Microprocessors and Microsystems 39 (2015) 122–134

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A hybrid multiple-character transition finite-automaton for string matching engine

Chien-Chi Chen, Sheng-De Wang [*]

National Taiwan University, 1, Roosevelt Road Section 4, Taipei 106, Taiwan

ARTICLE INFO

Article history:
Available online 17 February 2015

Keywords:
String matching
Deterministic and nondeterministic finite automaton
Aho-Corasick algorithm
Network intrusion detection system

ABSTRACT

The throughput of a string-matching engine can be multiplied up by inspecting multiple characters in parallel. However, the space that is required to implement a matching engine that can process multiple characters in every cycle grows dramatically with the number of characters to be processed in parallel. This paper presents a hybrid finite automaton (FA) that has deterministic and nondeterministic finite automaton (NFA and DFA) parts and is based on the Aho-Corasick algorithm, for inspecting multiple characters in parallel while maintaining favorable space utilization. In the presented approach, the number of multi-character transitions increases almost linearly with respect to the number of characters to be inspected in parallel. This paper also proposes a multi-stage architecture for implementing the hybrid FA. Since this multi-stage architecture has deterministic stages, configurable features can be introduced into it for processing various keyword sets by simply updating the configuration. The experimental results of the implementation of the multi-stage architecture on FPGAs for 8-character transitions reveal a 4.3 Gbps throughput with a 67 MHz clock, and the results obtained when the configurable architecture with two-stage pipelines was implemented in ASICs reveal a 7.9 Gbps throughput with a 123 MHz clock.

1. Introduction

String matching is used in many applications, such as in network intrusion detection systems (NIDS), and its efficiency dominates the performance of such applications. String matching typically includes exact string matching and regular expression matching; exact string matching is more efficient but less powerful than regular expression matching in searching for keywords in a text. The string matching algorithm of Aho and Corasick processes multiple keywords simultaneously, and locates all instances of these keywords in an n-character text with time complexity $O(n)$ [1]. Some applications like NIDS that must inspect a data stream online firstly use exact string matching to filter out suspicious data, and then use regular expression string matching to verify the filtered out data.

A hardware string-matching engine that is based on the AC-algorithm can effectively accelerate string matching in network applications [2–5]. However, network bandwidth is increasing as communication and integrated circuit technologies advance, so the performance of string-matching engines must also be improved to keep up with network throughput. Greatly increasing throughput in string matching depends on developing a hardware string-matching engine that can inspect multiple characters in parallel.

A string matching engine that is based on the AC-algorithm can be implemented in two ways: the first uses a deterministic finite automaton (DFA) and the second uses a nondeterministic finite automaton (NFA). In the DFA approach, a generalized architecture can be developed for processing various sets of keyword. This architecture is deterministic. Accordingly, the generalized architecture based on the DFA approach can be designed as a standalone device, which can be utilized for various keyword sets. However, the hardware efficiency of the DFA approach is poor, and the required space typically increases exponentially with the number of characters that are being inspected in parallel. The NFA approach is more efficient than the DFA approach in terms of hardware utilization, and the required space increases in proportional to the number of characters that are being inspected in parallel. Nevertheless, the hardware architecture varies with keyword sets in the NFA approach; accordingly, the NFA approach can be implemented only in programmable devices, such as FPGAs.

This paper proposes a hybrid approach, based on the AC-algorithm, that combines the NFA and DFA methods to provide the advantages of both hardware efficiency and a deterministic architecture. The proposed approach transforms an AC-trie, which

* Corresponding author at: Department of Electrical Engineering, BL-523 National Taiwan University 1, Roosevelt Road Section 4, Taipei 106, Taiwan. Tel.: +886 2 33663579.

E-mail address: sdwang@ntu.edu.tw (S.-D. Wang).

http://dx.doi.org/10.1016/j.micpro.2015.01.003
0141-9331/© 2015 Elsevier B.V. All rights reserved.

is a prefix tree that is based on the AC-algorithm, into a hybrid finite automaton, called AC-FA, that has NFA and DFA parts. The proposed hybrid AC-FA is converted into a multi-character hybrid AC-FA by iteratively performing concatenation operations, for the purpose of processing multiple characters in parallel. We have previously developed the algorithm for deriving multi-character transitions [6,7]. The states in an AC-trie with the same depth are grouped in the same level, and a level nearer to the root state is lower; the NFA part comprises the lower states. Since most states of an AC-trie are at lower levels, which belong to the NFA part, the growth of the number of transitions of a k-character AC-FA is almost linear with respect to k when the number of level in the NFA part is high enough.

This paper then develops a multi-stage architecture for implementing the proposed hybrid AC-FA. The transitions are grouped into stages based on their levels. The number of stages of the proposed multi-stage architecture is determined by the number of levels in the NFA part and the number of characters to be inspected in parallel. Since the number of stages can be determined as required, the proposed multi-stage architecture is further made into a configurable architecture that can be configured to process various keyword sets. The proposed configurable architecture can be utilized as a stand-alone device.

The proposed architecture is evaluated on FPGA and ASIC devices. The results of the evaluation reveal that the throughput and the hardware cost increase approximately linearly with respect to the number of characters to be inspected in parallel. In the implementation of k-character multi-stage architecture, for $k = 8$, the throughput is approximately 6.1 times and the hardware cost is approximately 2.7 times those for $k = 1$. The 8-character hybrid AC-FA with 300 keywords can be implemented with a 66.6 MHz clock and the achieved throughput is approximately 4.3 Gbps. The 8-character configurable architecture, including 512 rule units, can be implemented with a 54.18 MHz clock and the obtained throughput is approximately 3.5 Gbps. The proposed configurable architecture is also improved using the pipeline method. The 8-character configurable architecture with two-stage pipelines can be implemented at a clock rate of 123.44 MHz and the obtained throughput is approximately 7.9 Gbps, and the performance is roughly doubled that of the original configurable architecture. The main contributions of this paper are summarized as follows.

– The proposed multi-character hybrid AC-FA approach has the feature that the number of transitions increases almost linearly with respect to the number of characters to be inspected in parallel.
– A configurable architecture is developed, based on the proposed multi-stage architecture; it can process various keyword sets by simply updating the configuration data.

The rest of this paper is organized as follows. Section 2 reviews work on string matching. Section 3 briefly describes the AC-algorithm, and then develops the hybrid AC-FA and its implementation. Section 4 describes the construction of multi-character transitions and proposes a multi-stage architecture to implement the constructed multi-character transitions. Section 5 proposes the configurable architecture of the multi-character matching engine. Section 6 evaluates and discusses the proposed approach. Finally, Section 7 draws conclusions.

2. Related work

Many hardware-based approaches that are based on the AC-algorithm have been developed for accelerating string matching and these fall into two broad categories. One category focuses on improving the efficiency of hardware utilization because the AC-algorithm is a memory-exhausting algorithm, while the other focuses on improving the throughput of string matching, by increasing the clock rate of the hardware or by inspecting multiple characters simultaneously.

Owing to the progress and flexibility of the programmable devices such as FPGAs, developers can design and evaluate variant architectures according to the features of the AC-algorithm. However, the resources of programmable devices are limited, so hardware efficiency is important. To improve the memory efficiency, Tuck et al. proposed a bitmap-compression and path-compression approach to implement the AC-algorithm that effectively reduces the required memory and improves the performance of hardware implementation [8]. Zha and Sahni improved the bitmap-compression and path-compression approach such that it utilized much less memory [9]. Alicherry et al. implemented the AC-algorithm by integrating a ternary content addressable memory (TCAM) and an SRAM that utilizes the ternary matching of TCAM to match characters in negation expressions, subsequently reducing the space that is required for storing the state transitions [3]. Pao et al. and Lin and Liu developed pipeline architectures to implement an AC-trie that contains only goto functions of the AC-trie to reduce the space that is caused by extending failure functions [10,11]. Nan Hua et al. proposed another approach that was based on a block-oriented scheme, rather than the typical byte-oriented processing of patterns, to minimize the memory-usage [4]. The approach of Dimopoulos et al. partitions an entire AC-trie into numerous smaller tries to increase memory efficiency [12]. Nakahara improved the hardware efficiency of the implementation of regular expression matching by using the AC-algorithm to process shared patterns [2]. Becchi et al. presented a hybrid FA that combines the advantages of DFA and NFA to improve regular expression matching [13].

To multiply the throughput of string matching for a fixed same operating rate, some attempts have been made to develop string matching architectures that can inspect multiple characters in parallel. Such an architecture must account for both the complexity of hardware and the alignment problem, which arises in the k-character matching processes. Sugawara et al. developed a string matching method called suffix-based traversing (SBT), which extends the AC-algorithm to process multiple input characters in parallel and to reduce the size of the lookup table [14]. Alicherry et al. proposed a k-compressed AC-DFA to realize a k-character matching engine [3]. Some works have utilized multiple FSMs to achieve parallelism and solve the alignment problem; in these, each FSM processes a pattern, beginning at a different position, and then the matching results of the FSMs are combined using a specific logic [5,15,16]. Yamagaki et al. utilized additional states and transitions to solve the alignment problem [17].

3. Aho-Corasick algorithm and hybrid finite automaton

This section firstly describes the AC-algorithm. Then the original AC-trie is converted to a hybrid AC-FA which comprises an NFA part and a DFA part. Thereafter, the operations of the original AC-trie and the proposed hybrid AC-FA are explained and compared with each other. Finally, the proposed hybrid AC-FA is extended to a multi-character hybrid FA.

3.1. Aho-Corasick algorithm

The AC-trie that is shown in Fig. 1 is constructed with the keyword set {enhappy, happy, happen, happygo}, which is used as an example to explain the proposed approach. In the figure, the

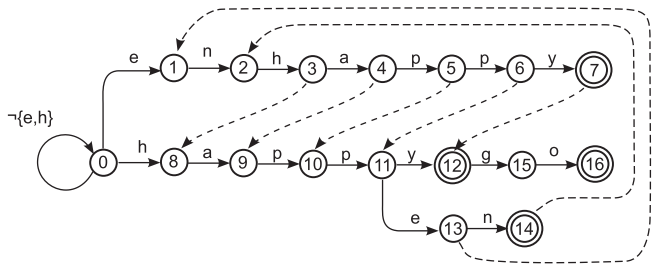


Fig. 1. Example of AC-trie.

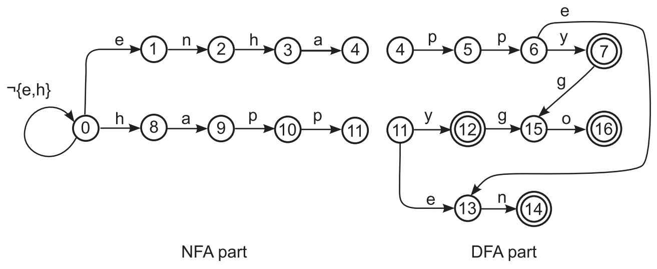


Fig. 2. Hybrid AC-FA.

circled numbers indicate states and the double-circled numbers indicate output states with non-empty matching outputs. The physical lines represent goto functions and the dashed lines represent failure functions. State 0 is the initial state. Every non-initial state has a failure function, and for clarity, the failure functions that are linked to the initial state are omitted.

In a matching cycle, the goto functions of the active state are checked first. If none of the goto functions is matched, then the active state transits to the state that is pointed to by the failure function of the current active state and the goto functions of the new active state are checked immediately. These steps are repeated until a goto function is matched. The fact that all non-initial states are eventually linked to the initial state through failure functions and the initial state has the goto functions for all characters in the character set ensures that a matched goto function can be found in every operating cycle. Later, the matching operation of the AC-trie will be discussed and compared with that of the proposed hybrid AC-FA with reference to an example.

In an AC-trie, each state represents a unique string. The linking of state S_1 to state S_2 by a failure function reflects the fact that the string that is represented by S_2 is the postfix of the string that is represented by S_1. In the case in which a failure function links state 4 to state 9, which represent 'enha' and 'ha' respectively, the string 'ha' is the postfix of 'enha'. Using failure functions, the AC-algorithm can locate all instances of keywords in a text string in a single-pass search.

However, the failure function may cause multiple transitions when an input character is being processed, making implementation inconvenient. Therefore, an AC-trie generally is converted into a deterministic finite automaton (DFA), called an AC-DFA, to simplify the implementation. After an AC-trie has been converted into an AC-DFA, only one transition is activated and one state has to be kept in every matching cycle; this deterministic property facilitates a simple implementation. Since the number of transitions increases rapidly after an AC-trie is converted into a DFA, its space utilization is inefficient. Alternatively, an AC-trie can be converted into a nondeterministic finite automaton (NFA), called an AC-NFA, by simply removing the failure functions and allowing multiple states to be activated simultaneously. Although the implementation of an AC-trie as an AC-NFA utilizes space efficiently, its circuit must normally be regenerated once the set of keywords changes.

3.2. Hybrid AC-FA

To develop a general architecture for implementing the AC-trie that minimizes the number of transitions, a hybrid finite automaton that is based on the AC-algorithm (hybrid AC-FA) that includes both DFA and NFA parts is developed. Fig. 2 shows a hybrid AC-FA that is obtained by converting the AC-trie in Fig. 1. In the NFA part, all of the failure functions are removed and only the goto functions are retained. In the DFA part, the failure functions are replaced with expanding goto functions. In a hybrid AC-FA, the number of NFA levels is defined as the total number of levels except that of the root state in the NFA part. For example, the hybrid AC-FA in Fig. 2 has three NFA levels. Notably, states 4 and 11 are in the DFA part.

Next, an algorithm for obtaining the transitions of a hybrid AC-FA is presented. For convenience of discussion, let a *k*-character transition, defined as $\delta_k(S_1, T) = S_2$, represent a transition from the current state S_1 to the next state S_2 on the *k*-character string *T*. Therefore, a transition of the hybrid AC-FA is denoted as $\delta_1(S_1, c) = S_2$, which represents a transition from state S_1 to state S_2 on character *c*.

Algorithm 1 is a generalized algorithm for deriving the transitions of a hybrid AC-FA from an AC-trie. In this algorithm, the input parameter N_{NFA} is the number of NFA levels. The parameter N_{NFA} is used to determine whether a state is in the NFA part or in the DFA part. The other two parameters, *G* and *F*, are the set of goto functions and the set of failure functions respectively. The output variable *NXSET* is the set of resulting 1-character transitions of the hybrid AC-FA.

Algorithm 1. Algorithm for deriving transitions of a hybrid AC-FA

```
Input  : $N_{NFA}$: the number of NFA levels
         $G$: goto function set
         $F$: failure function set
Output: $NXSET$: 1-character transition set
1  begin
2      $NXSET \leftarrow$ empty
3      for each goto $g(Si,c)=Sj$ in $G$ do
4      begin
5          $NXSET \leftarrow NXSET \cup \delta_1(Si,c)=Sj$
6      end
7      for each state $Si$ in DFA portion do
8      begin
9          $Sj \leftarrow f(Si)$
10         while $Sj$ in DFA portion do
11         begin
12             for each goto $g(Sj,c)=Sk$ of $Sj$ do
13             begin
14                 $NXSET \leftarrow NXSET \cup \delta_1(Si,c)=Sk$
15             end
16             $Sj \leftarrow f(Sj)$
17         end
18     end
19     return $NXSET$
20 end
```

In line 1, *NXSET* is initialized. Statements in the loop between lines 3 and 6 copy all of the goto functions to *NXSET*. Statements in the loop between lines 7 and 18 extend the transitions of the states in the DFA part following the failure functions. Whether a state is in the DFA part is checked by determining whether its level exceeds the parameter N_{NFA}.

In line 8, the state that is pointed to by the failure function of the processing state *Si* is assigned to *Sj*. If *Sj* is in the DFA part, then the process enters the loop between lines 10 and 17. In the loop between lines 12 and 15, all of the goto functions of *Sj* are converted to the transitions of *Si* and then added to *NXSET*. In line 15, *Sj* is updated with the state that is pointed to by its own failure function. The loop between lines 10 and 17 is repeated until *Sj* is not in the DFA part.

3.3. Matching operations

The matching operations of the original AC-trie and the hybrid AC-FA are explained and compared using the examples that are in Fig. 3. Fig. 3a and b show the matching processes of the original AC-trie and the hybrid AC-FA, respectively. The inspected text in all of these matching examples is ‘enhappenhappygo’.

First, the matching operation of the original AC-trie is discussed. In response to the characters ‘enhapp’, the transition traverses through the states 0, 1, 2, 3, 4, and 5 sequentially. In response to the seventh character ‘e’, none of the goto functions of state 6 matches this character, so the state transits to 11 according to the failure function of state 6 and the operation continues to match the goto functions of state 11. In response to the subsequent character ‘n’, the state transits to 14, which is an output state and a matching output, ‘happen’, is obtained. State 14 is a terminal state, so based on its failure function, the state transits to 2 when the next character is input and the matching process proceeds from state 2. Similarly, in response to the remaining characters, the transition passes through states 3, 4, 5, 6, 7, 12, 15, and 16 sequentially, where the transition from 7 to 12 is a transition that is determined by the failure function of state 7. The remaining matching operation yields matching outputs at states 7 and 16, which are ‘enhappy happy’ and ‘happygo’, respectively.

Next, the matching operation of the hybrid AC-FA is explained with reference to Fig. 3b. For clarity, the explanation focuses on the operations that differ from those of the original AC-trie. In this example, the states in the DFA part are underlined. As can be seen, multiple states can be activated simultaneously in the NFA part, while only the deepest state remains to be activated in the DFA part. In response to the characters ‘enhappen’, a transition sequence from 0 to 6 followed by 13 and 14, is generated. After the third character ‘h’, another transition sequence begins, and continues until state 11 is reached. Since 11 and 6 are the states in the DFA part, only the deeper state 6 can be active. These two transition sequences are merged into one and the matching operation proceeds to state 6. This transition sequence ends at state 14 with a matching output, which is ‘happen’. When the matching operation proceeds to the second ‘e’ character, the third transition sequence begins. As in the description above, the fourth transition sequence begins on the second ‘h’ character and merges with the third transition sequence at state 11. The third transition sequence has matching outputs at states 7 and 16.

Comparing Figs. 3a and b indicates that, in every matching cycle, when a non-initial state is activated, all of the states in the NFA part that are linked from the active state through the failure functions are activated simultaneously. Hence, failure functions are not required in the NFA part when multiple states can be activated simultaneously.

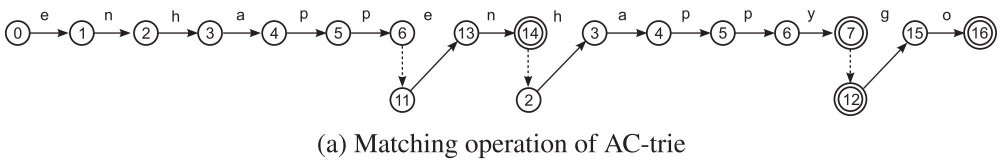
3.4. Implementation of hybrid AC-FA

The AC-NFA has a property that is the basis for developing the multi-stage architecture of the proposed hybrid AC-FA. An NFA that is derived from a given AC-trie has no more than one active state among states of equal depth. For convenience, states with the same depth are grouped in the same level. The states with the same depth represent a set of strings of the same length, so no more than one state in a level can be activated at one time. Consequently, only one register is required for saving the active state per level in the NFA part. As a result, the number of registers that are required to keep the active states in matching operations can be determined by the number of levels in the NFA part.

Fig. 4 shows the multi-stage architecture of the hybrid AC-FA. Stages 1 through 4, which are associated with the NFA part, include the transitions of levels 0 through 3, respectively. Stage 5, the terminal stage, is associated with the DFA part and includes all of the transitions of the DFA part. Since no more than one state can be activated in each stage, priority multiplexer PMUX0 determines the next state for the terminal stage from states NX[4] and NX[5], which are output from stages 4 and 5, respectively. The next state NX[i] that is generated by the *i*-th stage represents the matching result of that stage. The priority multiplexer PMUX1 obtains the final matching output from the matching results NX[1] through NX[5].

A priority multiplexer accepts multiple inputs and selects a valid input with the highest priority as the output; if none of the inputs is valid, then a default value, typically zero, is output. In the priority multiplexer, an input in a higher position has a higher priority so, for example, input D[1] has the highest priority.

Fig. 5 shows the block diagram of a stage unit. A stage unit includes multiple rule units, each of which is responsible for


(a) Matching operation of AC-trie

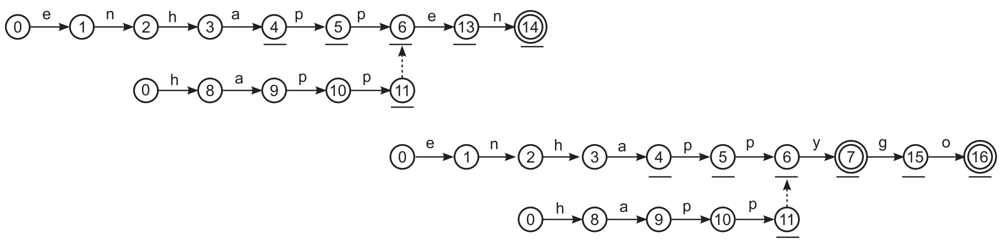

(b) Matching operation of hybrid AC-FA

Fig. 3. Matching examples of AC-trie and hybrid AC-FA.

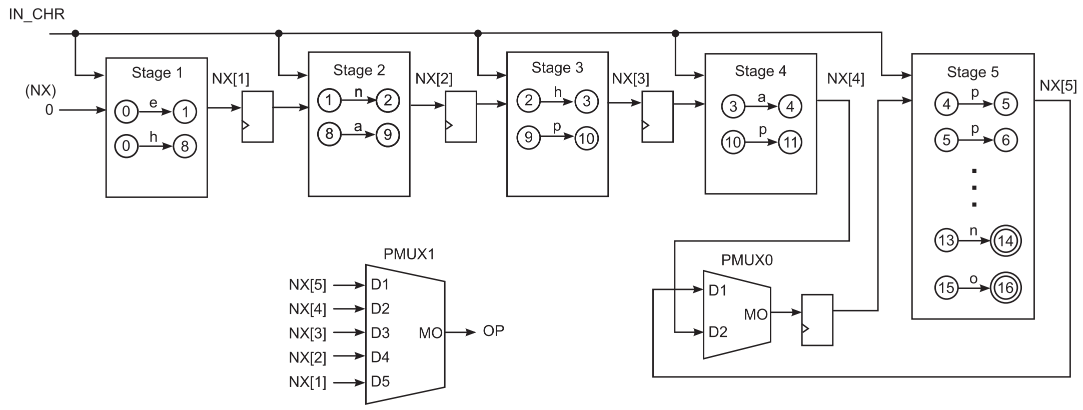


Fig. 4. Implementation of hybrid AC-FA.

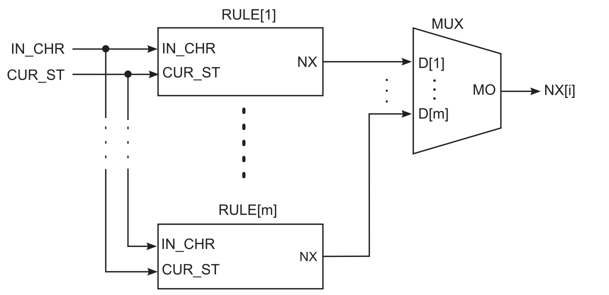


Fig. 5. Block diagram of a stage unit.

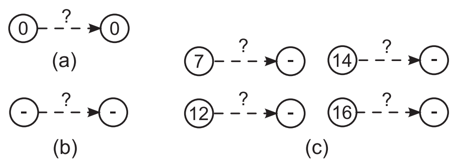


Fig. 6. Assistant transitions.

matching one transition. Therefore, the number of rule units in a stage must be greater than or equal to the number of transitions. A rule unit contains information about its corresponding transition and matches that information with the input current state CUR_ST and character IN_CHAR in the matching operation. A rule unit is triggered when its pattern is matched with the inputs and it then outputs the next state NX[i], as determined by its information.

4. Multi-character hybrid AC-FA

This section firstly explains how to derive multi-character transitions from a hybrid AC-FA with reference to examples. Next, a multi-character hybrid AC-FA is constructed using the derived multi-character transitions. Then, the implementation of the derived multi-character hybrid AC-FA is elucidated. Finally, an illustrative example is provided to elucidate the matching operation of the proposed multi-character hybrid AC-FA.

The alignment problem, which must be addressed in the derivation of multi-character transitions, arises in two cases: in one, a pattern does not begin at the first character of the inspected characters, whereas in the other, a pattern does not end at the final character of the inspected characters.

Considering the keyword ‘happen’, and the inspection of three characters in parallel; two 3-character transitions $\delta_3(0, \text{hap}) = 10$ and $\delta_3(10, \text{pen}) = 14$ can be obtained intuitively. When the text ‘enhappens’ is taken as the input, and split into three chunks, ‘enh’, ‘app’, and ‘ens’ for inspection, the keyword ‘happen’ cannot be found using the above two obtained transitions in this text because it begins at the third character of the first chunk and ends at the second character of the third chunk.

4.1. Derivation of multi-character transitions

In the proposed approach, multi-character transitions are derived from an AC-trie by concatenating multiple successive transitions. Three types of assistant transition are defined to solve the alignment problem that arises in the derivation of multi-character transitions. Fig. 6 illustrates examples of such assistant transitions.

The assistant transitions are indicated by dashed lines, to distinguish them from the normal transitions. In Fig. 6, the circled symbol ‘–’ represents a pseudo state and the symbol ‘?’ represents an arbitrary character. The pseudo state is a virtually defined state that does not actually exist in the AC-trie. The first type of assistant transition $\delta_1(0, ?) = 0$ (Fig. 6a) solves the alignment problem when the beginning of a pattern is not at the first of the inspected characters. The second type of assistant transition (i.e., $\delta_1(S_{op}, ?) = –$) preserves the matching output when a pattern does not end at the final character of the inspected characters. The transitions that begin from states 7, 12, 14, and 16 and end at pseudo states are assistant transitions of the second type (Fig. 6c). An assistant transition of the second type solves the alignment problem when a pattern does not end at the final inspected character. An assistant transition of the third type (i.e., $\delta_1(–, ?) = –$ (Fig. 6b)) can follow an assistant transition of the second type to complete a multi-character transition.

Fig. 7 shows examples of the derivation of 3-character transitions using concatenation operations. In the figure, ‘+’ indicates a concatenation operation. These examples also clarify the use of assistant transitions in the derivation.

First, with reference to Fig. 7a, concatenating two assistant transitions $\delta_1(0, ?) = 0$ yields a 2-character transition $\delta_2(0, ??) = 0$; then, concatenating the deriving transition with $\delta_1(0, \text{e}) = 1$ and $\delta_1(0, \text{h}) = 8$, respectively, yields two 3-character transitions $\delta_3(0, \text{??e}) = 1$ and $\delta_3(0, \text{??h}) = 8$. Although a 3-character transition $\delta_3(0, \text{???}) = 0$ can be obtained by concatenating the transition $\delta_2(0, ??) = 0$ with another $\delta_1(0, ?) = 0$, it is discarded because it is useless. The transitions $\delta_3(0, \text{??e}) = 1$ and $\delta_3(0, \text{??h}) = 8$ imply that state 0 is retained for the first two characters and then transits to state 1 or 8 on the third character ‘e’ or ‘h’. These derived transitions are thus utilized to solve the alignment problem that arises from the situation in which the first character of a keyword appears is the third inspected character.

In the example in Fig. 7b, state 6 has two goto functions, $\delta_1(6, \text{y}) = 7$ and $\delta_1(6, \text{e}) = 13$. Concatenating $\delta_1(6, \text{y}) = 7$ with two assistant transitions $\delta_1(-,?) = -$ yields a 3-character transition $\delta_3(6, y??) = -$, and concatenating $\delta_1(6, y) = 7$ with $\delta_1(7, g) = 15$ and $\delta_1(15, o) = 16$ yields another 3-character transition $\delta_3(6, ygo) = 16$. Concat enating $\delta_1(6, e) = 13$ with $\delta_1(13, n) = 14$ and $\delta_1(-,?) = -$ yields a 3-character transition $\delta_3(6, en?) = -$. States 7 and 14 are output states; state 14 is also a terminal state. Both states 7 and 14 are concatenated with assistant transitions to form complete 3-character transitions to preserve the matching outputs. Since the trailing two characters of the pattern in transition $\delta_3(6, y??) = -$ are wildcard characters, the transition is always triggered along with the transition $\delta_3(6, ygo) = 16$; such a conflict is resolved by using a priority multiplexer.

The described concatenation operation enables the k-character transitions with any required k value to be derived. If all of the levels of an AC-trie are in the NFA part, such that the hybrid AC-FA becomes an AC-NFA, then the number of derived k-character transitions is proportional to k. For a given AC-NFA with r_1 1-character transitions and n_{op} output states, the number of k-character transitions is $r_k = r_1 + (k-1) * n_{op}$. In the above example, the 1-character AC-NFA has 16 transitions and four output states, so the derived 3-character AC-NFA has $16 + (3-1) * 4 = 24$ transitions.

Algorithm 2 is a generalized algorithm for deriving multi-character transitions from an AC-trie. In this algorithm, the input parameter k is the number of characters to be inspected in parallel. The input parameter *NXSET* is the set of original 1-character transitions, derived according to Algorithm 1. The output variable *TRSET* is the set of resulting k-character transitions. By using multiple-level iterations, this algorithm derives the k-character transitions for every state in the original AC-trie.

Algorithm 2. Algorithm for deriving k-character transitions

```
Input  : k: number of characters
         NXSET: 1-character transition set
Output: TRSET: k-character transition set
1  begin
2      TRSET ← empty
3      for each state Si do
4      begin
5          NSET ← all transitions of Si in NXSET
6          repeat k − 1 do
7          begin
8              TMPS ← empty
9              for each transition NXi in NSET do
10             begin
11                 Sj ← next state of NXi
12                 for each transition NXj of Sj do
13                 begin
14                     NTR ← NXi + NXj
15                     TMPS ← TMPS ∪ NTR
16                 end
17             end
18             NSET ← TMPS
19         end
20         TRSET ← TRSET ∪ NSET
21     end
22     remove unused transitions from TRSET
23     return TRSET
24 end
```

In line 1, *TRSET* is initialized. The loop between line 3 and line 21 derives all of the k-character transitions of every state *Si*. In line 5, the 1-character transitions of state *Si* are duplicated to variable *NSET*. The loop between line 7 and line 19 is performed $k-1$ times, in which the 1-character transitions of *Si* are iteratively

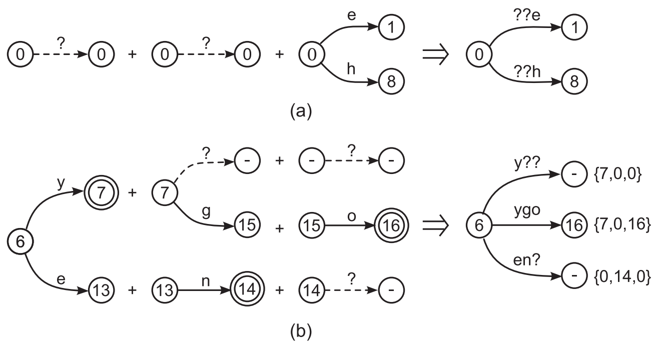


Fig. 7. Examples of concatenating operations.

concatenated with their successive 1-character transitions to derive the k-character transitions of *Si*. After executing the loop, *NSET* contains all of the k-character transitions of *Si*. In line 20, *NSET* is added to *TRSET*. The algorithm then returns to line 5 to process the next state immediately. The algorithm is terminated when all of the states have been processed. Finally, *TRSET* contains the derived k-character transitions.

Now, consider the loop between lines 7 and 19. In line 8, *TMPS* is initialized. In the loop between lines 10 and 17, every transition *NXi* in *NSET* is expanded. In line 11, the next state of *NXi* is assigned to *Sj*. In the loop between lines 13 and 16, *NXi* is concatenated with every transition *NXj*, beginning from *Sj*, to generate new transitions. In line 14, *NXi* is concatenated with *NXj* to obtain a new transition *NTR*, and then in line 15, *NTR* is added to *TMPS*. The number of the pattern characters of *NTR* is one more than the number of pattern characters of *NXi*.

Since the intermediate state is concealed after two transitions have been concatenated, the matching outputs must be reserved in the concatenation operation in line 14. Moreover, some transitions consisting of all assistant transitions that may be obtained in the concatenation process are not useful and are subsequently removed in line 22.

Fig. 8 depicts the complete 3-character hybrid AC-FA, constructed using the procedure that is described above. The transitions in the NFA part are depicted as three disjoint NFAs, each of which deals with a particular case of alignment. The nodes of the 3-character hybrid AC-FA are arranged according to their levels in the original AC-trie to clarify their relationship.

4.2. Implementation of multi-character hybrid AC-FA

This section develops a multi-stage string-matching architecture for implementing the derived multi-character hybrid AC-FA. Fig. 9 shows a block diagram of the proposed architecture for implementing a 3-character hybrid AC-FA that is derived from the example in Fig. 2. The inspected text is split into chunks of three characters, which are fed one at a time, through the input IN_CHRS into the string matching engine.

The number of stages, L, is determined by the number of NFA levels N_{NFA} and the number of characters to be inspected in parallel $k : L = N_{NFA} + k + 1$. In the presented example, the NFA-level is $N_{NFA} = 3$ and the number of characters to be inspected in parallel is $k = 3$, so the number of stages $L = 3 + 3 + 1 = 7$.

Stages 1 through 6 are arranged in three chains, each of which deals with an alignment case. Stage 1, the first stage in the first chain, deals with the case in which the first character of a pattern is the third character of the input chunk, so the preceding two characters of the transitions are both wildcard characters.

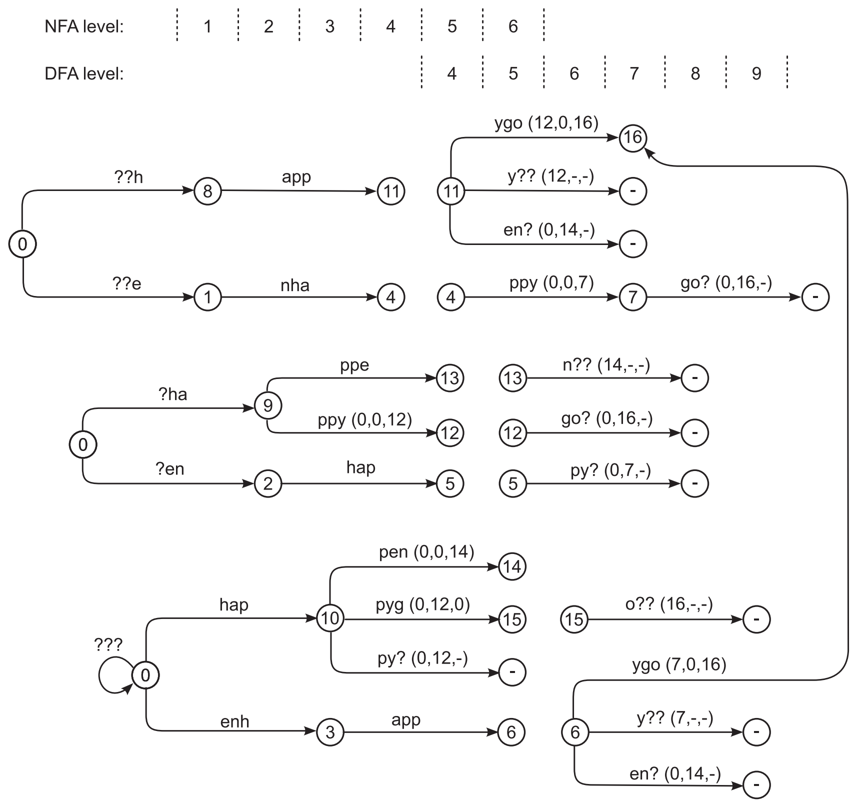


Fig. 8. The derived 3-character hybrid AC-FA.

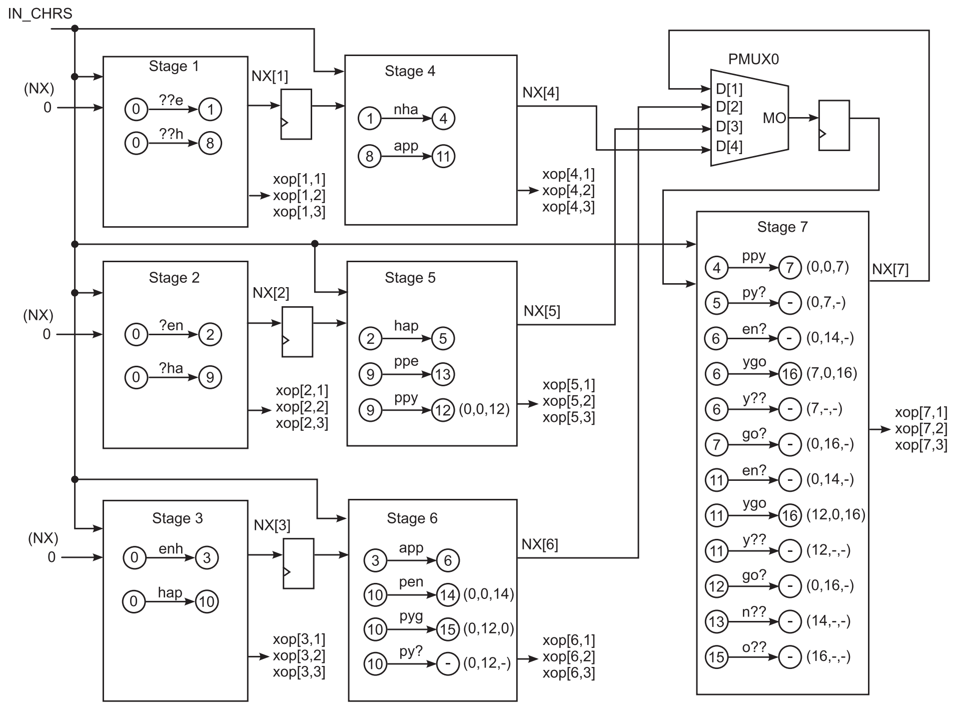

IN_CHRS
(NX) 0
Stage 1
NX[1]
xop[1,1] xop[1,2] xop[1,3]
Stage 2
NX[2]
xop[2,1] xop[2,2] xop[2,3]
Stage 3
NX[3]
xop[3,1] xop[3,2] xop[3,3]
Stage 4
NX[4]
xop[4,1] xop[4,2] xop[4,3]
Stage 5
NX[5]
xop[5,1] xop[5,2] xop[5,3]
Stage 6
NX[6]
xop[6,1] xop[6,2] xop[6,3]
PMUX0
D[1] D[2] D[3] D[4]
MO
Stage 7
NX[7]
xop[7,1] xop[7,2] xop[7,3]

Fig. 9. Block diagram of the multi-stage string matching architecture.

The transitions of the DFA part are all in stage 7, which is the terminal stage. All of the next states that are determined by the final stages of the three chains, which are stages 4–6, traverse into stage 7. Since no more than one state can be activated at a time in each stage, the priority multiplexer PMUX0 determines the next state for stage 7 from the next states, NX[4] through NX[7], which are output from stages 4 through 7. The transitions in the later stage correspond to the states at the deeper level of the AC-trie, so the next state that is determined by a later stage has a higher priority, so for example, the next state NX[7] that is determined by stage 7, which is the terminal stage, has the highest priority.

Since multiple transitions may be triggered simultaneously, the transitions in each stage are arranged by priority such that a transition with a higher priority has a higher position in the block diagram. For example, transition $\delta_3(6, \text{y??}) = -$ is always triggered when the transition $\delta_3(6, \text{ygo}) = 16$ is triggered, so the former has a lower priority. However, if two transitions are never triggered simultaneously, then their priorities are unimportant.

The final matching outputs are determined from the results of stages using priority multiplexers. Fig. 10 shows the priority multiplexer that is used to determine the final matching output OP[*i*] that corresponds to the *i*-th inspecting character. In the presented example, three priority multiplexers are required to determine three matching outputs – one for each.

Fig. 11 shows a block diagram of a stage unit. A stage unit includes multiple rule units, each of which is responsible for matching one transition. Therefore, the number of rule units must be at least the number of transitions in each stage. A rule unit contains the information about its corresponding transition and matches this information with the input current state CUR_ST and characters IN_CHRS when the matching operation is performed. A rule unit is triggered when its pattern is matched with

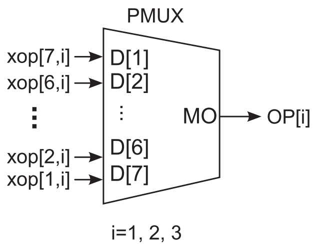


Fig. 10. Matching output circuit.

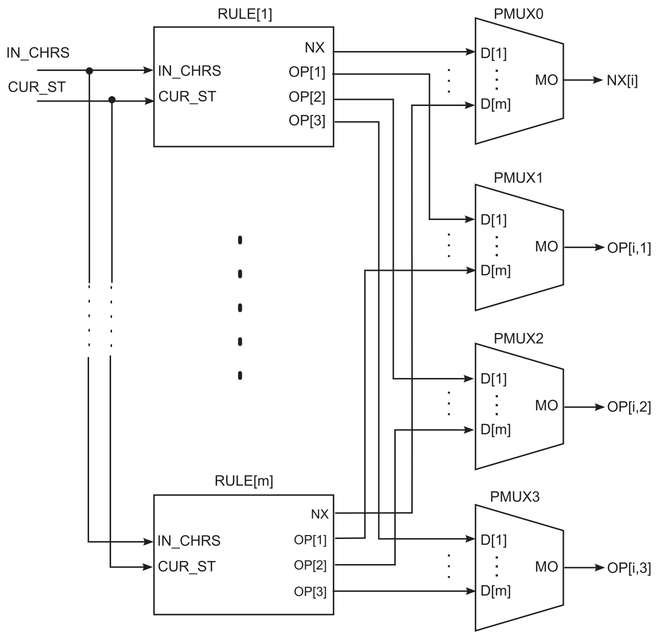


Fig. 11. Block diagram of a stage unit.

the inputs and it then outputs the next state NX and the matching outputs OP[1] through OP[3], as determined by its information.

4.3. Matching example

To demonstrate the effectiveness of the proposed approach, Fig. 12 shows an example of the matching process. The input text in this example is ‘enhappenhappygo’, which is split into chunks of three characters, which are processed in order in five matching cycles. In the first cycle, in response to the input ‘enh’, the transitions $\delta_3(0, \text{??h}) = 8$ in stage 1 and $\delta_3(0, \text{enh}) = 3$ in stage 3 are triggered. Neither transition has a matching output so no matching output exists in this cycle. The next states that are determined in stages 1 and 3 are sent to stages 4 and 6, respectively.

In the second cycle, the input ‘app’ and the next states that were determined in the preceding cycle trigger the transitions $\delta_3(8, \text{app}) = 1$ in stage 4 and $\delta_3(3, \text{app}) = 6$ in stage 6, which determine next states 11 and 6, respectively. Both states 11 and 6 are in the DFA part and the latter is at the greater depth, so state 6 is preserved and sent to stage 7. The matching outputs are all empty in this cycle.

In the third cycle, the input ‘enh’ and the next states that were determined in the preceding cycle trigger the transitions $\delta_3(6, \text{en?}) = -$ in stage 7, $\delta_3(0, \text{??h}) = 8$ in stage 1, and $\delta_3(0, \text{enh}) = 3$ in stage 3. The transition $\delta_3(6, \text{en?}) = -$ determines that the matching output that corresponds to the second input character is 14 or ‘happen’.

In the fourth matching cycle, the input characters ‘app’ and the next states that were determined in the preceding matching cycle triggered transitions $\delta_3(8, \text{app}) = 1$ in stage 4 and $\delta_3(3, \text{app}) = 6$ in stage 6. Priority multiplexer PMUX0 selects the next state that was determined by stage 6, which is 6, and sends it to stage 7. The matching outputs are all empty strings in this cycle.

In the fifth matching cycle, two transitions in stage 7, which are $\delta_3(6, \text{ygo}) = 16$ and $\delta_3(6, \text{y??}) = -$, are triggered in response to the input characters ‘ygo’ and the next states that were determined in the preceding matching cycle. The transition $\delta_3(6, \text{ygo}) = 16$ has a higher priority so it causes the next state to be 16 and the matching outputs that correspond to the three input characters are 7, 0, and 16 in this matching cycle. The first and third matching outputs, 7 and 16, correspond to the keywords ‘enhappy happy’ and ‘happygo’ respectively.

5. Configurable string-matching architecture

The numbers of rules vary among the stages in the proposed multi-stage string matching architecture. Additionally, the rules in each stage vary with the keyword set. Hence, the proposed multi-stage architecture is difficult to design and manufacture as a stand-alone device for various applications. However, the number of stages can be predetermined by the specific requirements that must be met – which are the number of levels in the NFA part and the number of characters to be inspected in parallel. Therefore, the proposed multi-stage architecture can be devised as a configurable architecture to provide flexibility in applications. In this configurable architecture, all rule units are grouped together and each can be dynamically allocated to a different stage by setting its own rule data. Consequently, this configurable architecture can process various keyword sets by simply updating the rule data.

Fig. 13 shows the main block diagram of the proposed configurable architecture, which includes rule, state, and output circuits. All rule units are grouped together in the rule circuit, in which each rule unit processes a transition. A rule unit can be allocated to a particular stage based on its own rule data. Each rule unit has registers that save rule data, which can be updated using the input SET_IN. The state circuit determines the next states of the stages,

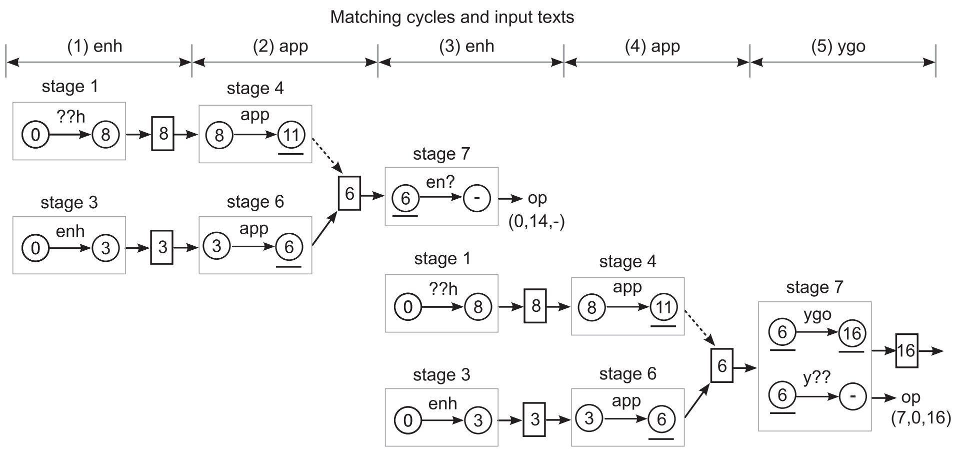


Fig. 12. Example of a matching process.

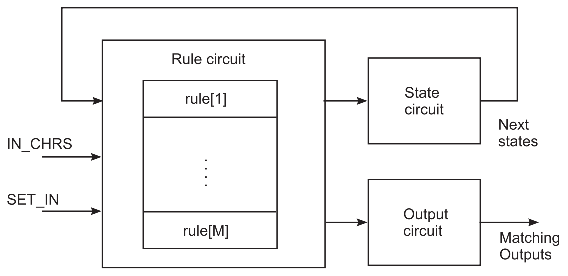


Fig. 13. Main block diagram of the configurable architecture.

Table 1
Example of transition rules.

STG	Pattern data			Output data				
	TMSK	P_ST	P_CHRS	OMSK	NX_ST	OP1	OP2	OP3
0	0001	0	??e	1001	1	–	–	
0	0001	0	??h	1001	8	–	–	
≈								
4	1111	1	nha	1111	4			
4	1111	8	app	1111	11			
≈								
7	1111	11	ygo	1111	16	12		16
7	1100	11	y??	0100	–	12	–	–
≈								

and the next states thus determined are looped back to the rule circuit as current states in the following matching cycle. The output circuit determines the final matching outputs that correspond to the inspected characters in the matching results output by all rule units.

5.1. Rule unit

Fig. 14 shows the block diagram of a rule unit. The rule data for each rule unit include stage information that is used to determine to which stage the rule unit belongs. Based on the stage information, the multiplexer selects the corresponding input state and the demultiplexer sends the resulting next state to the corresponding output.

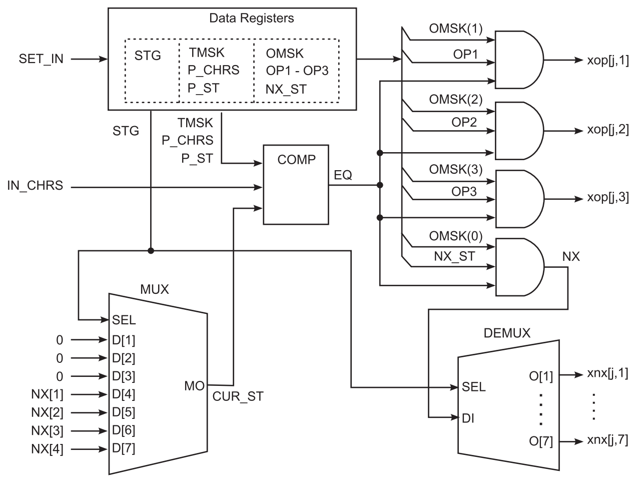


Fig. 14. Rule unit of the configurable architecture.

Each rule unit has registers to store the rule data associated with a transition. Table 1 presents some examples of transition rules. The rule data include a stage number, pattern data, and output data. The multiplexer MUX selects one of the inputs, 0 and NX[1] through NX[4], as the current state CUR_ST, based on the stage number STG. The comparator COMP compares the two inputs, IN_CHRS and CUR_ST, with the pattern data, and then outputs the matching result as signal EQ. The signal EQ is true when the inputs are matched with the pattern data; otherwise it is false. The four AND gates determine the outputs of the rule unit based on the signal EQ and the rule data OMSK, NX_ST, and OP1 through OP3.

The transition rules are now explained in detail with reference to the examples in Table 1. The first field, STG, contains the stage number. The remaining fields can be roughly separated into fields for pattern data and fields for output data. The pattern data include the ternary mask TMSK, the current state P_ST, and the pattern characters P_CHRS. The output data include the output mask OMSK, the next state NX_ST, and the matching outputs OP1 through OP3. The pattern data of a rule are compared with the input characters and the current state; if the result of the comparison is a match, then the rule is activated and the output data are sent out. The matching outputs OP1 through OP3 correspond to the first through third characters of P_CHRS, respectively.

Each bit of the ternary mask TMSK determines whether the corresponding pattern should be compared with the corresponding input or not. For instance, the bits of TMSK, from the most significant bit (MSB) to the least significant bit (LSB), correspond to the current state P_ST and the first to third characters of P_CHRS, respectively. A pattern is compared with its corresponding input when the corresponding ternary mask bit is '1'; otherwise the pattern is do not-care. For instance, in the two rules of stage 1, bits 3 through 1 of TMSK are all '0', and, meaning that P_ST and the first and second characters of P_CHRS are do not-care. Each bit of OMSK determines whether the corresponding output value is valid or not. Specifically, the bits of OMSK, from the MSB to the LSB, correspond to the next state NX_ST, and the matching outputs OP1 through OP3, respectively. For example, in the final rule, bit 3, bit 1, and bit 0 of OMSK are all '0', so NX_ST, OP2, and OP3 are not valid; accordingly, when this rule is activated, it affects neither the next state nor the matching outputs that correspond to the second and third input characters.

The examples of rules in Table 1 are $\delta_3(0, ??e) = 1$ and $\delta_3(0, ??h) = 8$, which are two rules in stage 1; $\delta_3(1, nha) = 4$ and $\delta_3(8, app) = 11$, which are two rules in stage 4, and $\delta_3(11, ygo) = 16$ and $\delta_3(11, y??) = –$, which are two rules in stage 7. Stage 7 is the terminal stage and is in the DFA part. In the table, '–' represents a do not-care output, while in practical implementation, the matching output is determined by the corresponding bit of OMSK. In each stage, the rules are arranged in order of priority. If multiple rules are triggered at the same time then a higher one has a higher priority. For instance, the two example rules in stage 7, $\delta_3(11, ygo) = 16$ and $\delta_3(11, y??) = –$, may be triggered simultaneously and the former has a higher priority in determining the next state and the matching outputs.

In the rule table, when a matching output is valid, it is expressed as a state number if the state is a non-initial state; otherwise, it is blank. For example, in the first rule of stage 7, the output OP1 is 12, which represents the string 'happy', and the output OP3 is 16, which represents the string 'happygo'.

5.2. State and output circuits

The circuits for determining the next states NX[1] through NX[3] are identical; each is implemented by a priority multiplexer, as shown in Fig. 15. The circuit for determining the next state NX[4] is composed of two levels of priority multiplexers, as shown in Fig. 16. The priority multiplexers PMUX4 through PMUX7 determine the next states from the matching results that correspond to stages 4 through 7 and then PMUX8 determines NX[4] from the results of PMUX4 through PMUX7. The output circuit is composed of multiple priority multiplexers, each of which has a block diagram that is similar to the block diagram that is shown in Fig. 10. The priority multiplexers must have M inputs, consistent with the M rule units here. Hence, the details of the output circuit are omitted.

The proposed configurable matching engine can be manufactured as a standalone chip. Accordingly, when the set of processing keywords is changed, only the transitions have to be regenerated according to the new keyword set, and the rule data of the units are updated by the newly generated transitions. Owing to its flexibility, the proposed configurable matching engine can be implemented in ASICs and is not limited to FPGAs. However, a trade-off must be made between flexibility and performance. When the proposed configurable matching engine is flexible, the circuit becomes complicated and the performance is worsened. Nevertheless, advanced semiconductor technologies can compensate for this degradation in performance.

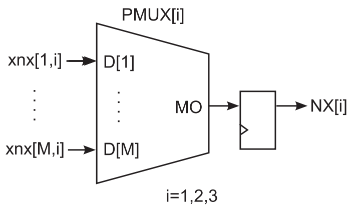


Fig. 15. State circuit for stages 1–3 of the configurable architecture.

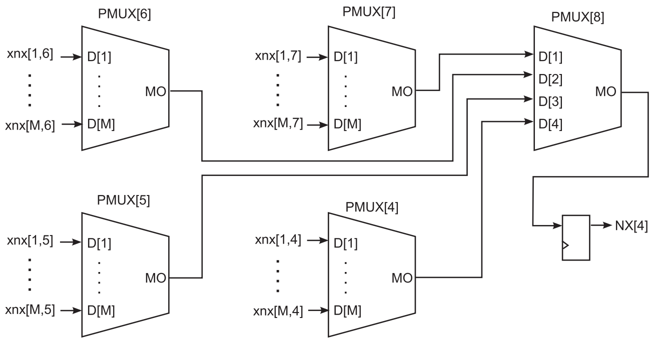


Fig. 16. State circuit for the terminal stage of the configurable architecture.

6. Evaluation and discussion

This section firstly compares the numbers of multi-character transitions with various NFA levels. Next, the proposed architectures are implemented in FPGA and ASIC devices to evaluate the hardware resources required for implementations and to estimate the achievable throughput. Finally, the results herein are compared with those obtained elsewhere.

6.1. Transitions versus NFA levels

In the evaluation of the number of multi-character transitions for various number of levels in the NFA part, 1,000 keywords, retrieved randomly from SNORT rules, are used [18]. The numbers of k-character transitions in cases with $k = 1$, 4, and 8, and NFA levels of 1–15, are determined. Fig. 17 plots the relationship between the number of transitions and the number of levels in the NFA part. The number of goto functions is 10,157 and the number of non-empty output states is 1130, so the number of transitions is $10,157 + (k-1) * 1130$ for a k-character AC-NFA. Dashed lines represent the numbers of transitions of the k-character AC-NFAs.

As seen in the graph in Fig. 17, the number of transitions increase dramatically as the number of NFA levels decreases below four; the curves are flat as the number of NFA levels increases above eight. When the number of NFA levels exceeds 15, the

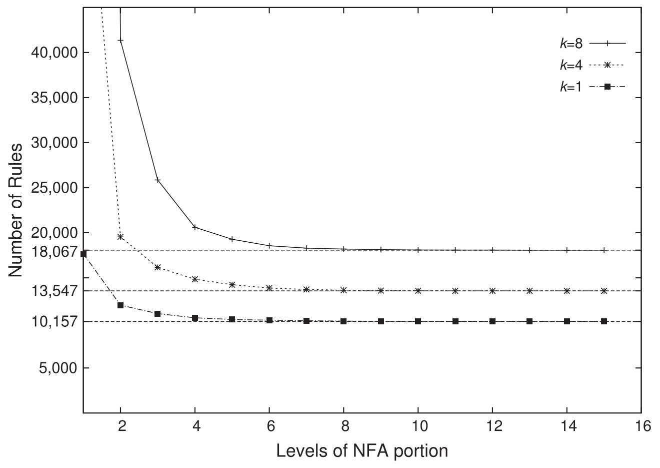


Fig. 17. Compare the rules for different levels of NFA.

number of transitions of the *k*-character hybrid AC-FA equals that of the *k*-character AC-NFA. This comparison reveals that the number of transitions increases almost linearly with *k* when the number of NFA levels exceeds eight. Evidently, the advantage of the hybrid approach increases with *k*. Although the length of the longest keyword is 80, the number of transitions does not increase with the number of NFA levels above 15. Restated, if the matching engine is implemented as an AC-NFA, then 80 stages are required.

6.2. Evaluation of implementations

The proposed architectures are evaluated on FPGA and ASIC devices. The proposed architectures were designed in VHDL code and then built using Altera's development tool, Quartus II. The built architectures were simulated in Modelsim to verify their logic functions. The devices chosen to evaluate the proposed architectures are Stratix IV FPGA and HardCopy IV ASIC from Altera. The FPGA device has 424,960 ALUTs and 424,960 registers and the ASIC device has 9,774,880 HCells.

Since the multi-stage architecture is implemented using decoders and combination logics rather than registers and comparators [19] and must be rebuilt when the keyword set is changed, it is only evaluated on FPGA devices. Since the rule data of the configurable architecture can be reconfigured when the keyword set is changed, it is evaluated on both FPGA and ASIC devices. The number of NFA levels is eight in all of the implementations, the numbers of stages are 10, 13 and 17 for $k = 1$, 4, and 8, respectively. Moreover, the achievable throughput is obtained by multiplying the data width by the clock rate.

To implement multi-stage architecture, 300 keywords are used in the experiments herein, yielding the results in Table 2. The total length of the 300 keywords is 3892 characters. Comparing the results with $k = 4$ and 8 with that with $k = 1$ reveals that the throughputs increase by factors of 3.4 and 6.1, as the numbers of used ALUTs increase by factors of 1.8 and 2.7, respectively. The growth rates of the used ALUTs are lower than those of the throughputs. A comparison of the results also reveals that the maximum operating clock rate is degraded as the number of transitions increases, because the critical path of a priority multiplexer with M inputs comprises log_2M chained multiplexers so the delay of the priority multiplexers increases with the number of inputs.

Next, the configurable architecture is evaluated on FPGA and ASIC devices. Each of the implementations involves 512 rule units for the purpose of verification and evaluation. The pipeline method of Soewito [20], which incorporates pipeline architecture and multi-thread operation, is used to increase the throughput in the evaluation of the configurable architecture. A two-stage pipeline is utilized in the evaluation of the configurable architecture.

Table 3 presents the results of implementing the original and pipeline configurable architectures in FPGAs. The results of implementing the original architecture in FPGAs are elucidated first. The derived throughputs increase by factors of around 2.7 and 4.3 as $k = 1$ increased to $k = 4$ and 8, respectively. With respect to the hardware resources, the numbers of used ALUTs increase by factors of 1.3 and 1.8, and the numbers of used registers increase by factors of 2.3 and 3.9 as $k = 1$ is increased to $k = 4$ and 8, respectively. Next, the results of implementing the pipeline architecture in FPGAs are discussed. The derived throughputs increased by factors of around 3.2 and 5.2 times as $k = 1$ was increased to $k = 4$ and 8, respectively. With respect to the hardware resources, the number of used ALUTs increased by factors of 1.3 and 1.7, and the number of used registers increased by factors of 2.1 and 3.4 as $k = 1$ was increased to $k = 4$ and 8, respectively.

Table 4 summarizes the results of implementing the original and pipeline configurable architectures in ASICs. The results of implementing the original architecture in ASICs are discussed first. The derived throughputs increased by factors of around 3.8 and 5.8 as $k = 1$ was increased to $k = 4$ and 8, respectively. The number of used HCells increased by factors of 1.6 and 2.5 as k was increased to 4 and 8, respectively. Next the results of implementing the pipeline architecture in ASICs are as follows. The derived throughputs

Table 2
Implementing the multi-stage architecture on FPGA.

	$k = 1$	$k = 4$	$k = 8$
Total rules	2813	3756	5013
Used ALUTs	11,741 (3%)	21,460 (5%)	31,258 (7%)
Used registers	132 (<1%)	204 (<1%)	300 (<1%)
Logic utilization (%)	3	6	9
Max. Freq.	83.93 MHz	76.44 MHz	66.6 MHz
Throughput	0.7 Gbps	2.4 Gbps	4.3 Gbps
LE/char	3.81	6.96	10.14

Table 3
Implementing the configurable architecture on FPGA.

	Original			2-Stage pipeline		
	$k = 1$	$k = 4$	$k = 8$	$k = 1$	$k = 4$	$k = 8$
Used ALUTs	60,119 (14%)	77,948 (18%)	106,632 (25%)	59,544 (14%)	79,033 (19%)	102,315 (24%)
Used registers	27,063 (6%)	60,921 (14%)	106,532 (25%)	35,090 (8%)	72,580 (17%)	120,749 (28%)
Logic utilization (%)	24	34	48	24	34	47
Max. freq. (MHz)	34.34	25.97	20.54	74.75	57.95	48.44
Throughput (Gbps)	0.3	0.8	1.3	0.6	1.9	3.1

Table 4
Implementing the configurable architecture on ASIC.

	Original			2-Stage pipeline		
	$k = 1$	$k = 4$	$k = 8$	$k = 1$	$k = 4$	$k = 8$
Used HCells	389,094	626,943	966,938	383,247	629,855	943,581
HCell utilization (%)	4	6	10	4	6	10
Max. freq. (MHz)	77.66	70.76	54.18	172.0	136.46	123.44
Throughput (Gbps)	0.6	2.3	3.5	1.4	4.4	7.9

Table 5
Comparison of different approaches.

Description	Clock (MHz)	Data width (bits)	Throughput (Gbps)
Our multi-stage approach	66.6	64	4.3
Our configurable approach	123	64	7.9
Pao et al. [10] Pipelined implementation	253	8	2.0
Scarpazza et al. [21] Software program	3200	64	1.6–40
Tripp [15] Parallel string matching engine	149	32	4.8

increased by factors of around 3.1 and 5.6 as $k = 1$ was increased to $k = 4$ and 8, respectively. Moreover, the number of used HCells increased by factors of 1.6 and 2.5 as $k = 1$ is increased to $k = 4$ and 8, respectively.

In implementation on ASIC, the maximum operating clock rate with $k = 4$ is 9% worse than that with $k = 1$, while the maximum operating clock rate for $k = 8$ is 23% worse than that with $k = 4$. The degradation of the clock rate is much greater for $k = 8$ perhaps because that the implementation with $k = 8$ uses 32-to-1 multiplexers and 1-to-32 demultiplexers, but the implementations with $k = 1$ and $k = 4$ use 16-to-1 multiplexers and 16-to-1 demultiplexers.

Comparing Tables 2 with 3 reveals that the maximum frequencies of the configurable architecture are lower than those of the multi-stage architecture for implementation on FPGA, perhaps because the configurable architecture is more complex than the multi-stage architecture. Furthermore, since the rule matching function is carried out using decoders with combinational logics rather than registers and comparators in the implementation of the multi-stage architecture, the hardware efficiency and achievable clock rate are much higher than in the implementation of the configurable architecture. In the implementations on both FPGA and ASIC, when k increases, the hardware utilization is increased and the maximum operating clock rate is degraded. The implementations on ASIC have higher clock rates and hardware efficiency. Tables 3 and 4 also reveal that a two-stage pipeline structure can approximately double the throughput, at the cost of only a slight increase in the required hardware resources.

6.3. Comparison of different approaches

Table 5 compares various string matching approaches. The performance data for the approaches other than the one presented herein are taken from the literature. Fairly comparing the approaches is difficult because they differ substantially. Nevertheless, this comparison provides insight into the proposed approach. The approach of Scarpazza et al. [21] is a software implementation on the IBM Cell/B.E. processor, in which columns *Clock* and *Data Width* represent the operating clock rate and the data width of the processor. In the other approaches, the columns *Clock* and *Data Width* represent the clock rate and the bits of the data bus in hardware implementations.

The throughput of the software approach that was presented by Scarpazza et al. varies from 40 Gbps with fewer than 200 keywords to 1.6 Gbps with more than 200 keywords. The pipelined architecture that was developed by Pao et al. [10] can operate at a clock rate of 253 MHz, with a single character processed per cycle and a throughput of 2.0 Gbps. The method of Tripp [15] can process four characters in parallel at a clock rate of 149 MHz, achieving a throughput of 4.8 Gbps. The comparison demonstrates that the throughput of a hardware string matching accelerator can be multiplied up by the inspecting multiple characters in parallel.

7. Conclusion

This work develops a novel hybrid approach, based on the AC-algorithm, for converting an AC-trie into a hybrid AC-FA, which has both NFA and DFA parts. The hybrid AC-FA is extended into a k-character hybrid AC-FA, which can process k characters in parallel. A multi-stage architecture for implementing the derived k-character hybrid AC-FA is developed. The number of the stages can be predetermined based on the number of levels in the NFA part and the number of characters to be inspected in parallel. Consequently, the proposed multi-stage architecture can be made into a configurable architecture. This configurable architecture can then be implemented as a stand-alone string-matching device and its configuration can be updated according to the set of keywords to be processed.

The evaluation of the number of k-character transitions with various NFA levels reveals that the number of k-character transitions increases almost linearly with k when the number of levels in the NFA part is high enough. The results of implementing the proposed multi-stage architecture in FPGA devices reveal that, as the number of characters to be inspected in parallel is increased from one to four and eight, the derived throughput is increased by factors of 3.4 and 6.1, respectively, whereas the hardware cost is increased by factors of only 1.8 and 2.7, respectively. Moreover, the multi-stage architecture with 8-character transitions can be implemented in FPGA at a clock rate of 66.6 MHz with a throughput of around 4.3 Gbps. The results of implementing the configurable architecture in ASICs reveal that the derived throughput is increased by a factor of around 5.8 and the hardware cost is increased by a factor of around 2.5 as the $k = 1$ is increased to $k = 8$. The configurable architecture with 8-character transitions can be implemented in ASICs at a clock rate of 54.18 MHz, yielding a throughput of around 3.5 Gbps. The pipeline scheme enables the proposed configurable architecture with 8-character transitions to be implemented at a nearly doubled clock rate of 123.44 MHz, with a doubled throughput of approximately 7.9 Gbps.

The proposed hybrid AC-FA approach has the advantages of both the DFA and the NFA approaches. This work also shows that the proposed multi-character hybrid AC-FA can be realized with multi-stage architecture. Both high performance and space efficiency can be achieved by combining DFA and NFA approaches.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.
[2] H. Nakahara, T. Sasao, M. Matsuura, A regular expression matching circuit: decomposed non-deterministic realization with prefix sharing and multi-character transition, Microprocess. Microsyst. 36 (8) (2012) 644–664.
[3] M. Alicherry, M. Muthuprasanna, V. Kumar, High speed pattern matching for network IDS/IPS, in: Proceedings of the 2006 14th IEEE International Conference on Network Protocols, 2006, ICNP '06, 2006, pp. 187–196.
[4] N. Hua, H. Song, T.V. Lakshman, Variable-stride multi-pattern matching for scalable deep packet inspection, in: INFOCOM 2009, IEEE, 2009, pp. 415–423.
[5] D. Pao, X. Wang, Multi-stride string searching for high-speed content inspection, Comput. J. 55 (10) (2012) 1216–1231.
[6] C.-C. Chen, S.-D. Wang, A multi-character transition string matching architecture based on Aho-Corasick algorithm, Int. J. Innovat. Comput. Inform. Control (IJICIC) 8 (12) (2012) 8367–8386.
[7] C.-C. Chen, S.-D. Wang, An efficient multicharacter transition string-matching engine based on the Aho-Corasick algorithm, ACM Trans. Archit. Code Optim. (TACO) 10 (4) (2013) 25.
[8] N. Tuck, T. Sherwood, B. Calder, G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection, in: INFOCOM 2004, Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, vol. 4, 2004, pp. 2628–2639.
[9] X. Zha, S. Sahni, Highly compressed Aho-Corasick automata for efficient intrusion detection, in: IEEE Symposium on Computers and Communications, 2008, ISCC 2008, 2008, pp. 298–303.
[10] D. Pao, W. Lin, B. Liu, A memory-efficient pipelined implementation of the Aho-Corasick string-matching algorithm, ACM Trans. Archit. Code Optim. 7 (2) (2010) 10:1–10:27.

[11] W. Lin, B. Liu, Pipelined parallel ac-based approach for multi-string matching, in: 14th IEEE International Conference on Parallel and Distributed Systems, 2008. ICPADS '08, 2008, pp. 665–672.
[12] V. Dimopoulos, I. Papaefstathiou, D. Pnevmatikatos, A memory-efficient reconfigurable Aho-Corasick FSM implementation for intrusion detection systems, in: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007, 2007, pp. 186–193.
[13] M. Becchi, P. Crowley, A hybrid finite automaton for practical deep packet inspection, in: Proceedings of the 2007 ACM CoNEXT Conference, CoNEXT '07, ACM, 2007, pp. 1:1–1:12.
[14] Y. Sugawara, M. Inaba, K. Hiraki, Over 10 Gbps string matching mechanism for multi-stream packet scanning systems, in: Field Programmable Logic and Application, Lecture Notes in Computer Science, vol. 3203, Springer, Berlin Heidelberg, 2004, pp. 484–493.
[15] G. Tripp, A parallel string matching engine for use in high speed network intrusion detection systems, J. Comput. Virol. 2 (1) (2006) 21–34.
[16] V. Rahmanzadeh, M. Ghaznavi-Ghoushchi, A multi-gb/s parallel string matching engine for intrusion detection systems, in: Adv-ances in Computer Science and Engineering, Communications in Computer and Information, vol. 6, Science, Berlin Heidelberg, 2009, pp. 847–851.
[17] N. Yamagaki, R. Sidhu, S. Kamiya, High-speed regular expression matching engine using multi-character NFA, in: International Conference on Field Programmable Logic and Applications, 2008, FPL 2008, 2008, pp. 131–136.
[18] Snort. https://www.snort.org.
[19] C. Clark, D. Schimmel, Scalable pattern matching for high speed networks, in: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, FCCM 2004, 2004, pp. 249–257.
[20] B. Soewito, Packet inspection on programmable hardware, Comput. Eng. Intell. Syst. 4 (2) (2013) 57–68.
[21] D.P. Scarpazza, O. Villa, F. Petrini, Exact multi-pattern string matching on the Cell/B.E. processor, in: Proceedings of the 5th Conference on Computing Frontiers, CF-'08, ACM, 2008, pp. 33–42.

Chien-Chi Chen is a PhD candidate in Department of Electrical Engineering, National Taiwan University. His research interests include embedded systems and reconfigurable architecture.

Sheng-De Wang was born in Taiwan in 1957. He received the B.S. degree from National Tsing Hua University, Hsinchu, Taiwan, in 1980, and the M. S. and the Ph. D. degrees in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1982 and 1986, respectively. Since 1986 he has been on the faculty of the department of electrical engineering at National Taiwan University, Taipei, Taiwan, where he is currently a professor. From 1995 to 2001, he also served as the director of computer operating group of computer and information network center, National Taiwan University. He was a visiting scholar in Department of Electrical Engineering, University of Washington, Seattle during the academic year of 1998–1999. From 2001 to 2003, He has been served as the Department Chair of Department of Electrical Engineering, National Chi Nan University, Puli, Taiwan. His research interests include embedded systems, reconfigurable computing, and intelligent systems.