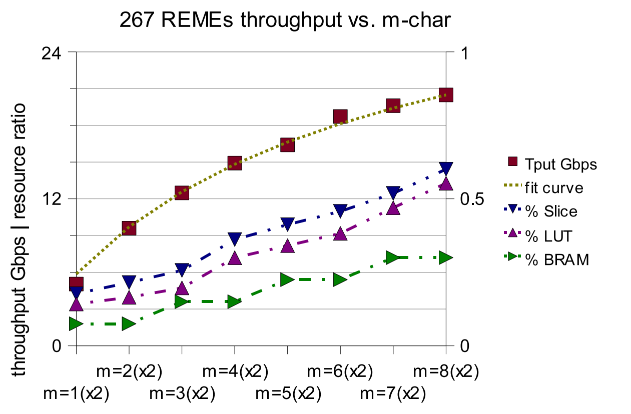


Figure 11: Throughput of 267 REMEs on Virtex 4 LX-100-12 vs. different m-character input sizes. Squares (left scale) are throughput; triangles (right scale) are resource usage ratios.

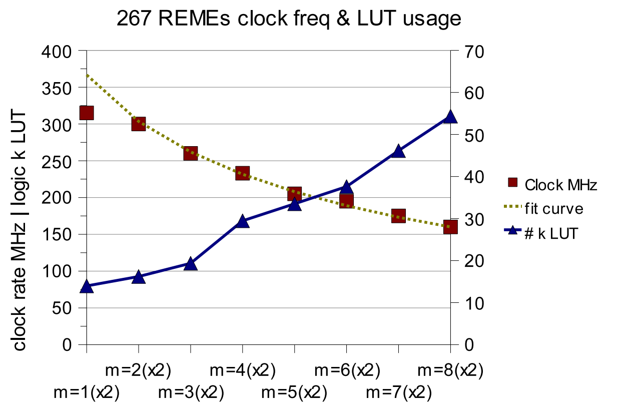


Figure 12: Clock rate and LUT usage of 267 REMEs on Virtex 4 LX-100-12. Squares (left scale) are clock rate; triangles (right scale) are LUT usage in thousands.

As discussed in Section 5.1, a state fan-out of 1 means the regular expression contains neither union nor Kleene closure (although it can still contain complex character classes). The number of regular expressions with state fan-in being 1 was less than those with state fan-out being 1, because in our implementations, the extra state fan-in produced by a Kleene closure on the first state of a regular expression (mostly in Snort `chat` category) could be safely ignored.

5.3 Implementation Results

To measure the performance of the "760-REME" (as in Table 1) circuit, we implemented it as 1- to 3-character input circuits and replicated the circuits 1 to 3 times on a single Virtex 4 LX-100-12 device. The idea was is to see how we could achieve higher throughput using increasingly more resources. The result is shown in Figure 10. The highest throughput was 14.4 Gbps, aggregated over 3 copies of $m = 2$ circuits.

There are two points to observe in this graph. First, multi-character input offers a viable way to scale throughput with *fewer* resources relative to circuit replication. The is evidenced by comparing `m = 1 (x3)` and `m = 2 (x2)` test cases (2nd and 3rd to the left, respectively), where scaling with multi-character input in the latter case not only obtains higher throughput but also uses *fewer* resources than scaling with only circuit replication.

To study the effect of multi-character input more closely, we implemented the "267-REME" set and scale m up to 8. We used the smaller set here only because 760-REME would take too may resources when implemented with $m > 4$. The results are in Figures 11 and 12. A few points to note about these graphs:

1. Comparing Figure 11 to Figure 10, the 267-REME tests achieve higher concurrent throughputs while using fewer resources than the 760-REME tests. This is expected since the 760-REME tests do almost 3x more work in parallel.

2. Figure 11 shows resource usage increases almost linearly with higher m-character inputs. However, concurrent throughput increases sub-linearly due to reduced clock frequencies at higher m values (see Item 4 below).

3. BRAM usage increases linearly with every two increments of m, because each dual-port block memory can serve the matching results of two input characters. The use of centralized character classification allows BRAM to stay underutilized in all cases.

4. In Figure 12, the clock frequency can be well approximated by $440\,\text{MHz}/\,(1 + 0.10 \log_4 m + 0.20m)$, where the 440 MHz in the numerator approximates the fastest switching frequency of a Virtex-4 LX device of speed grade -12, and the $0.10 \log_4 m$ and $0.20m$ in the denominator account for the additional gate and transmission delays, respectively. The clock frequency at $m = 1$ is limited by BRAM access and is excluded from the regression calculation.

In Item 4 above, it can be inferred that the "logic only" frequency without BRAM access is roughly $440/\,(1 + 0.2) =$

367 MHz when matching single-character inputs. This value coincides well with the maximum frequency achieved in [5] (362 MHz) on the same device technology (Virtex4-40-12).

Finally, as illustrated in Figure 10, we were able to put three copies of the 760-REME-2 circuit on a single Virtex 4 LX-100-12 device and still reach the same clock frequency, achieving 14.4 Gbps concurrent throughput for the 760 regular expressions.

5.4 Performance Comparison

Table 3 shows the comparison of our results with previous highest-throughput RE-NFA architectures on FPGA. We implemented single copies of our 760-, 523-, and 267-REME circuits on Virtex 4 LX-40-12 and calculated the throughput and LUT efficiency of the circuits according to Xilinx PAR reports. To compare performance across different implementations, we define *throughput efficiency* as follows:[5]

Definition 2. *The* throughput efficiency *of an REME circuit on FPGA, in units of* Gbps*state/LUT*, is defined as the concurrent throughput of the circuit divided by the number of LUTs the circuit uses per state.*

Comparing "760-REME-2" to Bispo *et al.* [5], our architecture required roughly the same amount of LUT per state but achieved almost 65.5% higher throughput (4.8 Gbps vs. 2.9 Gbps), due to our efficient architecture to match multiple input characters per clock cycle. In [5] the authors used counters to match repeating characters, which helped them reduce total LUT usage. We note that this technique is also applicable to our basic architecture but was *not* applied to our implementations here (see Section 6). They also performed common-prefix extraction on similar NFAs to reduce the total number of states. However, it is not clear to us that this *pattern-level* property should be treated as *architectural* improvement, since the same effect could have been accomplished by joining the common-prefix regular expressions by a union operator with distinct `MATCH` states. Instead of obscuring the difference between REM *pattern* and REM *architecture*, we focused on the architectural property in handling *arbitrary mix* of regular expressions.

Comparing "523-REME-4" to Clark *et al.*-4 [6], our architecture used 29% less LUT per state (2.2 vs. 3.1) while achieving slightly higher throughput (7.46 Gbps vs. 7.0 Gbps).[6] Although their throughput was also improved by multi-character input matching, they didn't describe a complete procedure to construct the multi-character input circuits automatically. While their analysis on the resource usage of the *resulting* circuits is in agreement with our findings (Section 4.1), it is not clear how simple or complicated the *constructing* process is to build their multi-character input circuits.

These REME implementations report the total number of *non-meta characters* in the patterns. However, this value was of no significance to us due to our use of BRAM for character classification (Section 3.3). As shown in Table 1 and discussed in Section 5.1, what most significantly affected our resource usage was the number of character-matching *states*, which were 5-25% lower than the number of non-meta characters, depending on how complex the character classes were. In the right-most two columns of Table 3, however, we assumed every state in our architecture equal to only one non-meta character reported by others.

Comparing our results to those obtained by the NFA-to-FPGA compilation in Mitra *et al.* [13], we achieved over 9x (3.3 vs. 0.34) the throughput efficiency. Although in [13] the authors reported a high throughput of 12.8 Gbps for 200 REMEs, the value was *aggregated* over 16 circuits, each consisting of up to 14 REMEs in parallel. On the same device (Virtex 4 LX-200) we would be able to replicate the "267-REME-4" circuit 24 times and obtain $\sim$ 180 Gbps concurrent throughput for 267 REMEs (the scalability was actually limited by number of available I/O pads). To compare the throughput efficiency in [13] with others, we used their *single-stream* throughput and estimated their use of LUT per state. Note however [13] use an automatic PCRE-to-VHDL compilation. They also implemented (partially) PCRE backreference and the SGI RASC core service modules to interface with a host, both of which are ignored in other research but can have negative impact on the throughput efficiency.

Because the REMEs in [16] are implemented on a different type of FPGA (Altera Stratix II EP2S180), their LUT usages and clock rates are not directly comparable to ours. Their (6-input) LUT usages are much lower than (4-input) ours, but they also achieved 20%-50% lower clock frequencies than our design. It takes $O\left(n \times \log_2 m\right)$ time to construct an n-state, m-character input RE-NFA in our approach, while it takes $O\left(n^3 \times \log_2 m\right)$ in [16]. The spatial transformation used by us is also more flexible, in the sense that an m-character input circuit with any $m \in \mathbb{N}$ can be produced in our approach, but only if $m = 2^k$ in [16].

6. CONCLUSION AND FUTURE WORK

We presented a novel and compact architecture for high-performance REM on FPGA. We used a modified McNaughton-Yamada construction to build the REME circuit, which was modular, uniform, and easy to map to structural HDL for FPGA implementation. Our design utilized the BRAM on modern FPGA devices to achieve high REME density. Copies of the same REME circuit can be stacked spatially to match multiple input characters per clock cycle. A 2-dimensional staging and pipelining architecture is used to localize signal routing and to achieve high clock frequency.

A number of improvements can be applied to our REME design. We can group REMEs into stages more intelligently to exploit pattern-level properties such as the common prefix extraction. We can also separate simple and complex regular expressions into different clock domains available on an FPGA device. Supposedly a pipeline of simple REMEs can be clocked higher than a pipeline of complex REMEs. It is also interesting to see how backreference could be implemented in our architecture, although due to the theoretical complexity of backreferencing, only to a limited degree.

[5]The throughput alone is insufficient because it does not account for the length and complexity of every regular expression. The LUT efficiency alone, on the other hand, does not reflect clock frequency or multi-character input.

[6]However, we are using a newer device (Virtex 4) than the one used in [6] (Virtex II).

Table 3: Comparison of our REME implementation with previous results.

	# non-meta char.	Multi-char. m	Tput. (Gbps)	# LUT per *state*	**Tput. efficiency (Gbps**state*/LUT)**
760-REME-2	∼ 20k	2	4.8	1.27	**3.8**
523-REME-2	∼ 15k	2	4.88	1.24	**3.9**
523-REME-4	∼ 15k	4	7.46	2.2	**3.4**
267-REME-4	∼ 8k	4	7.5	2.25	**3.3**
Bispo *et al.* [5]	19580	1	2.9	1.28	**2.3**
Clark *et al.*-1 [6]	17537	1	2.0	1.7	**1.9**
Clark *et al.*-4 [6]	17537	4	7.0	3.1	**2.3**
Mitra *et al.* [13]	N.A.	1	12.8/16	∼ 2.3	∼ **0.35**

7. REFERENCES

[1] *Bro Intrusion Detection System. http://bro-ids.org/.*

[2] *SNORT.ORG. http://www.snort.org/.*

[3] Michela Becchi and Patrick Crowley, *"A Hybrid Finite Automaton for Practical Deep Packet Inspection"*, CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference (New York, NY, USA), ACM, 2007, pp. 1–12.

[4] ———, *"An Improved Algorithm to Accelerate Regular Expression Evaluation"*, ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (New York, NY, USA), ACM, 2007, pp. 145–154.

[5] Joao Bispo, Ioannis Sourdis, Joao M.P.Cardoso, and Stamatis Vassiliadis, *"Regular Expression Matching for Feconfigurable Packet Inspection"*, FPT '06: Proceedings of the IEEE International Conference on Field Programmable Technology, 2006., Dec. 2006, pp. 119–126.

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

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

[8] Christopher L. Hayes and Yan Luo, *"DPICO: A High Speed Deep Packet Inspection Engine Using Compact Finite Automata"*, ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (New York, NY, USA), ACM, 2007, pp. 195–203.

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

[10] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese, *"Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia"*, ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (New York, NY, USA), ACM, 2007, pp. 155–164.

[11] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner, *"Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection"*, SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications (New York, NY, USA), ACM, 2006, pp. 339–350.

[12] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang, *"Optimization of regular expression pattern matching circuits on FPGA"*, DATE '06: Proceedings of the conference on Design, automation and test in Europe (3001 Leuven, Belgium, Belgium), European Design and Automation Association, 2006, pp. 12–17.

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

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

[15] R. Smith, C. Estan, and S. Jha, *"Backtracking Algorithmic Complexity Attacks against a NIDS"*, ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference, 2006., Dec. 2006, pp. 89–98.

[16] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya, *"High-Speed Regular Expression Matching Engine Using Multi-Character NFA"*, FPL '08: Proceedings of the International Conference on Field Programmable Logic and Applications, 2008., Aug. 2008, pp. 697–701.

[17] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz, *"Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection"*, ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (New York, NY, USA), ACM, 2006, pp. 93–102.