International Journal of Innovative
Computing, Information and Control
Volume **8**, Number **12**, December **2012**

ICIC International ©2012 ISSN 1349-4198
pp. **8367–8386**

# A MULTI-CHARACTER TRANSITION STRING MATCHING ARCHITECTURE BASED ON AHO-CORASICK ALGORITHM

Chien-Chi Chen and Sheng-De Wang

Department of Electrical Engineering
National Taiwan University
No. 1, Sec. 4, Roosevelt Road, Taipei 10617, Taiwan
{ ajaxchen; sdwang }@ntu.edu.tw

Received September 2011; revised January 2012

Abstract. *A hardware string matching architecture is usually used to accelerate string matching in various applications that need to filter content in high speed such as intrusion detection systems. However, the throughput of the hardware string matching architecture inspecting data character by character is limited by the achievable highest clock rate. In this paper, we present a string matching architecture based on the Aho-Corasick algorithm. The proposed architecture is able to inspect multiple characters simultaneously and the throughput of string matching can be multiplied. We first describe an intuitive algorithm to construct a multi-character finite state machine (FSM) that accepts multiple characters per transition based on an Aho-Corasick prefix tree (AC-trie). Then we propose an architecture for multi-character transition string matching consisting of multiple matching units for processing the transition rules that are generated from the derived multi-character FSM. The design of proposed architecture utilizes the properties of the failure links of an AC-trie to reduce the transition rules derived from the failure functions linked to the initial state. As a result, the state growth rate is moderate in the number of the derived multi-character transition rules as the number of the characters inspected at a time increases. The proposed architecture was implemented on an ASIC device for evaluation and the resulting throughput can achieve 4.5 Gbps for a 4-character string matching implementation operated at 142 MHz clock.*
**Keywords:** String matching, Finite state machine, Aho-Corasick algorithm

1. **Introduction.** String matching generally including exact string matching and regular expression matching is used in many applications. Exact string matching is more efficient though less flexible for searching keywords in a text as compared with regular expression matching. Aho and Corasick [1] have proposed a multiple-pattern string matching algorithm (AC-algorithm) that can locate all occurrences of multiple keywords in a string in a one pass search. The AC-algorithm has excellent performance for exact string matching. Some applications like network intrusion detection systems (NIDS) that need to inspect a data stream on line usually first use exact string matching to filter out expected data quickly, then check the filtered out results further by other more complicated and slower approaches, such as regular expression matching. For example, Snort [2], which is an open source project of NIDS, first uses the AC-algorithm to quickly filter out potentially malicious packets, then uses regular expression matching to check the filtered out packets further.

The network bandwidth has ever been increasing with the advances of fiber communication and integrated circuit technologies. In order to keep up with the throughput of the network, hardware accelerators of string matching are needed to speed up the throughput of the packet inspection in NIDS. However, most hardware implementations

are character based string matching architectures that inspect the packet data character by character. The throughput of a character based string matching architecture is limited by the operating clock. The advances of the semiconductor technology make it come true that much more devices can be put in the same area. However, on the other hand, the operating speed of the integrated circuit does not increase significantly. To take the modern CPU as an example, a single CPU can contain multiple cores while the operating clock only increases slightly. In addition, the higher the clock rate is, the more the power consumption is increased. Moreover, to increase the clock rate it needs more sophisticated circuit design technique to develop the device. Therefore, in order to speed up the throughput of the hardware accelerator of string matching while not to increase the operating clock, we need to design a multi-character string-matching engine that can inspect multiple characters in every clock cycle.

We will focus on the hardware string-matching architecture based on the AC-algorithm. The following problems need to be considered when designing a multi-character string-matching engine based on the AC-algorithm. First, the AC-algorithm is a character-based algorithm that can process only a character in a transition. We need to develop a systematic algorithm to convert an AC-trie to a multi-character finite state machine (FSM) that accepts the same keyword set. After that we can design a multi-character string-matching engine based on the derived multi-character FSM. Secondly, the multi-character FSM derived from an AC-trie should be able to deal with the alignment problem. We use a simple example to explain the alignment problem here. Considering the keyword "he" and the input text is "xhex", in which 'h' and 'e' do not appear in the beginning and the ending of the input text respectively, which means the keyword "he" does not align with the input text. Thirdly, when an AC-trie is converted to a multi-character FSM, the number of transitions will be increased explosively. Therefore, we need to develop some techniques for suppressing the growth of the transitions.

In this paper, we propose an approach to construct a multi-character FSM from an AC-trie and an architecture to implement the derived multi-character FSM. We first present an intuitive and efficient algorithm for generating multi-character transition functions that represent a multi-character FSM based on an AC-trie. In this algorithm, we use the 1-character transition functions of a deterministic finite automaton (DFA) converted from an AC-trie as basis and concatenate the 1-character transition functions iteratively to obtain all the multi-character transition functions of the multi-character FSM. The derived multi-character FSM can deal with the alignment problem naturally.

Our proposed hardware architecture for implementing the derived multi-character FSM consists of multiple matching units where each matching unit is responsible for one multi-character transition function. A transition function is represented as a transition rule that includes both the pattern and the matching result in the implementation. The proposed architecture is similar to the architecture of a Ternary Content Addressable Memory (TCAM), while the matching units of the proposed architecture store both the patterns and matching results and the output stage is implemented by a priority multiplexer instead of a priority encoder. In the proposed architecture, the transition functions represented in negation expressions is reduced by using the don't-cared character in patterns and assigning different priorities for transition rules. Consequently, the growth rate of the multi-character transition rules can be suppressed. For example, when implementing an $n$-character transition FSM in a lookup table, the growth of the required space is $256^n$ generally. While implementing the $n$-character transition FSM in our proposed architecture, the growth of the required space is about $n \times 5.7^n$ for 1000 keywords. The proposed architecture is evaluated in FPGA synthesis tools using ASIC devices. The results show that the operating clock is about 142 MHz for an implementation of five

hundred 4-character matching units. It is equivalent to an implementation of a 1-character transition string matching engine operated on 568 MHz. The main contributions of this paper can be summarized as follows.

– To the best of our knowledge, the algorithm proposed by this paper is the first algorithm using concatenating method to derive a multi-character FSM from an AC-trie. In addition, the derived multi-character FSM can resolve the alignment problem naturally.

– The proposed architecture for implementing the derived multi-character transitions is simple and regular and is also efficient in space.

Another advantage of the proposed architecture is that when the keyword set is changed it only needs to generate new transition rules for the new keyword set and update the contents of the matching units with the new generated transition rules. The hardware architecture does not need to be re-configured for the new keywords. The proposed architecture is not restricted to be implemented in a specific programmable or reconfigurable device like FPGA. The structure of the proposed architecture is simple and regular so that it can be easily implemented as a standalone integrated circuit (IC) chip.

The organization of this paper is as follows. Section 2 includes the related work for string matching. Section 3 describes how to construct multi-character transitions from an AC-trie. Section 4 describes the proposed architecture of the multi-character matching engine and how to generate the multi-character transition rules for the architecture. Evaluation and discussion are in Section 5. Finally, Section 6 gives conclusions and possible future work.

2. **Related Work.** There are many hardware architectures based on the AC-algorithm for accelerating string matching. The hardware architectures based on the AC-algorithm can be roughly divided to two categories. Since the AC-algorithm is a memory exhausted algorithm one category of the approaches focus on improving the efficiency of hardware utilization. The other category of the approaches focus on improving the throughput of string matching, such as increasing the clock rate of the hardware or, moreover, inspecting multiple characters simultaneously to multiply the throughput of string matching.

Using the lookup table to implement the AC-algorithm is most intuitive while subject to a poor hardware efficiency. To improve the hardware efficiency, Alicherry et al. [3] proposed an architecture consisting of TCAM and SRAM to implement the AC algorithm that utilizes the property of ternary matching of TCAM to achieve the matching of characters expressed in negation expressions. As a result, the space required for the transitions can be reduced. Pao et al. [4] and W. Lin and B. Liu [5] proposed pipeline architectures to implement the partial trie that only contains goto functions of the AC-trie so that it can reduce the space induced by failure functions. N. Hua et al. [6] proposed another approach based on a block-oriented scheme instead of usually byte-oriented processing of patterns to reduce the memory usage.

To achieve high-speed exact string matching, D. P. Scarpazza et al. [7] proposed an optimized software approach for a multi-core processor that splits keywords to fit in the local memories of the processing cores such that it can reach very high overall throughput. The throughput of the string matching engine that inspects one character every clock is limited by the clock rate. If the string matching engine can inspect multiple characters simultaneously every clock then the throughput can be multiplied. Y. Sugawara et al. [8] proposed a string matching method called suffix based traversing (SBT) that is an extension of the AC-algorithm to process multiple input characters in parallel and to reduce the size of the lookup table. The article of Alicherry et al. [3] also proposed a $k$-compressed AC DFA to achieve a parallel $k$-character matching engine. A $k$-compressed AC DFA only consists of the states corresponded to the states whose depth is a multiple

of $k$ in the original AC-trie and the leaf states of the original AC-trie. G. Tripp [9] proposed an architecture consisting of parallel multiple FSMs to achieve multi-character transitions. All the FSMs in the architecture proposed by G. Tripp are identical to that are implemented by lookup tables based on the AC-algorithm. In the architecture, each FSM is responsible for matching each character respectively from the multiple input characters, and there is a specific logic for combining the matching results from every FSMs. V. Rahmanzadeh and M. B. Ghaznavi-Ghoushchi [10] proposed another parallel string matching approach in which search patterns are sliced into multiple interleaved substrings and matching the substrings by parallel individual FSMs. It then obtains the matching result by combining the outputs of the FSMs.

N. Yamagaki et al. [11] proposed an intuitive algorithm to construct a $2^k$-character NFA from a 1-character NFA that is converted from a regular expression. The algorithm of N. Yamagaki et al. eliminates the alignment problem by adding a self edge to the initial node and a new final node to each final node. The algorithm proposed by N. Yamagaki et al. is not related to the AC-algorithm, while our proposed approach for constructing a multi-character FSM is based on an AC-trie.

## 3. Constructing Multi-character FSM.

In this section, we describe the approach for constructing a multi-character FSM that is represented by the multi-character transition functions derived from an AC-trie. The derived multi-character FSM keeps the property of the AC-trie that the number of states is the same and exact one state is active at a time. In the later section, this algorithm is used to generate the multi-character transition rules used in our proposed hardware architecture.

### 3.1. Aho-Corasick algorithm.

Figure 1 shows an AC-trie for the keywords {he, she, his, hers}, which is an example of the paper of Aho and Corasick [1]. We use this AC-trie as an example to explain our proposed approach in this paper. In Figure 1, the physical lines represent the goto functions and the dotted-lines represent the failure functions linked to the non-initial states. State 0 is the initial state or the root state of the AC-trie. Every non-initial state has failure functions while the failure functions linked to the initial state are not shown. The non-empty matching outputs are shown beside the corresponding states. Each state in an AC-trie represents a unique string which is the prefix of one of the keywords forming the AC-trie. For example, the initial state represents an empty string, state 4 represents "sh", and state 9 represents "hers".

Aho and Corasick has described the fact that the goto and failure functions can be converted to next move functions ($\delta$) so that they represent a DFA-version AC-trie. The

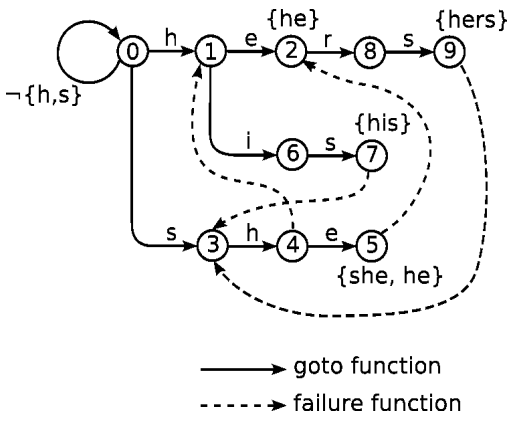


FIGURE 1. AC-trie for keywords {he, she, his, hers}

failure links are eliminated in the DFA-version AC-trie and an exact one state transition is made for each input character. The DFA-version AC-trie is convenient for the hardware implementation like the lookup-table approach. However, since the next move functions of each state include the goto functions of the states pointed to by the failure link, the number of the next move functions increases explosively and the space needed for the DFA-version AC-trie is much more than the space needed for the original AC-trie. For example, the next move functions of state 1 include two functions $\delta(1, \mathrm{e}) = 2$ and $\delta(1, \mathrm{i}) = 6$ which are its own goto functions and other three functions $\delta(1, \mathrm{h}) = 1$, $\delta(1, \mathrm{s}) = 3$, and $\delta(1, \neg\{\mathrm{h}, \mathrm{s}\}) = 0$ which are derived from the goto functions of state 0. For the sake of brevity, here we use "the transition functions of state $x$" to represent "the transition functions beginning from state $x$".

3.2. **Deriving multi-character transitions from AC-trie.** First, we define the next move functions of the DFA version of the AC-trie where each next move function accepts an input state and an input character then outputs a next state as 1-character transition functions and use them to derive the multi-character transition functions. The derived multi-character transition functions represent a multi-character FSM that accepts the same keyword set of the original AC-trie.

For the sake of convenience, an $n$-character transition function is denoted as $\delta_n(s, c) = s'$ that transits from state $s$ to $s'$ on an $n$-character symbol $c$. For example, $\delta_1(0, \mathrm{h}) = 1$ is a 1-character transition function which transits from state 0 to state 1 on character 'h', and $\delta_2(0, \mathrm{he}) = 2$ is a 2-character transition function which transits from state 0 to state 2 on characters 'he'.

Our proposed approach for deriving the multi-character transition functions from the 1-character transition functions is intuitive. We explain the approach by examples first. For example, concatenating the 1-character transition function $\delta_1(0, \mathrm{h}) = 1$ with its successive 1-character transition function $\delta_1(1, \mathrm{e}) = 2$ can obtain a 2-character transition function $\delta_2(0, \mathrm{he}) = 2$. Furthermore, concatenating the derived 2-character transition function $\delta_2(0, \mathrm{he}) = 2$ with its successive 1-character transition function $\delta_1(2, \mathrm{r}) = 8$ can obtain a 3-character transition function $\delta_3(0, \mathrm{her}) = 8$.

For another example, concatenating the 1-character transition functions $\delta_1(0, \neg\{\mathrm{h}, \mathrm{s}\}) = 0$ with its successive 1-character transition function $\delta_1(0, \mathrm{h}) = 1$ can obtain a 2-character transition function $\delta_2(0, \neg\{\mathrm{h}, \mathrm{s}\}\mathrm{h}) = 1$. Furthermore, concatenating the derived 2-character transition function $\delta_2(0, \neg\{\mathrm{h}, \mathrm{s}\}\mathrm{h}) = 1$ with its successive 1-character transition function $\delta_1(1, \mathrm{e}) = 2$ can obtain a 3-character transition function $\delta_3(0, \neg\{\mathrm{h}, \mathrm{s}\}\mathrm{he}) = 2$. Alternatively concatenating the derived 2-character transition function $\delta_2(0, \neg\{\mathrm{h}, \mathrm{s}\}\mathrm{h}) = 1$ with its another successive 1-character transition function $\delta_1(1, \mathrm{s}) = 3$ can obtain another 3-character transition function $\delta_3(0, \neg\{\mathrm{h}, \mathrm{s}\}\mathrm{hs}) = 3$. This transition means that the current state is 0 and the next state is determined by the third character 's' only. The 3-character transition function $\delta_3(0, \neg\{\mathrm{h}, \mathrm{s}\}\mathrm{hs}) = 3$ is an example showing that the multi-character transition functions derived by this approach solve the alignment problem naturally.

According to the examples described above, because there is a self linked transition of the initial state and all the non-initial states are eventually linked to the initial state through failure links in an AC-trie, there is no alignment problem for the derived multi-character FSM and no additional assistant state is required for constructing a complete multi-character transition. Consequently the states of the derived multi-character FSM are identical to the original AC-trie.

Figure 2 shows the example for deriving the 2-character transition functions beginning from state 1. There are five 1-character transition functions beginning from state 1. In

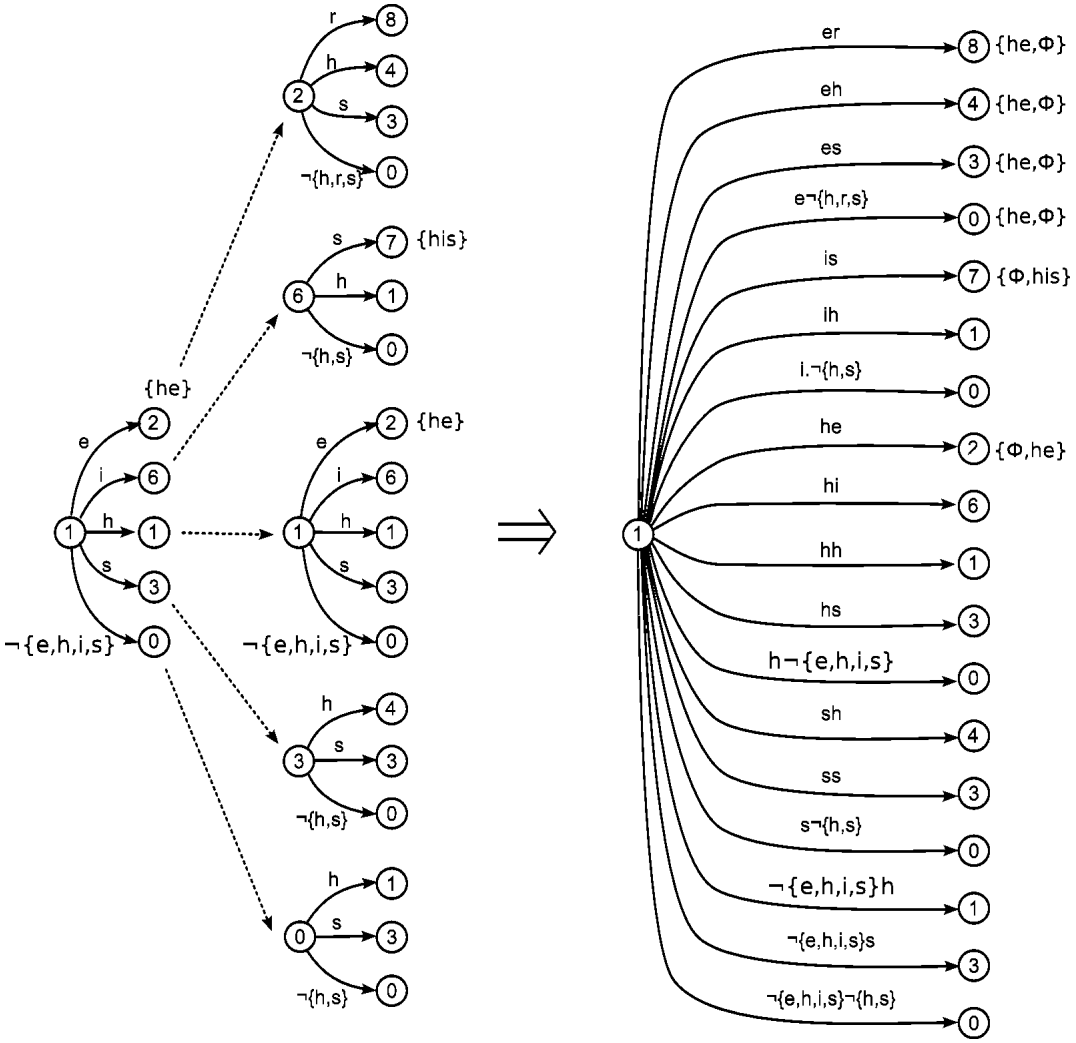


FIGURE 2. Deriving 2-character transition functions of state 1

which $\neg\{\mathrm{e,h,i,s}\}$ includes all the characters excepting 'e', 'h', 'i', and 's'. For the example of 1-byte character, $\neg\{\mathrm{e,h,i,s}\}$ should be 252 characters. Therefore, the transition function $\delta_1(0, \neg\{\mathrm{e,h,i,s}\}) = 0$ represents 252 transition functions that are linked to the initial state. For the sake of convenience, we use only one transition function to represent the 252 transition functions linked to the initial state.

The five 1-character transition functions of state 1 are $\delta_1(1,\mathrm{e}) = 2$, $\delta_1(1,\mathrm{i}) = 6$, $\delta_1(1,\mathrm{h}) = 1$, $\delta_1(1,\mathrm{s}) = 3$, and $\delta_1(1, \neg\{\mathrm{e,h,i,s}\}) = 0$. By concatenating $\delta_1(1,\mathrm{e}) = 2$ with the four transition functions of state 2, concatenating $\delta_1(1,\mathrm{i}) = 6$ with the three transition functions of state 6, concatenating $\delta_1(1,\mathrm{h}) = 1$ with the five transition functions of state 1, concatenating $\delta_1(1,\mathrm{s}) = 3$ with the three transition functions of state 3, and concatenating $\delta_1(1, \neg\{\mathrm{e,h,i,s}\}) = 0$ with the three transition functions of state 0 respectively, we can obtain all the 2-character transition functions of state 1.

When multiple characters are inspected simultaneously in each matching cycle, the information of passed states is hidden. Therefore, the matching outputs corresponding to the passed states have to be kept in the derivation process. For example, the labels shown beside the top four states 8, 4, 3, and 0 in the right part of Figure 2 are the output sets $\{\mathrm{he}, \phi\}$ which mean that the matching outputs corresponding to the first character 'e' and the second character 'r', 'h', 's', or $\neg\{\mathrm{r,h,s}\}$ are "he" and empty string ($\phi$) respectively. In addition, the label shown beside the state 2 is the output set $\{\phi, \mathrm{he}\}$ which means

that the matching outputs corresponding to the first and second characters 'h' and 'e' are empty string ($\phi$) and "he" respectively. For the sake of brevity, the output sets are not shown if all matched outputs are empty strings.

Following the procedure described above, concatenating every 1character transition function with its successive 1-character transition functions respectively can obtain all the 2-character transition functions for an AC-trie. Furthermore, concatenating every new derived 2-character transition function with its successive 1-character transition functions respectively can obtain all the 3-character transition functions for the AC-trie. Repeating the concatenating process iteratively can obtain the $n$-character transition functions for any required positive integer $n$. The derived $n$-character transition functions represent an $n$-character FSM which accepts $n$ characters in every transition.

```
Algorithm for constructing n-character transition set
Input.
   n: number of characters
   NXSET: 1-character transition set
Output.
   TRSET: n-character transition set
Method.
1. begin
2.   TRSET ← {}
3.   for each state Si do
4.   begin
5.     NSET ← all 1-character transitions of Si in NXSET
6.     repeat n-1 do
7.     begin
8.        TMPSET ← {}
9.        for each transition NXi in NSET do
10.        begin
11.           NX_ST ← next state of NXi
12.           for each transition NXj of NX_ST do
13.           begin
14.              NEW_TR ← concatenate NXi with NXj
15.              TMPSET ← TMPSET ∪ NEW_TR
16.           end
17.        end
18.        NSET ← TMPSET
19.     end
20.     TRSET ← TRSET ∪ NSET
21.   end
22.   return TRSET
23. end
```

ALGORITHM 1. Algorithm for deriving multi-character transitions

3.3. **Algorithm for deriving multi-character transitions.** Algorithm 1 shows a generalized algorithm for deriving the $n$-character transition functions from the next move functions of a DFA-version AC-trie. Using Algorithm 1 with the next move functions of the DFA-version AC-trie as 1-character transition set can derive the corresponding $n$-character transition set for any desired $n$. The input parameter $n$ is the number of characters to be inspected simultaneously. The input parameter *NXSET* contains the original 1-character transition set, and the result of this algorithm is the derived $n$-character transition set which is stored in variable *TRSET*. The algorithm consists of multiple level iterative loops to derive the $n$-character transition set for each state in the AC-trie.

At the beginning of algorithm, *TRSET* (line 2) is cleared. In the loop between line 3 and line 21, the $n$-character transition set for each state *Si* of the AC-trie is derived.

In line 5, all 1-character transition functions of state *Si* are duplicated to *NSET*. The loop between line 6 and line 19 is executed repeatedly $n-1$ times to concatenate the 1-character transition functions of *Si* with $n-1$ succeeding 1-character transition functions iteratively to obtain the $n$-character transition functions of *Si*. After executing the loop between line 7 and line 19 repeatedly $n-1$ times, *NSET* contains all the $n$-character transition functions beginning from *Si*. In line 20, add *NSET* to *TRSET*, and then go back to line 5 to continually process the next state. When all states are processed, the algorithm terminates.

Now we look into the loop between line 6 and line 19. In line 8, *TMPSET* is cleared. The loop between line 9 and line 17 expands each transition function *NXi* stored in *NSET*. In line 11, the next state of *NXi* is assigned to *NX_ST*. In the loop between line 12 and line 16, for each transition function *NXj* beginning from *NX_ST*, the transition function *NXi* is concatenated with transition function *NXj* to obtain new transition function *NEW_TR* in line 14 and the transition function *NEW_TR* is added to *TMPSET* in line 15. The number of pattern character of the transition function *NEW_TR* is one more than the number of the pattern character of the transition function *NXi*.

In a 1-character FSM, the matching output of each transition can be represented by the next state. However, in an $n$-character FSM for $n > 1$, if the $n$-character transition functions only output the next states, the information about the passed states will be hidden. Thus, we can only know the matching output corresponding to the last character, and the matching outputs corresponding to the preceding $n-1$ characters are lost. Hence, when using the derivation algorithm to derive the $n$-character transition functions, the matching results for every transition should be kept in the concatenating process, though the details are not included in Algorithm 1.

## 4. **Proposed Architecture.**

As an intuitive approach, the $n$-character FSM derived according to our proposed algorithm can be implemented by a lookup table. However, implementing the $n$-character FSM by a lookup table requires a huge amount of space to store the table because the space required for the lookup table grows exponentially to the number $n$.

For example, we can consider the generally used 1-byte character set whose size is 256. The AC-trie has nine states for the keywords {he, she, his, hers}. The lookup table for the 1-character transition functions should have $9 * 256 = 2,304$ elements. The lookup table for 2-character transition function should have $9 * 256^2 = 589,824$ elements, and each element includes two states. Furthermore, the lookup table for a 3-character transition function should have $9 * 256^3 = 150,994,944$ elements, and each element includes three states.

In the following, we propose an architecture for the derived $n$-character FSM that utilizes the property of the failure links in the AC-trie to reduce the required hardware cost. In an AC-tire, all the states are linked to the initial state eventually through the failure links. Therefore, the 1-character transition functions of the initial state are repeatedly included in the 1-character transition functions of every non-initial state. According to the definition of the AC-trie, two states linked by a failure link have a common suffix. When the failure function of a state is linked to the initial state, it means there is no matched prefix for any keywords, because the initial state represents an empty string.

Hence, when constructing the multi-character rules, we can determine the outputs and next state individually. For example, let us consider the case of matching two characters in parallel where the current state is 4 and the two input characters are “es”. The state will transit to state 5 on the first character ‘e’, and the matching output is “she”. For the next character ‘s’, because there is no transition on ‘s’ for state 5, following the failure

function of state 5 which points to state 2, the goto functions of state 2 will be matched. However, there is no transition on 's' for state 2 too and the failure function of state 2 points to state 0. Hence, the next state will be 3 which is determined according to the goto functions of the initial state. As the procedure described above, we can determine the matching output "she" by the current state 4 and the first input character 'e'. Since the next character 's' causes the failure function linked to the initial state, meaning that there are no matching prefix of any keywords, the matching process should go back to the initial state and start the matching process from the initial state. The next state is determined to be 3 by the initial state and the second character 's', meaning that there is a transition rule beginning from the initial state which matches the pattern "?s" and the next state should be 3.

According to the observation, we propose a hardware architecture which can effectively save the hardware cost. In the rest of this section, we first describe the hardware architecture and the transition rules used in the hardware architecture, and then explain how to derive the transition rules. At last, we use examples to explain the matching operations of the proposed architecture.

4.1. **Hardware architecture.** Figure 3 shows the overall hardware architecture of the multi-character transition string matching device and Figure 4 shows the block diagram of a matching unit.

The string matching device accepts $n$-character input IN_CHRS and produces $n$ matching outputs OPT1 to OPT$n$ that are corresponded to the first to the last characters of IN_CHRS respectively. The number $n$ is decided as required, such as 3, 4, or more. The more the characters are matched in every clock cycle the more the matching speed increases. However, the transition rules become more complicated and the hardware cost also grows.

The multi-character string matching device includes $m$ matching units, $n+1$ priority multiplexers, and $n+1$ registers. Each matching unit is responsible for executing one $n$-character transition rule. For the example as shown in Tables 1 and 2, there are 38 3-character transition rules and at least 38 matching units are required. Each matching unit matches its own pattern with current state CUR_ST and input characters IN_CHRS then outputs the matching results that include a next state NX_ST, $n$ matching outputs from OP1 to OP$n$, and control signals NX_FLG and OF1 to OF$n$. In it, each of matching outputs OP1 to OP$n$ is corresponding to each input character respectively.

The signals NX_FLG and OF1 to OF$n$ are control flags corresponding to NX_ST and OP1 to OP$n$ respectively. For example, when NX_FLG is '1', the data of NX_ST is valid, and when NX_FLG is '0', the data of NX_ST is not valid. For another example, when OF1 is '1', the data of OP1 is valid. While OF1 is '0', the data of OP1 is not valid.

The results of the matching units are sent to the $n+1$ priority multiplexers to determine the matching results of the current matching cycle. Priority multiplexers 0 to $n$ select the highest priority results to be sent to the outputs Dout and then to registers REG0 to REG$n$. Since the operations are synchronized with the clock signal CLK, the results of this matching cycle will be latched in registers REG0 to REG$n$ when the CLK is changed. REG0 is a state register storing the current state, and register REG1 to REG$n$ are output registers storing the $n$ matching outputs corresponding to the $n$ input characters.

The input signal SET_IN is designated for setting the contents of the matching units. The detail of the setting function is not shown in the block diagrams because the function is not the critical portion of this paper. When the keywords are changed, the generated new rules are sent to the matching units through the SET_IN.

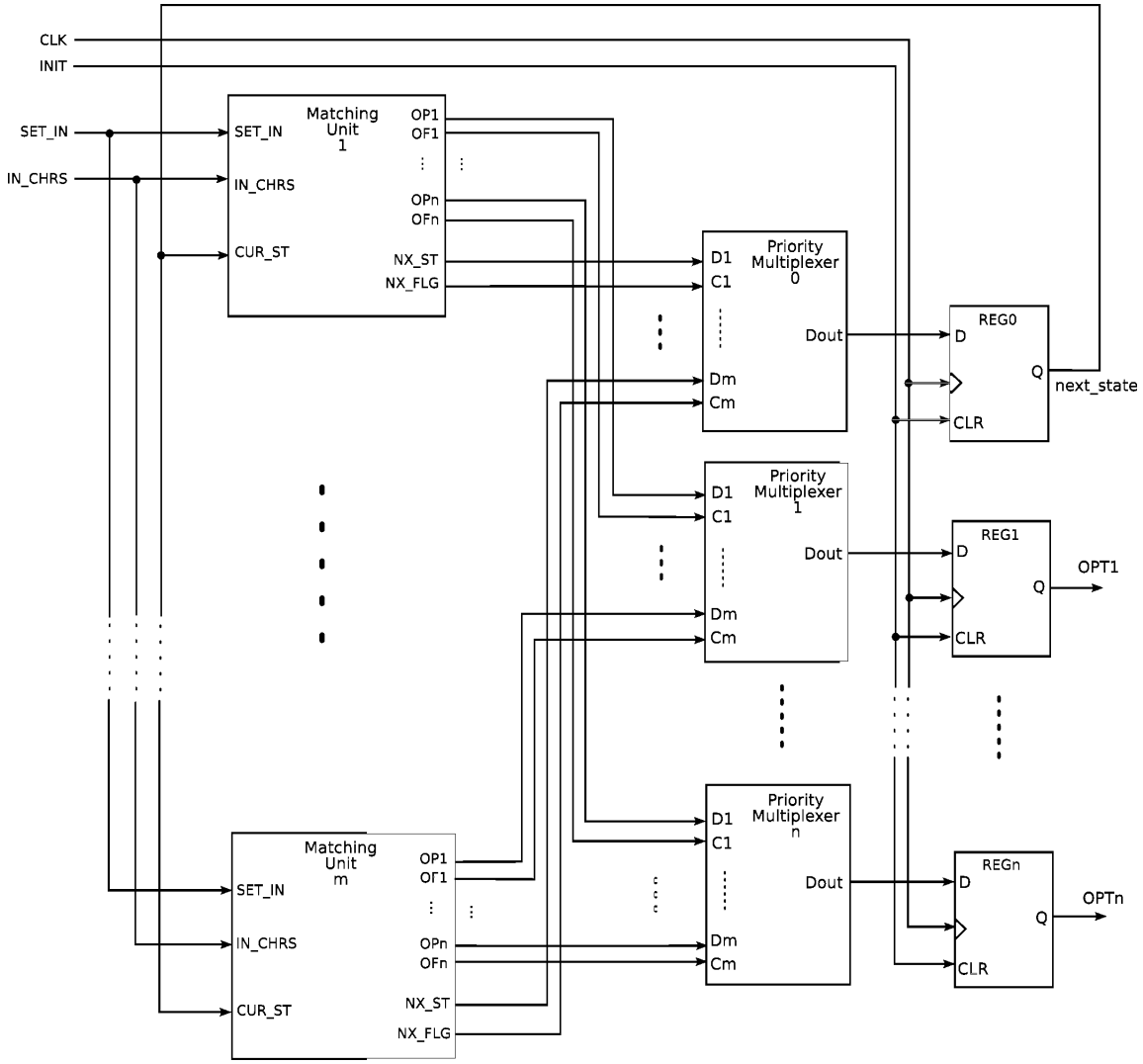


FIGURE 3. Hardware architecture of multi-character string matching

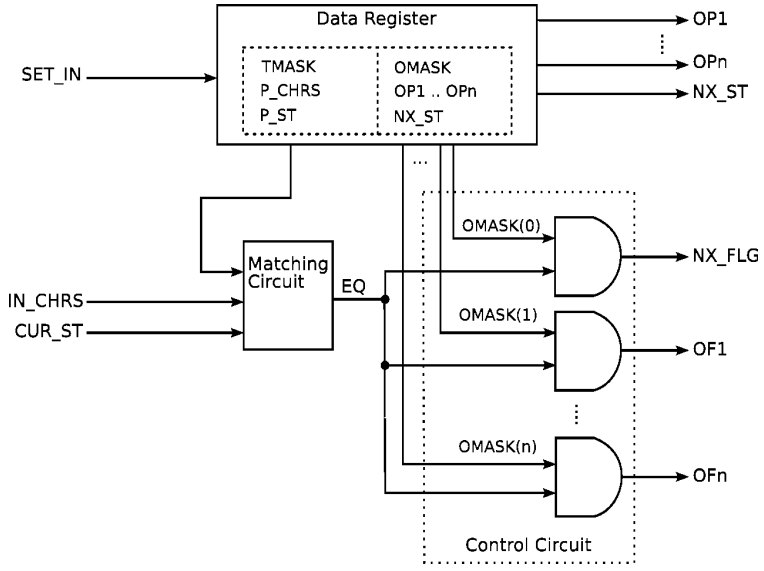


FIGURE 4. Architecture of matching unit

TABLE 1. 3-character transition rules 1 to 20

| rule no. | Pattern Data | | | Output Data | | | | |
|---|---|---|---|---|---|---|---|---|
| | TMASK | P_ST | P_CHRS | OMASK | NX_ST | OP1 | OP2 | OP3 |
| 1 | 1111 | 1 | ers | 1111 | 9 | he | | hers |
| 2 | 1100 | 1 | e?? | 0100 | – | he | – | – |
| 3 | 1111 | 1 | ish | 1111 | 4 | | his | |
| 4 | 1110 | 1 | is? | 0110 | – | | his | – |
| 5 | 1111 | 2 | rsh | 1111 | 4 | | hers | |
| 6 | 1110 | 2 | rs? | 0110 | – | | hers | – |
| 7 | 1111 | 3 | her | 1111 | 8 | | she he | |
| 8 | 1110 | 3 | he? | 0110 | – | | she he | – |
| 9 | 1111 | 3 | his | 1111 | 7 | | | his |
| 10 | 1111 | 4 | ers | 1111 | 9 | she he | | hers |
| 11 | 1100 | 4 | e?? | 0100 | – | she he | – | – |
| 12 | 1111 | 4 | ish | 1111 | 4 | | his | |
| 13 | 1110 | 4 | is? | 0110 | – | | his | – |
| 14 | 1111 | 5 | rsh | 1111 | 4 | | hers | |
| 15 | 1110 | 5 | rs? | 0110 | – | | hers | – |
| 16 | 1111 | 6 | she | 1111 | 5 | his | | she he |
| 17 | 1111 | 6 | shi | 1111 | 6 | his | | |
| 18 | 1100 | 6 | s?? | 0100 | – | his | – | – |
| 19 | 1111 | 7 | her | 1111 | 8 | | she he | |
| 20 | 1110 | 7 | he? | 0110 | – | | she he | – |

TABLE 2. 3-character transition rules 21 to 38

| rule no. | Pattern Data | | | Output Data | | | | |
|---|---|---|---|---|---|---|---|---|
| | TMASK | P_ST | P_CHRS | OMASK | NX_ST | OP1 | OP2 | OP3 |
| 21 | 1111 | 7 | his | 1111 | 7 | | | his |
| 22 | 1111 | 8 | she | 1111 | 5 | hers | | she he |
| 23 | 1111 | 8 | shi | 1111 | 6 | hers | | |
| 24 | 1100 | 8 | s?? | 0100 | – | hers | – | – |
| 25 | 1111 | 9 | her | 1111 | 8 | | she he | |
| 26 | 1110 | 9 | he? | 0110 | – | | she he | – |
| 27 | 1111 | 9 | his | 1111 | 7 | | | his |
| 28 | 0111 | 0 | her | 1111 | 8 | | he | |
| 29 | 0110 | 0 | he? | 0110 | – | | he | – |
| 30 | 0111 | 0 | his | 1111 | 7 | | | his |
| 31 | 0111 | 0 | she | 1111 | 5 | | | she he |
| 32 | 0111 | 0 | shi | 1111 | 6 | | | |
| 33 | 0011 | 0 | ?he | 1111 | 2 | | | he |
| 34 | 0011 | 0 | ?hi | 1111 | 6 | | | |
| 35 | 0011 | 0 | ?sh | 1111 | 4 | | | |
| 36 | 0001 | 0 | ??h | 1111 | 1 | | | |
| 37 | 0001 | 0 | ??s | 1111 | 3 | | | |
| 38 | 0000 | 0 | ??? | 1111 | 0 | | | |

The number of priority multiplexers is determined by the characters processed in parallel; for example, when there are $n$ characters compared in parallel, $n+1$ priority multiplexers are required. One priority multiplexer is used to select the output of next state, and the other $n$ priority multiplexers are used to select the $n$ matching outputs respectively.

Here we explain the operation of a priority multiplexer briefly. The control signals C1 to C$m$ are used to control the corresponding data inputs D1 to D$m$ respectively to decide which one of D1 to D$m$ should be output. The priority of C1 is the highest, while the priority of C$m$ is the lowest. When there are multiple control signals enabled, the data corresponded to the highest priority control signal will be selected to output. For example, when C2 and C3 are both '1', because the priority of C2 is higher than C3, the data D2 will be selected to output through Dout. In addition, control signals C1 to C$m$ are single-bit signals, while data inputs D1 to D$m$ are multiple-bit signals. The widths of inputs D1 to Dm are determined by the matching outputs and the number of states.

Referring to the block diagram shown in Figure 4, each matching unit includes data registers, a matching circuit, and a control circuit. The block enclosed by dash line is a control circuit. The control circuit consists of multiple AND gates. The control circuit is used to perform AND of the output EQ of the matching circuit with the corresponding bits of output mask OMASK to obtain the control flags NX_FLG and OF1 to OF$n$. Transition rule is stored in the data registers. The data registers are logically partitioned to two groups. The first group registers store the pattern data, and the second group registers store the output data. The matching circuit matches the input data with the pattern data. If the matching result is matched, then the output EQ is true; otherwise the output EQ is false.

The pattern data include the ternary mask TMASK, the pattern characters P_CHRS, and the current state P_ST. The output data include the output mask OMASK, the matching outputs OP1 to OP$n$, and the next state NX_ST. The details of pattern data and output data are explained together with the description of the transition rules.

4.2. **Transition rules.** Tables 1 and 2 show the examples of 3-character transition rules for the keywords {he, she, his, hers}. There are total 38 3-character transition rules and the rules are divided to two tables to fit the layout of the page. We list all the rules because we think it can help clarify our proposed approach.

In the tables, '$-$' represents a don't-care output, while in the real work it is determined by the corresponding bits in OMASK. The first field is rule number which is only for the sake of convenience in explanation and no register is required to store the rule number in the matching unit. The orders of the rules in the table represent the priorities of the rules. The priorities of rule 1 to 38 are from highest to lowest.

The fields of each rule are grouped to pattern data and output data. The pattern data include ternary mask TMASK, current state P_ST, and pattern characters P_CHRS. The output data include output mask OMASK, next state NX_ST, and matching outputs OP1 to OP$n$. The pattern data are compared with input characters and the current state, if the compare result is matched then the rule is activated and the output data with flag are generated. The matching outputs OP1 to OP3 are corresponded to the first, second and third characters of the pattern characters P_CHRS respectively.

Each bit of the ternary mask TMASK determines if the corresponding pattern data should be matched or not. For example, each bit from the most significant bit (MSB) to the least significant bit (LSB) of TMASK is corresponded to the current state P_ST, the first, second, and third pattern characters of P_CHRS respectively.

A pattern data character is compared only when the corresponding ternary mask bit is '1'; otherwise the pattern data character is don't-care. For example, the bit 3 of TMASK

in rules 28 to 38 are all '0', it means the current states P_ST of these rules are don't-care. Each bit of the output mask OMASK determines if the corresponding output data is valid or not. For example, each bit from the MSB to the LSB of OMASK in the rules is corresponded to the next state NX_ST, and the matching outputs OP1 to OP3 respectively. For example, bit 3, bit 1, and bit 0 of the OMASK in rule 2 are all '0', which means the NX_ST, OP2, and OP3 are not valid, that is, when rule 2 is activated, it does not affect both the next state and the matching outputs corresponded to the second and third input characters.

In Tables 1 and 2, the matching outputs are expressed as output strings. However, in a hardware implementation, the matching outputs can be expressed as the corresponding state numbers. For example, since the OP2 of rule 7 is "she he" which is corresponded to the output of state 5, the OP2 ("she he") of rule 7 can be represented by the state number 5. In this manner, the outputs can be stored in fixed width spaces instead of storing the variable length string data, and it is convenient for hardware design.

We now estimate the hardware cost for the proposed architecture. Suppose each input character is $b_c$ bits, P_ST and NX_ST are $b_s$ bits, and OP1 to OP$n$ are $b_p$ bits. The widths of TMASK and OMASK are $n$+1 bits. The total width of OP1 to OP$n$ is $n \times b_p$ bits and the width of P_CHRS is $n \times b_c$ bits. The required space $b_m$ of each matching unit for storing one $n$-character transition rule is calculated as following:

$$b_m = (n+1) + b_s + n \times b_c + (n+1) + b_c + n \times b_p = (b_c + b_p + 2) \times n + 2b_s + 2 \quad (1)$$

From (1), we can know the space for storing the rule in each matching unit is proportional to the number of characters ($n$) matched in parallel. If the used character set is usually an 8-bit one and the matching outputs are represented by state numbers, then $b_c = 8$ and $b_p = b_s$, and (1) is simplified as following:

$$b_m = (b_s + 10) \times n + 2b_s + 2 \quad (2)$$

From (2), we can know that the number of states determines the widths of the registers and further determines the space for rules and complexity of comparators.

4.3. **Generate multi-character transition rules.** After describing the proposed architecture and the transition rules, now we explain how to generate the transition rules used in the proposed architecture. For the sake of distinguishability, we call the new derived transition functions and transition rules used in the proposed architecture as reduced transition functions and reduced transition rules.

We use the examples shown in Figures 5 and 6 to explain how to generate the reduced multi-character transition rules. Because the failure links pointing to initial state are pruned in the deriving procedure, we add assistant transition functions for assisting to complete multi-character transition functions. For example, in Figure 5, $\delta(2, ?) = -$ and $\delta(-, ?) = -$ are assistant transition functions. In which, '?' represents don't-care and '$-$' represents non-existed null state or null output.

Figure 5 shows the examples of the reduced 1-character transition functions of states 1, 2, and 5. In Figure 5(a), we can see there are only two 1-character transition functions of state1 $\delta(1, \mathrm{e}) = 2$ and $\delta(1, \mathrm{i}) = 6$, compared with the example shown in Figure 2, and three 1-character transition functions $\delta(1, \mathrm{h}) = 1$, $\delta(1, \mathrm{s}) = 3$, and $\delta(1, \neg\{\mathrm{e}, \mathrm{h}, \mathrm{i}, \mathrm{s}\}) = 0$ derived from the failure functions linked to the initial state are pruned. In Figure 5(b), because state 2 has a matching output, we add an assistant transition function for assisting the derivation of multi-character transition functions. In Figure 5(c), since state 5 is a terminal state which has a matching output, we add an assistant transition function to state 5. Furthermore, since the failure function of state 5 is linked to state 2, the 1-character transition functions of state 2 should be included in the 1-character transition functions

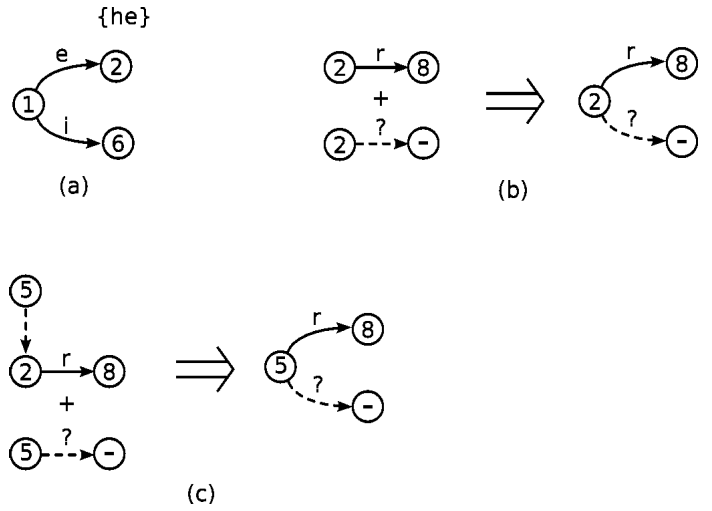


FIGURE 5. Examples of reduced 1-character transition functions of state 1, 2, and 5

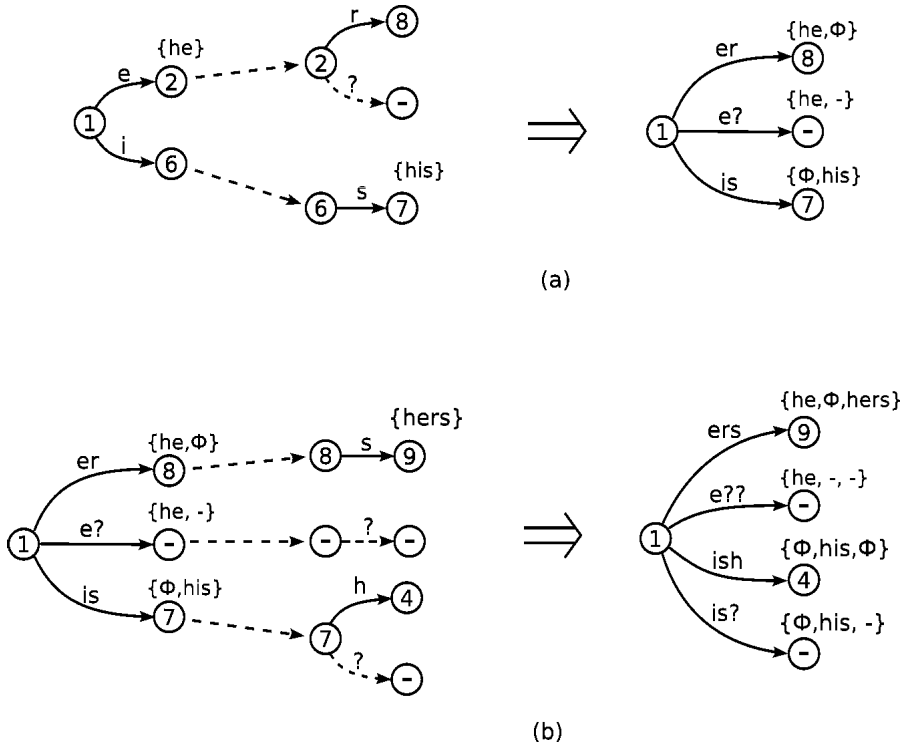


FIGURE 6. Examples of constructing reduced 3-character transition functions for state 1

of state 5. Following the above procedure, we can obtain all the reduced 1-character transition functions for every state in the AC-trie.

Figure 6 shows the example for constructing the reduced 3-character transition functions for state 1. The constructing procedure is similar to the example shown in Figure 2. In Figure 6(a), concatenating the reduced 1-character transition functions of state 1 with their successive reduced 1-character transition functions can obtain the reduced 2-character transition functions of state 1. In Figure 6(b), concatenating the derived reduced 2-character transition functions of state 1 with their successive reduced 1-character transition functions can obtain the reduced 3-character transition functions of state 1.

From the example described above, we can see that the transition functions of each state do not include the transition functions beginning from the initial state and the transition functions of each state are reduced.

Algorithm 1 is also used to generate the transition rules used in the proposed architecture. While we use the reduced 1-character transition functions instead as the input *NXSET* for Algorithm 1. In addition, the line 5 of the algorithm must be modified as the following:

$NSET \leftarrow$ all 1-character transitions of $Si$ in $NXSET$ excluding assistant transitions

The assistant transition functions are excluded, because the assistant transition functions need not to be expanded.

After generating the $n$-character transition rules, we must arrange the rules in an appropriate order. The priority of a rule with exact matching pattern should be higher than the priority of a rule with partial matching pattern. For example, in Tables 1 and 2, the rules of the same state (the field P_ST in the tables) are arranged together. The current states of rule 1 and rule 2 are the same, which is state 1, and the pattern characters of rule 1 and rule 2 are "ers" and "e??", respectively. Since the last two characters of the pattern characters of rule 2 are do not care, the priority of rule 2 is lower than the priority of rule 1 and rule 2 is arranged behind rule 1. In addition, since the current states of all the transition rules of initial state need not to be matched (i.e., don't care), all the transition rules of initial state are arranged behind the transition rules of the other non-initial states.

An intuitive approach is to arrange the rules according to the order of the binary values of the ternary mask TMASK. A rule with a larger TMASK value has a higher priority. For example, "1111" should be the highest. The orders of the rules with the same TMASK are not important because these rules will not be matched at the same time. The ternary mask of rule 38 is "0000" and its priority is the lowest, so that it is arranged at the last of the rule table. Rule 38 will be matched always to ensure the current state will stay at initial state when no other rules are matched.

4.4. **Example of multi-character transition matching.** In order to let readers understand the transition rules and the operation of the proposed architecture more easily, we use the example shown in Figure 7 to explain the operation of the proposed architecture of multi-character string-matching engine. In this example, the input string "ushehe" is divided into two 3-character strings "ush" and "ehe" and each 3-character string is processed in a matching cycle. In each matching cycle, we show the matched rules and how the results determined by the matched rules.

At the beginning, current state is reset to the initial state. In matching cycle 1, after accepting the first 3-character string "ush", rules 35, 36, and 38 are matched. Rule 35 which has the highest priority decides the next state and all the three outputs corresponding to the three input characters. In this matching cycle, the matching outputs are all empty string. Since the priorities of rules 36 and 38 are lower than the priority of rule 35, rules 36 and 38 do not affect the next state and any matching outputs.

In matching cycle 2, after accepting the second 3-character string "ehe", rules 11, 33, and 38 are matched. Rule 11 which has the highest priority decides the matching output corresponding to the first character to be "she he". Rule 33 which has the next high priority decides the next state to be state 2 and the matching outputs corresponding to the second and third characters to be an empty string and "he" respectively. Rule 38 which has the lowest priority does not affect the next state and any matching outputs.

Matching cycle 1 CUR_ST: 0 IN_CHRS: ush

| Rule no | TMASK | P_ST | P_CHRS | OMASK | NX_ST | OP1 | OP2 | OP3 |
|---|---|---|---|---|---|---|---|---|
| 35 | 0011 | ? | ?sh | 1111 | 4 | | | |
| 36 | 0001 | ? | ??h | 1111 | 1 | | | |
| 38 | 0000 | ? | ??? | 1111 | 0 | | | |
| Next state and matching outputs: | | | | | 4 | | | |

Matching cycle 2 CUR_ST: 4 IN_CHRS: ehe

| Rule no | TMASK | P_ST | P_CHRS | OMASK | NX_ST | OP1 | OP2 | OP3 |
|---|---|---|---|---|---|---|---|---|
| 11 | 1100 | 4 | e?? | 0100 | - | she he | - | - |
| 33 | 0011 | ? | ?he | 1111 | 2 | | | he |
| 38 | 0000 | ? | ??? | 1111 | 0 | | | |
| Next state and matching outputs: | | | | | 2 | she he | | he |

FIGURE 7. Example of matching operations

5. **Evaluation and Discussion.** In this section, we evaluate the proposed approach in two ways. First, we compare the numbers of generated transition rules for different number of keywords and different $n$-character FSMs. Secondly, we implement the proposed architecture on an ASIC device using FPGA synthesis tools to evaluate the utilization of hardware resource and estimate achievable throughput.

We used the keywords retrieved from the rules of SNORT as the sample for evaluation. We evaluated the numbers of rules and the required spaces for $n$-character FSMs of 200, 400, 600, 800, and 1000 keywords for the cases of $n = 1, 2, 3$, and 4.

Table 3 shows the numbers of transition rules for different number of keywords and different $n$-character FSMs. The numbers in the parentheses, for $n = 1$ is the ratio of the rules for $n = 1$ to total length of keywords, and for $n = 2$ to 4 are the ratios of the rules between $n$ and $n - 1$. For the sake of brevity, we use $len$ to represent the total length of keywords, and use $r1$ to $r4$ to represent the numbers of transition rules of $n$-character FSMs for $n = 1$ to 4. The curves shown in Figure 8 represent the relationships between the rules and $n$ for different sets of keywords. These curves are represented by $r_n = len \times k^n$, where the value of $k$ is dependent on the keyword set. The value of $k$ is 2.4, 3.6, 4.5, 5.2, or 5.7 for 200, 400, 600, 800, or 1000 keywords respectively. The value of $k$ is obtained by the arithmetic mean of the ratios $len/r_1$ and $r_{i-1}/r_i$ for $i = 1$ to 4. The curve for 200 keywords is not shown in Figure 8 since it is too close to the x-axis as compared with others. From the results shown in Table 3 and Figure 8, we can see that the number of rules is increased more rapidly when the keywords are increased. The reason for the increase in rules is that when keyword set size grows the failure functions linked to non-initial states are increased, and furthermore the next move functions ($\delta$) of the DFA-version AC-trie are increased more. Hence, in order to decrease the number of transition rules, the keywords should be partitioned to small subsets that each subset is matched by a separate hardware.

Table 4 shows the required spaces of each matching unit and total matching units for different numbers of keywords and different parallel characters $n$, the required total spaces are derived by multiplying the unit spaces with total rules showed in Table 3. In it, $b_s$ is the width of the registers which is represented in bits. The space of a matching unit

TABLE 3. Rules for different $n$-character FSMs

| keywords (total length) | $n=1$ ($r_1/len$) | $n=2$ ($r_2/r_1$) | $n=3$ ($r_3/r_2$) | $n=4$ ($r_4/r_3$) |
|---|---|---|---|---|
| 200 (2,321) | 4,775 (2.06) | 12,053 (2.52) | 31,030 (2.57) | 75,712 (2.44) |
| 400 (4,692) | 16,011 (3.41) | 61,653 (3.85) | 222,899 (3.62) | 777,428 (3.49) |
| 600 (8,032) | 35,064 (4.37) | 169,307 (4.83) | 757,859 (4.48) | 3,306,210 (4.36) |
| 800 (10,728) | 53,756 (5.01) | 308,207 (5.73) | 1,571,908 (5.10) | 7,806,812 (4.97) |
| 1000 (13,374) | 73,465 (5.49) | 458,623 (6.24) | 2,590,140 (5.65) | 14,249,657 (5.50) |

*len*: total lenth of keywords

$r_1$ to $r_4$: numbers of rules for $n = 1$ to 4

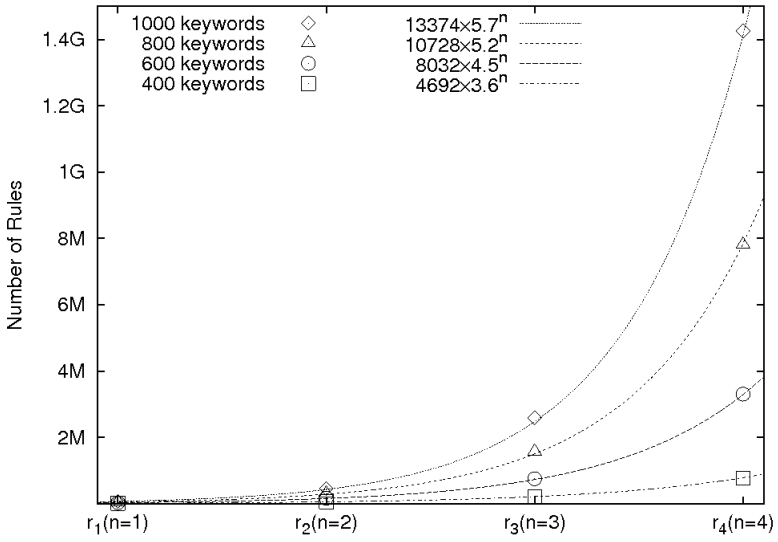


FIGURE 8. Relationship between growths in rules and $n$

is derived by (2) for 8-bit character set. The spaces are all represented in bits. In this table, it also lists the numbers of states of the generated AC-tries for different numbers of keywords, and the numbers in the parentheses are the width (bits) of the registers for storing the state numbers. Different numbers of keywords generate different numbers of states, thus determining the width of the registers for storing the state numbers and the total required space for a transition rule. In the other side, when $n$ is increased, the number of pattern characters and matching outputs and the widths of pattern mask and output mask of the transition rule are increased accordingly.

The relationship between the required space and $n$ can be obtained by simply multiplying the equation $r_n = len \times k^n$ with the space of a matching unit which is represented in (2). The resulting equation is $S_n = len \times k^n \times [(10 + b_s) \times n + 2b_s + 2]$. Consequently, the growth of the required space is about $n \times k^n$.

We implement the 4-character FSM for 512 and 1024 matching units in VHDL for evaluation. The implementations were compiled and synthesized using Altera Quartus II tools for an Altera's HardCopy IV ASIC HC4E35FF1517. According to the data

TABLE 4. Spaces for different $n$-character FSMs

| Keywords (total length) | Total states ($b_s$) | $n=1$ | | $n=2$ | | $n=3$ | | $n=4$ | |
|---|---|---|---|---|---|---|---|---|---|
| | | Unit Space | Total space | Unit Space | Total space | Unit Space | Total space | Unit Space | Total space |
| 200 (2,321) | 1,790 (11 bits) | 45 | 215K | 66 | 795K | 87 | 2.7M | 108 | 8.2M |
| 400 (4,692) | 3,597 (12 bits) | 48 | 769K | 70 | 4.3M | 92 | 21M | 114 | 89M |
| 600 (8,032) | 6,081 (13 bits) | 51 | 1.8M | 74 | 13M | 97 | 74M | 120 | 397M |
| 800 (10,728) | 8,029 (13 bits) | 51 | 2.7M | 74 | 23M | 97 | 152M | 120 | 937M |
| 1000 (13,374) | 9,916 (14 bits) | 54 | 4.0M | 78 | 36M | 102 | 264M | 126 | 1.8G |

*spaces are all represented in bits.

TABLE 5. Implementation of 4-character FSM on ASIC

| | 512 Matching units | 1024 Matching units |
|---|---|---|
| Used HCells | 304,666 | 608,436 |
| HCell Utilization | 3% | 6% |
| Total registers | 45,432 | 91,045 |
| Clock | 142 MHz | 95 MHz |
| Throughput | 4.5 Gbps | 3.0 Gbps |

TABLE 6. Comparison of different approaches

| Description | Clock (MHz) | Data Width | Throughput (Gbps) |
|---|---|---|---|
| Our proposed approach | 142 | 32 bits | 4.5 |
| D. Pao et al. [4] Pipelined implementation | 253 | 8 bits | 2.0 |
| Scarpazza et al. [7] Software program | 3200 | 64 bits | $1.6 \sim 40$ |
| Tripp [9] Parallel string matching engine | 149 | 32 bits | 4.7 |

sheet of the product, the ASIC has 9774,880 HCells, where an HCell is a logic array cell used in the HardCopy IV series devices. The results of compilation and synthesis of the implementations are shown in Table 5. In the table, the results include total used HCell, the utilization of HCell and registers, the maximum achievable operating clock, and the derived maximum achievable throughput. The achievable throughput is obtained by multiplying the data width which is 32 bits with the clock rate. In the implementations, the widths of the state registers and the matching output registers are 8 bits, and the widths of TMASK and OMASK are 5 bits. As of the maximum operating clock, the implementation of 512 matching units is higher than the implementation of 1,024 matching units. The reason for the degradation of the operating clock is that the priority multiplexers are implemented by chained multiplexers. The minimal critical path of a priority multiplexer is $\log_2 M$ chained multiplexers for $M$ matching units. Therefore, the delay of the priority multiplexer is longer for the more matching units. If the priority multiplexers are re-designed to reduce the time delays, the operating clock should be improved.

Table 6 shows the comparison of different approaches. The approach proposed by Scarpazza et al. [7] is a software implementation on the processor of IBM Cell/B.E., where the data in columns Clock and Data Width are the operating clock and the data width of the processor, respectively. In the other approaches, the data in columns Clock and Data Width are the clock rate and the bits of the data bus of the hardware implementations, respectively. The best throughput is 40 Gbps provided by Scarpazza et al. [7] with the constraint on less than 200 keywords. However, in the situation of more than 200 keywords the throughput of the approach of Scarpazza et al. is degraded to 1.6 Gbps. The pipelined architecture proposed by D. Pao et al. [4] can be operated at 253 MHz. However, since the approach of D. Pao et al. only can process one character every clock the throughput is 2.0 Gbps, which is less than our approach operating at 142 MHz. Our results are competitive to that of Tripp [9] that can process four characters every clock and the operating clock is 149 MHz and the throughput can be 4.7 Gbps. Our proposed approach has the advantage that it can generate the multi-character rules from the provided keywords systematically. In addition, the structure of our proposed architecture is simple and regular that can be easily implemented in different devices or can be designed as a stand alone integrated circuit (IC). Our proposed architecture is also scalable that allows us to design a string matching engine including the matching units as needed. The advantages of the hardware string matching accelerator are revealed from the comparison. The modern CPU is a sophisticated product that can run at very high speed and has wide data width. While a preliminarily developed hardware string matching accelerator running at much lower speed can achieve the throughput that is achieved by a soft program running at a very powerful CPU. Moreover, a hardware string matching accelerator that can inspect multiple characters in parallel can achieve multiplied throughput at the same clock rate.

6. **Conclusions and Future Work.** In this paper, we have described an intuitive and efficient algorithm to construct multi-character transition functions that represents a multi-character FSM from an AC-tire. The proposed algorithm iteratively concatenates the 1-character transition functions derived from the AC-trie to construct the multi-character transition functions. Then we proposed a hardware architecture to implement the derived multi-character FSM that utilizes the property of failure functions of the AC-trie to reduce the number of derived multi-character transition rules, and thereby the hardware cost can be reduced. In the evaluation, we first evaluate the required hardware cost for different $n$-character FSMs by using the keywords extracted from SNORT rules. The evaluations for required space show that when the keywords are increased the complexity of AC-trie is increased rapidly, and the required space for storing the $n$-character rules is also increased rapidly. The results of the evaluations for required space suggest that the keywords should be partitioned to small groups and each small group of keywords is matched by a small scale architecture. Then we implement the proposed architecture on an ASIC and the result of simulation shows that our architecture can achieve 4.5 Gbps for an implementation of 4-character FSM operated at 142 MHz clock.

The proposed architecture for multi-character transition string matching is simple and intuitive such that it can be easily implemented for any required number of characters inspected in parallel every matching cycle. In addition, the proposed architecture is flexible for applications. When the searching keywords are changed, we need only to generate new transition rules for the new keywords and update the registers of the matching units with the new transition rules.

There are some possible future works for this proposed architecture. Since the proposed architecture is regular and is similar to the architecture of a TCAM, the proposed architecture can be easily designed as a standalone integrated circuit (IC) referring to the

circuit of the TCAM. The standalone device based on the proposed architecture is appropriate for the applications that need to match multiple keywords at a time in high speed. In addition, we can redesign the proposed architecture to further reduce the number of the derived multi-character transition rules. For example, the proposed architecture can be redesigned in an NFA approach that can eliminate the transitions induced from the failure functions of the AC-trie. Furthermore, the operating clock of the proposed architecture can be pulled up by incorporating pipeline techniques to further increase the throughput of string matching.

**Acknowledgement.** The work is partially supported by a research project from National Science Council, Taiwan, under the grant number NSC-100-2218-E-002-011.

## REFERENCES

[1] A. V. Aho and M. J. Corasick, Efficient string matching: An aid to bibliographic search, *Commun. ACM*, vol.18, pp.333-340, 1975.

[2] *SNORT Network Intrusion Detection System*, http://www.snort.org.

[3] M. Alicherry, M. Muthuprasanna and V. Kumar, High speed pattern matching for network IDS/IPS, *IEEE ICNP*, pp.187-196, 2006.

[4] D. Pao, W. Lin and B. Liu, A memory-efficient pipelined implementation of the Aho-Corasick string-matching algorithm, *ACM Trans. on Archit. Code Optim.*, vol.7, pp.1-27, 2010.

[5] W. Lin and B. Liu, Pipelined parallel AC-based approach for multi-string matching, *IEEE ICPADS*, pp.665-672, 2008.

[6] N. Hua, H. Song and T. V. Lakshman, Variable-stride multi-pattern matching for scalable deep packet inspection, *IEEE INFOCOM*, pp.415-423, 2009.

[7] D. P. Scarpazza, O. Villa and F. Petrini, Exact multi-pattern string matching on the cell/b.e. processor, *ACM CF*, 2008.

[8] Y. Sugawara, M. Inaba and K. Hiraki, Over 10Gbps string matching mechanism for multi-stream packet scanning systems, *Field Programmable Logic and Application*, vol.3203, pp.484-493, 2004.

[9] G. Tripp, A parallel string matching engine for use in high speed network intrusion detection systems, *Journal in Computer Virology*, vol.2, pp.21-34, 2006.

[10] V. Rahmanzadeh and M. B. Ghaznavi-Ghoushchi, A multi-Gb/s parallel string matching engine for intrusion detection systems, *Advances in Computer Science and Engineering*, vol.6, pp.847-851, 2009.

[11] N. Yamagaki, R. Sidhu and S. Kamiya, High-speed regular expression matching engine using multi-character NFA, *IEEE FPL*, pp.131-136, 2008.