Hardware Implementation for Scalable Lookahead Regular Expression Detection

Masanori Bando, N. Sertac Artan, Nishit Mehta, Yi Guan, and H. Jonathan Chao

Department of Electrical and Computer Engineering
Polytechnic Institute of NYU, Brooklyn, NY

Abstract—Regular Expressions (RegExes) are widely used in various applications to identify strings of text. Their flexibility, however, increases the complexity of the detection system and often limits the detection speed as well as the total number of RegExes that can be detected using limited resources. The two classical detection methods, Deterministic Finite Automaton (DFA) and Non-Deterministic Finite Automaton (NFA), have the potential problems of prohibitively large memory requirements and a large number of concurrent operations, respectively. Although recent schemes addressing these problems to improve DFA and NFA are promising, they are inherently limited by their scalability, since they follow the state transition model in DFA and NFA, where the state transitions occur per each character of the input. We recently proposed a scalable RegEx detection system called Lookahead Finite Automata (LaFA) to solve these problems with three novel ideas: 1. Provide specialized and optimized detection modules to increase resource utilizations. 2. Systematically reordering the RegEx detection sequence to reduce number of concurrent operations. 3. Sharing states among automata for different RegExes to reduce resource requirements.

In this paper, we propose an efficient hardware architecture and prototype design implementation based on LaFA. Our proof-of-concept prototype design is built on a fraction of a single commodity Field Programmable Gate Array (FPGA) chip and can accommodate up to twenty-five thousand (25k) RegExes. Using only 7% of the logic area and 25% of the memory on a Xilinx Virtex-4 FX100, the prototype design can achieve 2-Gbps (gigabits-per-second) detection throughput with only one detection engine. We estimate that 34-Gbps detection throughput can be achieved if the entire resources of a state-of-the-art FPGA chip are used to implement multiple detection engines.

I. INTRODUCTION

Regular Expressions (RegExes) can represent complex conditions with simple notations. A wide range of applications, from Network Intrusion Detection and Prevention Systems (NIDPS) [1], [2] to compilers [3] and DNA multiple sequence alignment [4], [5] adopt RegExes. In particular, NIDPSs Bro [1] and Snort [2] and Linux Application Level Packet Classifier (L7 filter) [6] use RegExes to represent attack signatures or packet classifiers. Finite automata (FA) are widely used for the RegEx detection problem and FA consists of states and transitions that connect states to each other. The RegEx detection starts from an origin state and continues making transitions for each input character to the next states, which are determined by the current states and the input character. If the number of transitions triggered by an input is only one, the FA is called Deterministic Finite Automata (DFA); otherwise, it is called Non-deterministic Finite Automata (NFA). DFA is fast because the detection time complexity per input character is constant. However, to guarantee the constant detection time complexity, DFA pre-computes and allocates many extra states, leading to a prohibitively large storage space requirement in many cases. NFA, on the other hand, allows multiple concurrent state transitions per input character. These concurrent state transitions decrease the detection speed, but they also reduce the number of states compared to DFA. Sometimes, NFA is implemented on hardware using logic gates allowing parallelized state transitions to maintain reasonable speed. This implementation method often limits NFA-based approaches to hard-coded RegEx set on the logic gates. As a result, rule updates for such systems need logic gate reconfiguration.

Lookahead Finite Automata (LaFA) [7] is an NFA-based architecture. Yet, it differs from NFA by introducing three ideas to mitigate the problems of NFA. 1. Specialized and optimized detection modules to detect different parts of a RegEx, namely RegEx components, breaking away from simple automaton with per-character state transitions. 2. Systematically reordering the RegEx detection sequence to provide new venues for memory and logic resource optimization. 3. Sharing states among automata for different RegExes to reduce resource requirements. Thus, RegEx updates are much easier with LaFA, which only requires memory content updates as contrasted to traditional NFA, which requires logic gate reconfigurations. LaFA is also compact and can fit into a fraction of a single chip. In this paper, we propose an efficient hardware architecture and prototype design based on LaFA with various optimizations for its implementation.

The remainder of this paper is organized as follows. The next section introduces basic principles of the LaFA architecture. Section III introduces our LaFA hardware implementation with the detailed architecture of individual building blocks. In section IV, we evaluate our hardware prototype. Section V discusses related Regular Expression work. Section VI concludes the paper.

II. LOOKAHEAD FINITE AUTOMATA (LAFA)

LaFA is a finite automaton for RegEx detection that is optimized for scalability. Our focus in this paper is the hardware implementation of LaFA. Before discussing the LaFA hardware implementation, let us briefly summarize LaFA
architecture. LaFA construction processes are illustrated with an example summarized in figure 1. Figure 1(a) shows three RegExes, and figure 1(b) shows the three NFAs corresponding to these three RegExes. LaFA uses three methods to achieve memory efficient high-speed RegEx detection. Firstly, LaFA takes advantage of well-studied exact string detectors [8]–[12] that require minimum memory and logic. From the NFA, the states and transitions corresponding to the characters of simple strings are identified and merged into super states as shown in figure 1(c). An exact string detector scans input text against a signature table and generates a match signal if any of the signatures in the table appears in the input text. This match signal is called an Event in this paper. By using an exact string detector, input to the RegEx detection system is reduced from per character to per signature (exact string). This first contribution (Event-based detection) opens up the possibility to use detection sequence reordering, which is our second method. The detection sequence is reordered according to their characteristics, and detects more specific components before less specific ones. This is shown in figure 1(d). RegEx notation can represent wider ranges of characters, which is called character class. For instance, character class [a-z] represents any lowercase alphabetical character from “a” to “z”. Thus, 26 characters can be matched out of 256 ASCII characters to this character class, whereas only one character can be matched in an exact string/character match. Thus, we say exact character match is more specific than character class match. LaFA detects more specific and less common components before more common and less specific wider range RegEx contents. In other words, it is easy to detect exact character/strings and they are also expected to appear less frequently so we detect them first, whereas others are harder to detect (for instance, character classes) and they are likely to appear more frequently. As a result, reduces the number of transitions. In this paper, exact string is also called Simple String to distinguish it from non-exact string such as the character class called Variable String. Finally, our third method is a novel approach to reduce the number of states. The states of components that can be shared among RegExes are merged in figure 1(e). Traditional FA creates states by rules, even though some states are never activated simultaneously such as [a-z] among the three example RegExes. LaFA shares such states among multiple RegExes using specially designed modules. Using the three methods above, LaFA imposes fewer memory requirements. The embedded memory in an FPGA chip is more than enough to store the entire LaFA data structure, eliminating any need for external memory.

A. LaFA Architecture

In this section, we present the basic architecture of LaFA, as shown in figure 2. LaFA consists of two main building blocks: the Detection Block and the Correlation Block. The Detection Block is responsible for detecting the simple strings and variable strings in the input string. The Correlation block correlates detected Events to determine whether the input matches any of the RegExes in its RegEx set. The rest of this section illustrates the operation of these two building blocks with various examples.

B. Correlation Block

The Correlation Block maintains detection sequence and timing. The Correlation Block also holds information of each component in the RegExes. The information of each component is called Node Information, and the place to hold that node information is called Node Information Memory. Node information memory is described in section III.

1) Detection Sequence and Timing Verification: LaFA reduces memory requirements and gains detection speed via Event-based detection, more specifically, by separating detection and correlation processes. As a trade-off, LaFA needs to verify the detection sequence and timing whenever an Event is generated. The sequence of input character and arrival time is shown below along with two example RegExes, RegEx1 and RegEx4. We add another RegEx, RegEx4, to our original example for a more general case. The unit of time is defined as the interval required to receive one character.

\[
\begin{align*}
\text{RegEx1:} & \quad abc \{a-z\} ap \quad | \quad S_1 \quad V_1 \quad S_2 \\
\text{RegEx4:} & \quad abc \{a-z\} \{x\} \{y\} \quad | \quad S_1 \quad V_2 \quad S_3 \\
\text{Input:} & \quad a \ b \ c \ e \ f \ g \ x \ y \ z \\
\text{Time:} & \quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
\end{align*}
\]

Fig. 3. Example of simple string-based detection operations and timing verifications. The figures are sorted in chronological order. (VS=Variable String)

Initially, the Active RegEx List is as shown in Figure 3(a). When the component is active, LaFA performs a transition;
otherwise, it ignores the Event. In the figure, nodes are represented as boxes with “0” or “1”. “0” symbolizes that the node is inactive while “1” is the symbol of an active node. The simple string \( S_1 \) : “abc” is the first component in both \( \text{RegEx1} \) and \( \text{RegEx4} \); therefore, they are always active, analogous to the initial state of an NFA. Upon detecting a simple string, \( S_1 \) : “abc” at time 3, the simple-string detector generates an Event and passes it to LaFA. When the Event is received, based on the RegEx stored in the node, the next simple strings, i.e., \( S_2 \) : “op” of \( \text{RegEx1} \) and \( S_3 \) : “xyz” of \( \text{RegEx4} \) are activated by setting the bit corresponding to \( \text{RegEx1} \) and \( \text{RegEx4} \) to active (“1”) as shown in Figure 3(b). Next, at time 9, \( S_3 \) is detected and the detection time of \( S_3 \) is verified based on the distance between \( S_3 \) and \( S_1 \). If the timing verification is successful, then, as Figure 3(c) shows, a query is sent to the variable string detection module to verify the RegEx element \( V_2 \). The processing time of the detection module is denoted by \( \Delta \). In our prototype design, \( \Delta \) for all modules are three clock cycles, which is shorter than the detection time of simple strings, excluding short simple strings, so the result of the variable string detection will be ready before the detection of the following simple string. Hence, after finishing the processing and successful verification by the detection module, a RegEx match is generated as Figure 3(d) shows.

2) **Node Information:** As we see in figure 3, detection timing and sequence are verified. To perform this process, the pointer to the next state and the detection timing is stored in node information memory.

### C. Detection Modules

One aim of this paper is to investigate efficient architectures to detect variable strings, and we will not discuss detail of well-studied exact string detectors [8]–[13]. We will show five variable string modules: **Time Lookup Module (TLM)**, **Contents Lookup Module (CLM)**, **Repetition Detection Module (RDM)**, **Frequently appearing Repetition detection Module (FRM)**, and **Short simple string Detection Module (SDM)**. Except for RDM, all modules use previously detected character histories to verify a queried variable string. Thus, those modules are named **Buffered Lookup Modules**. For some repetition types of RegEx components, it is more suitable to process incoming characters in real time. Thus, this detection type is named as **In-line Lookup Module**, and RDM is of this type. Each module has special characteristics and is optimized to detect one or a few particular variable string types. Using these five variable detection modules, any variable string can be detected efficiently. The following section describes these modules with examples.

1) **Time Lookup Module (TLM):** TLM detects character classes and negated character classes with no repetition such as \([a-z]\) or \([0-9]\). TLM incorporates an input buffer, which stores characters recently received from a packet in chronological order. Using this buffer, TLM can answer queries such as **Does the character at time \( t \) belong to a (negated) character class \( CC \)?** Let us consider the detection process of \( \text{RegEx1} \) as an example. For the detection of \( \text{RegEx1} \), a lowercase alphabetical character (i.e., from \( a \) to \( z \)) must be detected between simple strings “abc” and “op”. Let us assume for the time being that detection sequence and detection timing is already verified by the correlation block. The last portion that needs to be verified is whether the input character between “abc” and “op” (i.e., the input character at **time 4**) was a lowercase alphabetical character or not. In the example, since at **time 4**, \( x \), a lowercase letter, appearing in the input, \( \text{RegEx1} \) is detected.

\[
\text{RegEx1: abc}-a-z\text{lop} | S_1 V_1 S_2
\]

\[
\begin{array}{c}
\text{Input: a b c x o p} \\
\text{Time: 1 2 3 4 5 6}
\end{array}
\]

2) **Contents Lookup Module (CLM):** CLM detects negated single characters with repetitions such as \(['a'][3], ['a'][3,5]\), or \(['a'][3]\). Let us follow an example to see the detection procedure of the CLM using \( \text{RegEx5} \). We assume the detection order and timing are already verified at the correlation block. The CLM has multiple memory locations to store detection timestamps of each character. For example, the first character “a” is detected at **time 1**, so **timestamp 1** is stored for character “a”. **Timestamp 2** will be stored for character “b”, and so on. To match \( \text{RegEx5} \), “x” should not appear from **time 4** to **time 7**. Let us call this time period the **inviolable period**. To verify that information, the CLM uses timestamps stored previously. In the example, “x” was detected at **time 7** and the time is stored at character timestamp memory in the CLM. From this information, we can conclude that the example input does not match \( \text{RegEx5} \) because “x” was detected during the inviolable period (at **time 7**).

\[
\text{RegEx5: abc}-x[3,5]\text{lop} | S_1 V_2 S_2
\]

\[
\begin{array}{c}
\text{Input: a b c d e f x o p} \\
\text{Time: 1 2 3 4 5 6 7 8 9}
\end{array}
\]

3) **Repetition Detection Module (RDM):** The RDM detects character classes and negated character classes with repetitions such as \([a-z][3]\), \([a-z][3,5]\), and \([a-z][3]\). Repetition components consist of a base \((a-z)\) along with the minimum and maximum repetition boundaries (e.g., 3 and 5 for \([a-z][3,5]\)). The RDM is the only in-line detection module. The RDM consists of several identical sub-modules, and each sub-module can detect character classes and negated character classes with repetitions. These sub-module operate in an on-demand manner. Assume a RegEx detection process is in progress and the following component to be inspected is a character class or negated character class repetition. The correlation block sends a request to the RDM for the detection of this component. The request consists of Pattern ID and the Minimum and Maximum repetition boundaries, where the base is represented by a Pattern ID. The RDM assigns one of the available sub-modules for detecting this component. The assigned sub-module then inspects the corresponding input characters to see whether they all belong to the base range and if the number of repetitions is between the minimum and maximum repetition boundaries. Once number of repetitions reaches to the minimum repetition value, a next simple string
is activated. The next simple string inactivate when number of repetitions reaches to the maximum repetition value.

Let us consider an example of detecting \texttt{RegEx6} against character input \texttt{abcxop}. At time 3, simple string \texttt{S1:“abc”} is detected. The following repetition component \texttt{[a-z]3,5} is programmed in the RDM sub-module immediately and starts checking the input character. From time 4 to time 6 consecutively, the module receives three characters that match the base \texttt{(a-z)}. The minimum repetition value is satisfied, then the sub-module sends an activate signal to the next component. Now simple string \texttt{S2:“op”} is ready to receive an Event. However, this is still not the end. The sub-module continues checking the input characters. The following characters continuously match the base until the maximum repetition value is reached. If there is no match in this interval, the sub-module deactivates the next component, \texttt{S2:“op”} in the example case.

\texttt{RegEx6: abc[a-z][3,5]op | S1 V3 S2}

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
Input & a & b & c & x & x & x & o & p \\ 
\hline
Time & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 
\hline
\end{tabular}

4) Frequently Appearing Repetition Detection Module (FRM): The FRM can detect any type of repetition using a buffered lookup approach. When analyzing the RegEx sets, we rank popularity of base in the RegEx set. For instance, \texttt{[^n]\{r\}} is the base that most frequently appears in the Snort RegEx set. Both FRM and RDM verify repetition components, but FRM can be shared among multiple RegExes that helps to improve resource efficiency. Thus, in programming phase, the popular bases are programmed in the FRMs. The base is programmed in memory, so updating the base does not require logic modification. Each FRM has a counter that counts the number of consecutive input characters that match the base. When the sequence breaks, the count and timestamp are stored in a history memory inside of the FRM. Thus, for the query operation, we can find repetition behavior in a particular time range from the number of repetitions with the timestamp.

5) Short Simple String Detection Module (SDM): LaFA classifies simple strings into two types, Regular Simple String (simply called simple string) and Short Simple String. Shorter strings naturally generate more Events as described below. To reduce occurrence of Events generated by the simple string, LaFA uses another buffered lookup type of module called the SDM. To maximize the advantage of the Event-based detection approach, it is important to select an appropriate signature size (the number of characters in a regular simple string signature). Imagine that the signature table in the simple-string detector contains many one-character signatures. A simple-string detector may generate an Event for every single character input. (This is the reason that the traditional RegEx detection architectures suffer.) Assume all characters arrive with the same probability and the signature table has \( s \) kinds of signatures. The probability to generate an Event by receiving one character is \( \frac{s}{256} \). The probability will be reduced exponentially, \( \frac{s}{256^l} \), as the number of characters, \( l \), increases. We classify strings with at least \( l = 5 \) characters as a simple string and less than that is classified as a short simple string.

III. LaFA HARDWARE IMPLEMENTATION

In the previous section, we show the operation of two main building blocks of LaFA. In this section, we describe the detailed hardware implementation of LaFA building blocks and related implementation issues.

An overview of the entire LaFA block diagram is given in figure 4. LaFA consists of \( n \) structurally identical tracks. Before entering LaFA, input characters (8 bits per character) are processed at a simple-string detector that produces 16 bits of a matching result (an event), when it detects a simple string. The event is essentially a string ID (SID) corresponding to the detected string. Once these events reach LaFA, they are broadcasted only to the tracks that has this string. Note that some simple strings appear in multiple RegExes, but those RegExes are deployed in separate tracks. Since each track work in parallel, LaFA maintains constant processing time. The status of the detected string ID is checked in the active RegEx bitmap. Along with the status, node information of the corresponding string ID is fetched from the node information memory. The address of the node memory is stored in the pointer memory. Another operation is performed in parallel. LaFA proceeds with further detection operations if, and only if, the detected simple string is active and the detection timing is correct. Timing History Memory is used to retrieve the simple-string detection timing. A variable string is verified next in either a buffered lookup or in-line lookup type of module after satisfying timing and status. If the component is the end of a RegEx (e.g., \( V_1 \) is the end of a component in \texttt{RegEx} : \( S_1 V_1 S_2 \) after reordering since the detection sequence becomes \( S_1 = \leftarrow S_2 = \leftarrow V_1 \) and all components in the RegEx are matched, then LaFA generates a RegEx match signal as a result. The following sections show detailed hardware implementation of each building block.

A. Hardware Implementation of Correlation Block

The correlation block, shown in figure 4, consists of two sub-modules, timing history memory and node information memory. Let us start describing these two important data structures first.

1) Timing History Memory: Timing history memory consists of two parts as shown in figure 5. \texttt{timing storage memory}, stores the detection time of simple strings. \texttt{timing pointer memory}, stores pointers to the timing storage memory. These pointers are pre-programmed for each RegEx to relate the consecutive strings in the RegEx. For each simple string \( S_j \) in a RegEx, other than the first string, a pointer \( pr_j \), is programmed to the timing pointer memory to point to the timing storage memory location \( S_i \) (previous simple string in the RegEx). The pointers are used to correlate the detection time of two simple strings in a RegEx. Both memory blocks are indexed with string IDs.

2) Node Information Memory: The data structure of the node information memory, which consists of 96 bits, is shown in figure 6. The most-significant three bits (bits 95 to 93) are used to identify the module ID. For example, TLM, CLM, and RDM are assigned module IDs 001, 010, and 011, respectively. The 92nd bit indicates whether the pattern is negated (1) or
filter that is common to many detection modules. The modules of the timing pointer memory (correspond to SRegEx1 different tracks as shown in the figure 5. For the timestamp can be obtained at this location. From this time information, is detected, the timestamp of the previous simple string addressing in our prototype design. Note that although we only need 96 bits for the repetition type indicator is used to indicate if the repetition is a frequent repetition (FRM) or not (RDM). The repetition type indicator is set to 1, which means the input character matches a character class. The filter returns “1”, which means the input character matches the character class. The filter returns “0” for the input character “A” because “A” is not in the character class. Taking this approach, any character class can be represented using only 256 bits. Different character classes are programmed in different rows, where the rows are indexed with the pattern ID. This memory-based character class filtering approach can be reconfigured very easily by updating the bitmap contents, i.e., by updating memory.

3) Correlation Operation: Using the four memories, we walk though correlation operations by revisiting the detection of RegEx1. As shown in figure 5 detection timestamps 3 and 6 are stored at locations corresponding to simple strings S1 and S2, respectively, once these strings are detected. The contents of the timing pointer memory (correspond to S2) points to the location of the previous simple string, namely S1 in the timing storage memory. Note that the RegEx1 and RegEx4 both contain S1, so these two RegExes are programed in different tracks as shown in the figure 5. For the RegEx1, S2 points S1. For the RegEx4, S3 also points S1. Once S2 is detected, the timestamp of the previous simple string S1 can be obtained at this location. From this time information, timestamp S1, and the minimum and maximum times from node information memory, the Execute block computes and verifies the detection time. The Execute block also receives status of the simple string from the Active RegEx Bitmap. Putting all information together, the simple string S2 can be verified. Another port of the node information is used at the detection modules.

B. Hardware Implementation of Detection Modules

We start with describing a technique we call character class filter that is common to many detection modules. The modules dealing with both character classes and negated character classes use character class filter, which is a compact way to detect any character classes. A character class filter consists of multiple 256-bit bitmaps stored in block RAMs, where each bitmap represents a character class. Figure 7 shows an example character class filter that represents three character classes (\[a-z\], \[a-zA-Z\] and \[w\]). Let us look at the character class \[a-z\] as an example. Initially, the 256-bit bitmap corresponding to character class \[a-z\] is all set to zero. Then, the bits corresponding to the range \(\text{ASCII 97 to 122}\) are set to 1. The following example shows how this character class filter works. There are two pieces of information, an input character and a pattern ID, that are used to query the filter. As an example, consider the filtering operation of input characters “a” and “A” against the character class \[a-z\] specified by the pattern ID. For the input character “a,” the filter returns “1,” which means the input character matches the character class. The filter returns “0” for the input character “A” because “A” is not in the character class. Taking this approach, any character class can be represented using only 256 bits. Different character classes are programmed in different rows, where the rows are indexed with the pattern ID. This memory-based character class filtering approach can be reconfigured very easily by updating the bitmap contents, i.e., by updating memory.

Fig. 6. Contents of the Node memory in detail.

Buffered Lookup Modules

All buffered lookup modules maintain up-to-date information in the buffers ready for a query. In our design, all buffers are implemented using dual-port memory to support simultaneous update and query. More specifically, one port (update port) is reserved for updates and the other port (query port) is used for query. The following part of the section describe the detailed architecture of each buffered lookup module.

1) Hardware Implementation of TLM: Figure 8 shows the block diagram of TLM. In each clock cycle, the received input
character is stored in a block RAM called character buffer through the update port. On demand queries are performed through the query port. For queries in the form Does the character received at time \( t \) belong to a (negated) character class \( CC \)?, TLM receives a time value \( t \) as the input and returns the character received at time \( t \) as a result. As shown in figure 8, the TLM consists of three stages. The TLM resolves target time (address) from the detection time of a string and string length at Stage 1 by subtracting the string length from the string detection time to find the time the character class is expected. The character at time \( t \) is obtained in Stage 2. The obtained character is compared with the target character class using the character class filter described previously to see whether the character at time \( t \) belongs to the character class. For the prototype design, we reserve 7 bits of a pattern ID that can support up to 128 different patterns in each TLM. This number of patterns can be incremented by adding more block RAMs.

![Fig. 8. Block diagram of TLM (CC:Character Class)](image)

2) Hardware Implementation of CLM: CLM is efficiently designed to detect negated character repetition components. To realize the functionalities described in Section II, we designed a hardware architecture shown in figure 9. CLM stores timestamps of every ASCII character. Thus, 256 memory locations are reserved, each for one ASCII character. Two comparators are implemented: one to compare the timestamps with the minimum timing, and the other one with the maximum timing. The timestamp memory and two comparators together are named Comparison Block in the figure. For the query, we simply compare the stored timestamps against the inviolable period described in section II-C2. If the timestamp does not overlap with the inviolable period, we can conclude the query character is NOT detected during the target range. From this process, negated character repetition can be verified.

To prevent timestamp overwrites before it is needed, \( C \) Comparison Blocks are implemented. The operations of the comparison blocks are described as follows: A timestamp of an input character is stored in memory one-by-one. In other words, the first timestamp of a character is stored in the first memory (BRAM-1 in the figure), the second timestamp of the character is stored in the second memory (BRAM-2), and so on. Since every character arrives at different times, one head pointer used to track the first update point of each character. The update is applied only for the memory to which that pointer is pointing. The pointer loops among these \( C \) memory locations. In this process, the oldest timestamp is overwritten by the latest one. Thus, the CLM always maintains \( C \) latest timestamps for all characters (256 ASCII characters).

The \( C \) comparison-blocks have independent memories and comparators, so \( C \) comparisons are performed simultaneously.

![Fig. 9. Block diagram of CLM](image)

In-Line Lookup Modules

3) Hardware Implementation of RDM: The RDM consists of character class filters and counters. A counter is incremented when an input character is matched with the target character class. The RDM resets the counter and frees the submodules for new queries if there is any mismatch in the input. As an in-line module, RDM must verify the input character from immediately after the simple string. To match the timing gap, the input characters are delayed fixed number of clocks in registers waiting for necessary information such as pattern ID, Maximum Repetition, and Minimum Repetition. The delayed character inputs are screened using the character class filter. The match signal goes to the repetition checker. The repetition checker performs the timing verification process described in section II-C3. Supporting multiple simultaneous repetition components, \( R \) repetition checkers are implemented.

![Fig. 10. Block diagram of RDM (CC:Character Class)](image)

Additional Buffered Lookup Modules

4) Hardware Implementation of FRM: LaFA takes buffered lookup approaches to verify frequently appearing repetitions. Multiple repetition components sharing the same base can share one FRM, making efficient use of resources. For example, base \( \{\text{"\n\input\r\}1000, \text{"\n\r}\} \) appears very frequently in various repetition types \( \{\text{"\n\input\r\}1000, \text{"\n\r}\} \). The detailed architecture is shown in figure 11. A character class filter is used to detect a base. The consecutive repetition is counted in the counter. The count and timestamp are stored in a history memory when the consecutive sequence is broken. In the prototype design, we keep \( F \) history in one FRM. For query operation, 24-bit comparators verify the repetitions and timing. 

5) Hardware Implementation of SDM: The block diagram of the simple-string detector is illustrated in figure 12. The
SDM detects short simple strings. We define a short simple string as a string with less than five characters. The SDM architecture is very similar to the TLM except that in the SDM, there are multiple memory blocks to support multiple character lookups at a clock cycle.

For a programming operation, these memories are accessed one-by-one and store the character. Refer to the architecture in figure 12. The first character is stored in the first memory (BRAM-1), the second character is stored in the second memory (BRAM-2), and so on. After storing the fourth character in the fourth memory (BRAM-4), the fifth character is stored in the first memory but in the second row. Once all memory is used (reaching to the last row), the operation starts overwriting the first memory locations. In our prototype design, eight thousands characters (32 kbits) of history data can be stored in a SDM. For a query operation, the target address of the memories (BRAM-1 to BRAM-4) is calculated based on the length of previous strings, the length of the short simple string, and the detection time. Four characters will be retrieved from the memories based on the address information. The retrieved characters will be aligned and compared with the target short simple string pattern. The patterns of the short simple strings are stored in short simple string pattern memory, located in the left bottom of the figure. It stores the four ASCII code of each short simple string (32 bits = 8 bits*4 characters) and four don’t-care bits. The don’t-care bits are used for the short simple string with a length less than four characters. For example, if the short simple string is only two characters, the comparator only compares the first two characters. The reference patterns and the set of characters retrieved from the memories are compared. If all result returns match, then the SDM concludes that the short simple string is verified.

To evaluate the operational speed and resource requirements of LaFA, we implemented LaFA on a Xilinx Virtex-4 FX100 FPGA using the Xilinx ISE 10.1.3 design suite. The Virtex-4 FX100 FPGA has 42,176 slices and 376 block RAMs. The design is downloaded on the FPGA on a board PLDA XpressFX [14]. Input to the LaFA is generated through a synthesizable test-bench that is placed on the same FPGA chip. The proof-of-concept prototype is implemented using following settings mentioned in section III: \( n=3, C=8, R=2 \), and \( F=4 \) based on our simulation results [7]. The prototype design is running at a clock frequency of 250MHz, which is equivalent to 2Gbps detection throughput per engine. Logic and memory utilization are summarized in table I for each module in one track. The prototype design has three tracks and consumes only 3,835 slices and 96 blocks RAMs, which is 7% and 25% of the total resources, respectively, showing the resource efficiency of LaFA. As expected, the block RAM usage dominates the resource consumption of LaFA. As a proof-of-concept design, we avoid any further optimizations on the prototype. For instance, block RAM in the FRM can be replaced with distributed RAMs, reducing the block RAM requirement without a significant impact on the logic usage.
achieve 2Gbps of performance using a fraction of memory on one FPGA. Duplicating the LaFA engines on one FPGA chip can achieve higher throughput. For instance, recently released Xilinx Virtex-6 FPGA contains 32Mbits of block RAM. Since 17 LaFA engine can fit in the FPGA, we expect the total throughput can reach to 34Gbps. The simulation results for RegEx sets of different sizes demonstrates that memory requirements of LaFA increases nearly linear [7].

TABLE II
LAFA SIMULATION RESULT FOR MULTIPLE REGEX SETS.

<table>
<thead>
<tr>
<th>RegEx Sets</th>
<th># of Concurrent Transitions</th>
<th>Memory Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnortMisc</td>
<td>49</td>
<td>40/6</td>
</tr>
<tr>
<td>SnortWebCL</td>
<td>532</td>
<td>313/6</td>
</tr>
<tr>
<td>SnortSpy</td>
<td>630</td>
<td>390/8</td>
</tr>
<tr>
<td>SnortComb</td>
<td>1537</td>
<td>423/9</td>
</tr>
<tr>
<td>Bro</td>
<td>827</td>
<td>263/3</td>
</tr>
<tr>
<td>LinuxL7</td>
<td>111</td>
<td>328/5</td>
</tr>
</tbody>
</table>

VI. RELATED WORK

Due to multiple concurrent transitions, NFA-based implementations will be too slow in software. Thus, researchers proposed implementations on hardware with a high level of parallelism [12], [17]–[21]. Although such schemes achieve fair RegEx detection speed by using solely logic gates, the inflexibility of implementing signatures on logic gates limits the updatability and scalability of NFA implementations.

DFA-based implementations are also actively being investigated. The main focus of these approaches [15], [16], [22]–[28] is to reduce the memory requirement by reducing the number of states and transitions. These approaches replace the problematic parts of DFA with NFA or other architectures to minimize memory consumption. However, after those efforts, the DFA memory requirement and scalability issues are still not completely solved. A detailed evaluation of these approaches can be found in [7].

VI. CONCLUSIONS

In this paper, we describe a detailed architecture and implementation of the Lookahead Finite Automata (LaFA). LaFA is a resource-efficient single chip RegEx detection system that achieves scalability and high-speed by introducing specialized and optimized detection modules, systematically reordering detection sequences, and sharing states among detectors. The first two contributions reduce the “multiple concurrent transition problem” in NFA and the third one reduces hardware resource (memory and logic) usage. The entire LaFA data structure is implemented on block RAMs to allow easy updates.

Our proof-of-concept prototype design demonstrates that LaFA is resource-efficient and the resources of a fraction of a single commodity FPGA chip is enough to accommodate up to twenty-five thousand (25k) RegExes using LaFA. The prototype also demonstrates that 2-Gbps detection throughput can be achieved using a single LaFA engine. We estimate that 34-Gbps detection throughput can be achieved if the entire chip resources are used to implement multiple detection engines with a single state-of-the-art FPGA chip.

REFERENCES