An Efficient Multicharacter Transition String-Matching Engine Based on the Aho-Corasick Algorithm

CHIEN-CHI CHEN and SHENG-DE WANG, National Taiwan University

A string-matching engine capable of inspecting multiple characters in parallel can multiply the throughput. However, the space required for implementing a matching engine that can process multiple characters in parallel generally grows exponentially with respect to the characters to be processed in parallel. Based on the Aho-Corasick algorithm (AC-algorithm), this work presents a novel multicharacter transition Nondeterministic Finite Automaton (NFA) approach, called *multicharacter AC-NFA*, to allow for the inspection of multiple characters in parallel. This approach first converts an AC-trie to an AC-NFA by allowing for the simultaneous activation of multiple states and then converts the AC-NFA to a k -character AC-NFA by an algorithm with concatenation operations and assistant transitions. Additionally, the alignment problem, which occurs while multiple characters are being inspected in parallel, is solved using assistant transitions. Moreover, a corresponding output is provided for each inspected character by introducing priority multiplexers to determine the final matching outputs during implementation of the multicharacter AC-NFA. Consequently, the number of derived k -character transitions grows linearly with respect to the number k . Furthermore, the derived multicharacter AC-NFA is implemented on FPGAs for evaluation. The resulting throughput grows approximately 14 times and the hardware cost grows about 18 times for 16-character AC-NFA implementation, as compared with that for 1-character AC-NFA implementation. The achievable throughput is 21.4Gbps for the 16-character AC-NFA implementation operating at a 167.36MHz clock.

Categories and Subject Descriptors: C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems; I.5.5 [Pattern Recognition]: Implementation

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: String-matching, deterministic and nondeterministic finite automaton, intrusion detection system

ACM Reference Format:

Chen, C.-C. and Wang, S.-D. 2013. An efficient multicharacter transition string-matching engine based on the Aho-Corasick algorithm. *ACM Trans. Architect. Code Optim.* 10, 4, Article 25 (December 2013), 22 pages. DOI: <http://dx.doi.org/10.1145/2541228.2541232>

1. INTRODUCTION

String matching plays a major role in many applications, including network intrusion detection systems (NIDS) and bioinformatics. String matching generally includes exact string matching and regular expression matching; exact string matching is more efficient yet less flexible for searching keywords in a text than regular expression matching. Some applications like NIDS that must inspect the data stream online usually use exact string matching first to locate suspected data efficiently and then verify the results by other more effective and complex approaches, including regular expression matching. Network bandwidth has increased in line with advances in telecommunication and integrated circuit technologies, necessitating the acceleration of packet inspection of

Author's address: S.-D. Wang; email: sdwang@ntu.edu.tw.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax: +1 (212) 869-0481 or permission@acm.org.

© 2013 ACM 1544-3566/2013/12-ART25 \$15.00

DOI: <http://dx.doi.org/10.1145/2541228.2541232>

NIDS to sustain the network throughput. Hardware string-matching implementation can greatly facilitate packet inspection.

A considerable amount of research on hardware string-matching accelerators is based on the exact string-matching algorithm of Aho and Corasick [1975] (AC-algorithm). According to the AC-algorithm, a prefix tree that consists of goto and failure functions can be created, commonly known as an AC-trie. All occurrences of keywords can be located in a one-pass search through the goto and failure functions in the matching process. However, failure functions in an AC-trie causing multiple state transitions in a matching cycle make it inconvenient to implement the AC-algorithm in a deterministic hardware circuit. Therefore, a general approach is to convert the AC-trie to a Deterministic Finite Automaton (DFA), commonly referred to as AC-DFA, to eliminate the failure functions and then implement the AC-DFA in a lookup table. However, the lookup table approach is space expensive because the number of transitions grows rapidly after converting to a DFA [Tuck et al. 2004; Dimopoulos et al. 2007]. Especially when implementing multicharacter transition-matching architecture, the required space grows exponentially with respect to the number of characters to be inspected in parallel. In early work, most hardware implementations of the string matching inspect the packet data character by character so that the throughput is limited by the achievable clock rate. Some recently developed hardware string-matching approaches have attempted to inspect multiple characters in a single cycle to further increase the throughput of packet inspection [Alicherry et al. 2006; Tripp 2006; Pao and Wang 2012; Rahmanzadeh and Ghaznavi-Ghouschi 2009]. While multiple characters are being inspected in parallel, besides the complexity being increased, a common problem that has to be considered is the alignment problem of starting and ending characters. The alignment problem includes two cases: a pattern does not begin at the first character of the inspecting characters, and a pattern does not end at the last character of the inspecting characters. Next, consider the inspecting characters “they” as an example, in which the pattern “he” begins at the second character and ends at the third character. In this scenario, neither the beginning of the pattern aligns with the first character of the inspecting characters nor the ending of the pattern aligns with the last character of the inspecting characters.

This work presents an AC-algorithm-based approach to enhance the performance of exact string matching. The proposed approach first converts an AC-trie to a non-deterministic finite automaton (NFA), called *AC-NFA*, by removing the failure functions and allowing for the simultaneous activation of multiple states. An intuitive algorithm is also developed to derive multicharacter transitions from the AC-NFA, where each transition can match multiple characters at a time. This derivation algorithm also includes using assistant transitions and a pseudo state to solve the alignment problem. An NFA consisting of the k -character transitions derived from an AC-NFA is referred to herein as a k -character AC-NFA. The derived k -character transitions grows linearly with respect to the number k —for instance, the space required for implementing a k -character AC-NFA is $O(k)$. This work also develops a hardware structure to implement the multicharacter AC-NFA. Owing the possibility of simultaneous activation of multiple states, a corresponding output must be provided for each inspected character similar to the AC-algorithm. The proposed hardware structure involves the use of priority multiplexers to select the matching outputs of the active states in the higher level as the matching results.

The proposed AC-algorithm-based approach is implemented on FPGAs for evaluation. Evaluation results indicate that the throughput and hardware costs grow nearly linearly with respect to the characters inspected in parallel. The resulting throughput grows approximately 14 times, and the hardware cost grows approximately 18 times for a 16-character AC-NFA implementation, as compared with a 1-character AC-NFA

implementation. An implementation of 16-character transitions with 1,000 keywords can operate at 167.36MHz clock and a derived throughput of around 21.4Gbps. Additionally, an implementation of 16-character transitions with 2,000 keywords can operate at 137.34MHz clock and a derived throughput of around 17.6Gbps. The spaces of around 1.25 and 11.8 LEs/character are required to implement the 1-character and 16-character AC-NFAs. The main benefits of this article can be summarized as follows:

- a systematic approach that performs concatenation operations iteratively is devised to derive the AC-algorithm to a multicharacter AC-NFA, in which assistant transitions are introduced to solve the alignment problem. The space required for implementing a derived k -character AC-NFA is $O(k)$; and
- the features of the AC-algorithm that provide a matching output in every matching cycle are preserved by introducing priority multiplexers to determine the corresponding matching output for all inspected characters in every matching cycle in the proposed multicharacter AC-NFA approach.

The rest of this article is organized as follows: Section 2 introduces related string-matching literature. Section 3 then describes the AC-algorithm briefly and implementation of the AC-trie as an NFA. Next, Section 4 describes the derivation and implementation of the multicharacter AC-NFA. Section 5 evaluates the proposed approach and compares it with related work. Conclusions are finally drawn in Section 6.

2. RELATED WORK

Among the string-matching algorithms, the algorithms of Aho-Corasick [1975] and Bloom [1970] are used in many applications for filtering out specific data efficiently. The Bloom algorithm accelerates the string matching by allowing for a small number of falsely matched patterns; however, a further exact verification is required for confirming whether the result is false positive. The Bloom algorithm can be implemented in hardware with high space efficiency [Dharmapurikar et al. 2003]. In contrast, the AC-algorithm is an exact string-matching algorithm that can locate multiple patterns in a text with linear time complexity. Since the proposed approach in this work is mainly based on the AC-algorithm, the following discussion of related work focuses on the AC-algorithm.

Due to the progress and flexibility of the programmable devices such as FPGA, developers can design and evaluate variant architectures according to the features of the AC-algorithm. Nevertheless, the resources of programmable devices are limited, with some works attempting to increase the hardware efficiency. To improve the memory efficiency, Tuck et al. [2004] developed a bitmap-compression and path-compression approach for the AC-algorithm, capable of reducing the required memory and improving the performance on hardware implementation. Zha and Sahni [2008] improved the bitmap-compression and path-compression approach by requiring considerably less memory. Alternatively, Alicherry et al. [2006] implemented the AC-algorithm by integrating a Ternary Content Addressable Memory (TCAM) and an Static Random-Access Memory (SRAM) that utilizes ternary matching of TCAM to achieve the matching of characters expressed in negation expressions, subsequently reducing the space required for storing the state transitions. Pao et al. [2010] and Lin and Liu [2008] used pipeline architectures to implement the character trie that only contains goto functions of the AC-trie to reduce the space introduced by the failure functions. Hua et al. [2009] developed another approach based on a block-oriented scheme instead of the usual byte-oriented processing of patterns to minimize the memory usage. Dimopoulos et al. [2007] developed a Split-AC algorithm that partitions a whole AC-trie into multiple smaller tries to increase memory efficiency.

Because of the flexibility of programmable devices, some works have developed string-matching architectures that can inspect multiple characters in parallel to multiply the throughput of string matching. However, developing an approach capable of inspecting multiple characters in parallel must consider both the complexity and the alignment problem incurred in k -character matching processes. As an extension of the AC-algorithm, Sugawara et al. [2004] proposed a string-matching method called *Suffix-Based Traversing* (SBT) to process multiple input characters in parallel and reduce the lookup table size. Alicherry et al. [2006] proposed a k -compressed AC DFA to achieve a parallel k -character matching engine. A k -compressed AC DFA consists only of the states whose depth is a multiple of k in the original AC-trie and the leaf states of the original AC-trie, where the alignment problem is solved by using additional shallow states. Other works used multiple FSMs to achieve parallelism and solve the alignment problem, where each FSM is responsible for processing a pattern beginning at a different position, respectively [Tripp 2006; Pao and Wang 2012; Rahmanzadeh and Ghaznavi-Ghouschi 2009]. However, those approaches require specific logics to combine the matching results from the FSMs. The approaches of Yamagaki et al. [2008] and Katashita et al. [2006a, 2006b] solve the alignment problem by using additional states and transitions.

However, Salmela et al. [2006] developed a software approach capable of processing multiple characters in parallel. That approach uses short substrings of length q , referred to as q -grams, which process q characters as a single character, and bit parallelism to increase filtering efficiency. Nevertheless, their approach is designed to match a set of keywords with the same length. Because of advanced semiconductor technologies, multiple processing cores can be packaged in a single CPU or GPU chip. Recently, many software implementations of the AC-algorithm use the power of the multicore in CPU or GPU to accelerate string matching. For example, Scarpazza et al. [2008a, 2008b] proposed an optimized algorithm for the IBM Cell/B.E processor, which is a heterogeneous multicore processor comprised of a 64-bit processor core and 8 synergistic processor cores, to achieve high-performance exact string matching. In that algorithm, keywords are split to fit in the local memories of the processing cores to reach extremely high throughput for each processor. Yang et al. [2010] and Yang and Prasanna [2013] derived an approach using a head-body finite automaton (HBFA) to improve the match ratio on multicore processors and implements the HBFA in multiple threads on the multicore system to achieve high throughput. Villa et al. [2009] presented a software approach for the AC-algorithm on the Cray XMT multithreaded shared memory machine, capable of achieving a throughput of 28Gbps. The approach of Tumeo et al. [2010] assigns different packets to different CUDA/GPU threads, as proposed by NVIDIA, to increase the efficiency of pattern matching. Tumeo et al. later evaluated several software implementations of the AC-algorithm on shared and distributed memory architectures [Tumeo et al. 2012]. Herath et al. [2012] applied multicore CPUs to accelerate the string matching used in biology applications.

Software and hardware approaches significantly differ in achieving the parallelism. Software approaches achieve parallelism by splitting an input text into multiple chunks and then processing the chunks by multiple threads, respectively, where each thread still inspects the input text character by character. Conversely, hardware approaches achieve parallelism by inspecting multiple characters in parallel. Both approaches can multiply the throughput of string matching. However, software and hardware approaches also differ in that the former can have a larger dictionary size.

3. AHO-CORASICK ALGORITHM AND NFA APPROACH

This section first describes the AC-algorithm and the AC-trie briefly. Removing the failure functions allows us to convert the AC-trie to an NFA, called an *AC-NFA*. The

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

matching operations of AC-trie and AC-NFA are also compared. Finally, how to obtain the final matching output from AC-NFA is described.

3.1. Aho-Corasick Algorithm and AC-Trie

Figure 1 illustrates an AC-trie built on the keyword set {he, she, his, hers}, which is an example taken from the work of Aho and Corasick. In this figure, the circled numbers denote states, which are called *nodes* alternatively; in addition, the double-circled numbers are output states or output nodes with nonempty matching outputs. The physical lines refer to goto functions, and the dashed lines denote failure functions. State 0 is also known as the initial state or the root node. Every noninitial state has a failure function; the failure functions linked to the initial state are not shown for clarity. A property of the failure function is one that only links a state to another state in the lower depths.

A matching cycle in the AC-algorithm is defined as a period that begins with inputting a character and ends with outputting a matching state. An AC-trie contains only one active state at any time. In a matching cycle, the goto functions of the active state are checked first. If none of the goto functions is matched, then the state transits to a new state through the failure function and the goto functions of the new activated state are checked continuously. The fact that all noninitial states are linked to the initial state through the failure functions and the initial state has the goto functions for all characters ensures that a matched goto function can be found in every matching cycle. During matching, an input string is processed character by character; the character under processing is referred to as an inspecting character. A matching cycle leads to a matching output, which is represented by a state number. A situation in which a keyword is matched implies that the matching output is a nonzero state number; otherwise, the matching output is state 0.

Figure 2(a) illustrates an example of the matching operation of the AC-trie with an input string “ushers.” The first character “u” is neither matched with “h” nor “s”; the state stays at 0. For the following three characters “she,” the state transits from 0 to 3, 4, and 5 sequentially. At state 5, the matching output is {she he}. State 5 has no goto function. Consequently, when the following character “r.” is processed, the state transits to state 2 through the failure function of state 5 and then transits to 8 according to the matched goto function of state 2. For the final character “s,” the state transits to 9 and the matching output is “hers.”

With the failure functions, all matched keywords can be found in a one-pass search by using an AC-trie. However, when an AC-trie is implemented in hardware, the complexity increases due to the property in which more than one state transition is often

Fig. 2. Matching operations of AC-trie and AC-NFA.

Fig. 3. AC-NFA and output selection circuit.

required to find a matched goto function through the failure functions. A general approach converts an AC-trie to a DFA to eliminate the failure functions and then implements the DFA in a lookup table. However, the space utilization of the lookup table approach is generally inefficient, because the transitions of a DFA-version AC-trie are growing explosively and the lookup table is sparse.

3.2. AC-NFA Approach

If the failure links are removed and simultaneously activation of multiple states is allowed, an AC-trie becomes an NFA. Figure 3(a) illustrates the AC-NFA obtained from the AC-trie in Figure 1 by removing the failure links. After converting an AC-trie to an AC-NFA, all matched transitions are done concurrently. The parallelism implicit in hardware makes it more feasible to keep track of the concurrent state transitions. The transitions of an AC-NFA are only the goto functions of the original AC-trie. In the proposed approach, the complexity in terms of number of transitions remains the same, whereas the failure functions are transformed into the concurrent transitions that fit the hardware intrinsically.

Figure 2(b) illustrates the example of the matching operations of an AC-NFA with the same input string “ushers.” No transition is triggered after accepting the first character “u.” State 3 is activated after accepting the second character “s.” Following

acceptance of the third character “h,” states 4 and 1 are activated simultaneously. States transit to 5 and 2, respectively, on the fourth character “e.” Both states 5 and 2 output matched strings that are {she he} and {he}, respectively. However, the final matched output should be {she he} since it includes the string “he.” After accepting the final two characters “r” and “s,” the state transits from 2 to 8, then 9. The final input character also activates state 3. Comparing Figures 2(a) and 2(b) reveals that when a state is activated in AC-trie, all of the states linked to through the failure functions from the active state are activated simultaneously in AC-NFA in every matching cycle. Therefore, the failure functions are unnecessary if simultaneous activation of multiple states is allowed.

In an AC-trie, each state represents a unique string. If a failure function links state S_1 to state S_2 , then the string represented by S_2 is the postfix of the string represented by S_1 . For example, state 2 represents the string “he,” and state 5 represents the string “she”; in addition, the failure function of state 5 points to state 2, and the string “he” is the postfix of the string “she.” In an AC-trie, the matching output is simply the active state since only one state is activated at any time. Although activation of multiple states is allowed in an AC-NFA, the proper matching output from the multiple active states must be determined. For example, like the earlier case, states 5 and 2 are activated simultaneously. Since failure functions link higher states to lower-level states, the highest-level activated state in an AC-NFA should determine the final matching output.

In the matching operation of the AC-algorithm, only a matching output is generated after every matching cycle. In the proposed NFA approach, the priority multiplexer PMUX shown in Figure 3(b) is used as an output selection circuit to determine the final matching outputs from the activated output nodes. Our example contains four output nodes so that the priority multiplexer PMUX has four input groups (E1, D1) to (E4, D4), where inputs E1 through E4 are control signals and inputs D1 through D4 are data signals. The control signals E1 through E4 are used to indicate whether the data inputs D1 through D4 are valid or not, respectively. If the inputs E_i and E_j are both true and $i < j$, then the priority of input E_i is higher than that of input E_j ; in addition, the priority multiplexer selects the input D_i to be output through Dout. When none of the inputs is valid, the output Dout is not valid either. However, the output Dout of PMUX can output 0 if no matched output is available. Notation $st(s)$ refers to the status of node s , which is true when node s is activated. Since higher-level nodes have a higher priority, signal $st(9)$ has the highest priority and is connected to E1. The data sent to inputs D1 through D4 are the corresponding state numbers. Consider the previous example. Following acceptance of the string “ushe,” both nodes 5 and 2 are activated and both $st(5)$ and $st(2)$ are true. Moreover, since the priority of $st(5)$ is higher than that of $st(2)$, PMUX selects the data 5 input from D3 as the matching output.

4. MULTICHARACTER AC-NFA APPROACH

In developing a string-matching engine capable of inspecting multiple characters in parallel, the approaches for solving the alignment problem can be classified into two categories. One category of previous work [Tripp 2006; Pao and Wang 2012] solves the alignment problem by using multiple copies of the same string-matching engines. Alternatively, the other category [Yamagaki et al. 2008; Katashita et al. 2006a] solves the alignment problem by additional transitions that include wildcard characters and redundant states. The approach of Yamagaki et al. first adds redundant states and wildcard characters for the first and final nodes and then concatenates every transition with its successive transition. The approach of Katashita et al. first generates multiple copies for each pattern, in which each copy is concatenated with different numbers of wildcard characters before and after that pattern and then merges the common

prefixes of the patterns and the copies to generate a multicharacter AC-NFA. The proposed approach also uses additional states and transitions to solve the alignment problem. With the defined assistant transitions, the proposed approach can simply use concatenation operations iteratively to derive the multicharacter AC-NFA.

The multicharacter NFA derived by the proposed approach should be the same as the results of previous approaches, such as those of Yamagaki et al. [2008] and Katashita et al. [2006a]. However, those approaches differ in the way that they deal with the final matching results. In the work of Katashita et al., the string-matching engine provides a matching flag that indicates whether the corresponding pattern is matched or not for each pattern. In the work of Yamagaki et al., a matching result is provided for each regular expression, whereas the matching result can be a matched state or a flag that indicates whether the corresponding regular expression is matched or not. This work focuses on developing a multicharacter string-matching engine that can preserve the properties of the AC-algorithm, providing a matched state for each inspected character. To achieve this objective, the final matching outputs in the implementation of the k -character AC-NFA are determined using priority multiplexers.

This section first introduces some definitions and then explains how to derive the multicharacter transitions from an AC-NFA by examples. The derivation algorithm is mainly based on an approach of multicharacter AC-DFA developed in our previous work [Chen and Wang 2012]. Next, a whole multicharacter AC-NFA is constructed using the derived multicharacter transitions, followed by a description of the implementation of the derived multicharacter AC-NFA. Finally, an illustrative example explains the matching operation of the proposed multicharacter AC-NFA.

4.1. Deriving Multicharacter Transitions

Exactly how to derive multicharacter transitions from an AC-trie is described as follows. In the deriving procedure, the desired multicharacter transitions are derived by using the goto functions of an AC-trie as one-character transitions. Before the algorithm is described, some definitions used in deriving the multicharacter transitions are provided.

Definition 1. A k -character transition $\delta_k(S_1, T) = S_2$ represents the state transition from the current state, S_1 , to the next state, S_2 , on a k -character string T .

For example, a two-character transition $\delta_2(1, \text{er}) = 8$ represents a situation in which the state transits from 1 to 8 on a two-character string “er.”

Definition 2. A transition $\delta_k(S_2, T_2) = S_3$ is a successive transition of the transition $\delta_l(S_1, T_1) = S_e$ if $S_2 = S_e$, implying that the starting state of $\delta_k(S_2, T_2) = S_3$ is simply the end state of $\delta_l(S_1, T_1) = S_e$.

A new multicharacter transition can be obtained by concatenating a transition with its successive transition. The concatenation operation is defined as follows:

Definition 3 (Concatenation of Two Transitions). Given a k -character transition $\delta_k(S_1, T_1) = S_2$ and an l -character successive transition $\delta_l(S_2, T_2) = S_3$, where S_1 , S_2 , and S_3 denote states, T_1 represents a k -character string, and T_2 represents an l -character string. Then, the concatenation of the two transitions is a $(k+l)$ -character transition $\delta_{k+l}(S_1, T_1T_2) = S_3$.

In addition to the goto functions, some assistant transitions must be defined to help derive the multicharacter transitions that can solve the alignment problem. Figure 4 illustrates examples of some assistant transitions. The circled character “-” in the figures represents a pseudostate that is a virtually defined state and nonexistent in an AC-trie. Here, the symbol “?” represents an arbitrary character. Next, three types

Fig. 4. Assistant transitions.

Fig. 5. Examples of deriving 3-character transitions.

of assistant transitions are defined to facilitate the construction of the multicharacter transitions capable of solving the alignment problem.

Definition 4 (Assistant Transitions). The first type of the assistant transitions is defined as $\delta_1(0, ?) = 0$, which denotes a transition from state 0 to state 0 on an arbitrary character. The second type of assistant transitions is defined as $\delta_1(S_{op}, ?) = -$, which denotes a transition from an output state S_{op} to a pseudostate on an arbitrary character. The third type of the assistant transitions is defined as $\delta_1(-, ?) = -$, which denotes a transition from a pseudostate to another pseudostate on an arbitrary character.

The assistant transitions are represented by dashed lines to distinguish them from the goto functions. The first-type assistant transition (i.e., $\delta_1(0, ?) = 0$) deals with the alignment problem in which the beginning of a pattern does not appear in the first character of the inspecting characters. The second-type assistant transition (i.e., $\delta_1(S_{op}, ?) = -$) preserves the matching output for a situation in which the ending of a pattern does not appear in the final character of the inspecting characters. The transitions beginning from states 2, 5, 7, and 9, respectively, and ending at pseudostates (as shown in Figure 4) are examples of the second-type assistant transitions. The third-type assistant transition (i.e., $\delta_1(-, ?) = -$) can follow a second-type assistant transition to form a multicharacter transition.

Next, the examples shown in Figure 5 explain how to derive multicharacter transitions by repeating the concatenation operation. In this figure, “+” denotes the concatenation operation. First, referring to the example in Figure 5(a), concatenating

two 1-character transitions $\delta_1(0, ?) = 0$ and $\delta_1(0, ?) = 0$ obtains a 2-character transition $\delta_2(0, ??) = 0$; then concatenating the new derived transition with $\delta_1(0, h) = 1$ obtains a 3-character transition $\delta_3(0, ??h) = 1$. Despite obtaining a 3-character transition $\delta_3(0, ???) = 0$, concatenation of the 2-character transition $\delta_2(0, ??) = 0$ and another transition $\delta_1(0, ?) = 0$, it is discarded owing to its uselessness. By using the 3-character transition $\delta_3(0, ??h) = 1$, the alignment problem arising from the situation in which the first character of a keyword appears at the third character of the input string is dealt with. The transition $\delta_3(0, ??h) = 1$ implies that the state stays at state 0 for any of the first two characters and then transits to state 1 on the third character “h.”

Next, consider the example in Figure 5(b), in which the concatenation of $\delta_1(0, ?) = 0$ and $\delta_1(0, h) = 1$ is a 2-character transition $\delta_2(0, ?h) = 1$; in addition, concatenating the newly derived transition with $\delta_1(1, e) = 2$ yields a 3-character transition $\delta_3(0, ?he) = 2$. For a 1-character transition, the matching output can be represented by the destination state. However, the intermediate states are concealed in a multicharacter transition, and the matching outputs corresponding to the intermediate states are unavailable. Therefore, the matching outputs corresponding to the intermediate states must be preserved in the deriving procedure. In these examples, the matching outputs corresponding to the characters are denoted beside the destination state where an empty string is represented by the symbol ‘ Φ ’. The matching outputs are not shown for a state if all matching outputs are empty strings.

Finally, consider the example in Figure 5(c), in which concatenating $\delta_1(1, e) = 2$ with $\delta_1(2, r) = 8$ obtains a 2-character transition $\delta_2(1, er) = 8$. However, since state 2 is an output state, $\delta_1(1, e) = 2$ must be concatenated with the assistant transition $\delta_1(2, ?) = -$, which obtains $\delta_2(1, e?) = -$, to preserve the matched output of node 2. Concatenating the derived transition $\delta_2(1, er) = 8$ with $\delta_1(8, s) = 9$ obtains a 3-character transition $\delta_3(1, ers) = 9$. Concatenating the other derived transition $\delta_2(1, e?) = -$ with the assistant transition $\delta_1(-, ?) = -$ obtains a 3-character transition $\delta_3(1, e??) = -$, which is used to preserve the matched output corresponding to node 2. Derivation of the transition $\delta_3(1, e??) = -$ demonstrates the use of the assistant transitions $\delta_1(2, ?) = -$ and $\delta_1(-, ?) = -$. By using the described concatenation procedure, all of the 3-character transitions from an AC-trie can be obtained. Furthermore, all k -character transitions can be derived from a given AC-trie for any required number k by simply repeating the concatenation operation iteratively.

From the previous examples, we can infer that when the characters under simultaneous inspection are increased by one more, the number of transitions increased is equal to the number of output states. This observation is owing to that each output state is concatenated with an assistant transition to preserve the matching output for a situation in which the following character is not matched. Next, consider the transition $\delta_1(1, e) = 2$ as an example, in which concatenating it with the successive transition $\delta_1(2, r) = 8$ can obtain $\delta_2(1, er) = 8$ and concatenating it with the assistant transition $\delta_1(2, ?) = -$ can obtain $\delta_2(1, e?) = -$. The latter transition $\delta_2(1, e?) = -$ preserves the matching output for a situation in which only the first character is matched with ‘e.’

Consequently, the number of k -character transitions grows linearly with respect to k , that is, the number of characters inspected in parallel. The result can be summarized as the following theorem.

THEOREM 1. *For a given AC-trie with the number of 1-character transitions denoted as r_1 and the number of output states denoted as n_{op} , the number of k -character transitions is $r_k = r_1 + (k - 1) * n_{op}$.*

For example, the AC-NFA in our example contains nine 1-character transitions and four output states; thus, the number of 3-character transitions is $9 + (3 - 1) * 4 = 17$. We

ALGORITHM 1: Algorithm for Constructing k -Character Transition Set**Input:** k : number of characters $NXSET$: 1-character transition set

(including goto functions and assistant transitions)

Output: $TRSET$: k -character transition set**Method:**

```

1. begin
2.    $TRSET \leftarrow \text{empty}$ 
3.   for each state  $S_i$  do
4.     begin
5.        $NSET \leftarrow \text{all 1-character transitions of } S_i \text{ in } NXSET$ 
6.       repeat  $k-1$  do
7.         begin
8.            $TMPSET \leftarrow \text{empty}$ 
9.           for each transition  $NX_i$  in  $NSET$  do
10.            begin
11.               $NX\_ST \leftarrow \text{next state of } NX_i$ 
12.              for each transition  $NX_j$  of  $NX\_ST$  do
13.                begin
14.                   $NEW\_TR \leftarrow \text{concatenate } NX_i \text{ with } NX_j$ 
15.                   $TMPSET \leftarrow TMPSET \cup NEW\_TR$ 
16.                end
17.              end
18.               $NSET \leftarrow TMPSET$ 
19.            end
20.           $TRSET \leftarrow TRSET \cup NSET$ 
21.        end
22.      remove unused transitions from  $TRSET$ 
23.      return  $TRSET$ 
24. end

```

can, therefore, infer that the space required for implementing a k -character AC-NFA is $O(k)$.

4.2. Algorithm for Deriving Multicharacter Transitions

Algorithm 1, a generalized algorithm, derives multicharacter transitions from an AC-trie. In this algorithm, input parameter k is the number of characters to be inspected in parallel. The input parameter $NXSET$ contains the original 1-character transitions that include the goto functions and assistant transitions. The output variable $TRSET$ contains the resulting k -character transitions. By using multiple-level iterations, this algorithm derives the k -character transitions for every state in the original AC-trie.

In line 2, $TRSET$ is initialized. Statements in the loop between lines 3 and 21 derive all of the k -character transitions of every state S_i . In line 5, the 1-character transitions of state S_i are duplicated to variable $NSET$. The loop between lines 7 and 19 is repeated $k-1$ times, in which the 1-character transitions of S_i are concatenated with their successive 1-character transitions iteratively to derive the k -character transitions of S_i . After executing the loop, $NSET$ contains all of the k -character transitions of S_i . In line 20, $NSET$ is added to $TRSET$. The algorithm then returns to line 5 to process the next state continuously. The algorithm is terminated when all of the states are processed. Finally, $TRSET$ contains the derived k -character transitions.

Fig. 6. A 3-character AC-NFA and its output selection circuit.

Because the intermediate state is concealed after two transitions are 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.

According to Theorem 1, the number of i -character transitions is $r_i = r_1 + (i - 1) * n_{op}$, implying that r_i concatenation operations are required to derive the i -character transitions from $(i-1)$ -character transitions. Therefore, the total required concatenation operations T_c to derive the 1-character transitions to k -character transitions can be obtained as follows:

$$T_c = \sum_{i=2}^k (r_1 + (i - 1) * n_{op}) = (k - 1) * r_1 + \frac{2k}{k - 1} * n_{op}$$

Consequently, the time complexity of Algorithm 1 is $O(k * r_1 + n_{op})$.

4.3. Implementation of Multicharacter AC-NFA

Figure 6 depicts the 3-character transitions derived previously as three disjoint 3-character AC-NFAs. Although the disjoint 3-character AC-NFAs can be merged to a single 3-character AC-NFA, the disjoint AC-NFAs are explained more clearly. Since the final matching outputs are determined by priority multiplexers and the matching

outputs of the nodes in higher levels have higher priorities, the nodes of the three individual AC-NFAs are arranged according to their levels in the original AC-trie to more clearly illustrate the relationship of the nodes.

For clarity, pseudonodes are labeled as V1 through V8. The matching outputs containing nonempty strings are denoted along with the corresponding node. For example, the notation (2, 0, 9) along with node 9 implies that when state 9 is activated, the matching outputs that correspond to the three inspecting characters are states 2, 0, and 9, or “he,” an empty string, and “hers.” Level numbers denoted at the top are the levels corresponding to the original AC-trie. During the matching operation of the AC-algorithm, a matching output is generated after every matching cycle. In this example, three characters are inspected in parallel, allowing for generation of three corresponding matching outputs, OP1 through OP3, after every matching cycle. The output selection circuit consisting of three priority multiplexers, PMUX1 through PMUX3, as shown on the right side determines the final matching outputs OP1 through OP3 from the matching results of the activated nodes. As described earlier, matching outputs of the nodes in higher levels have higher priorities.

Since “?” represents an arbitrary character, multiple transitions could be matched simultaneously in the same level. For example, transition $\delta_3(0, \text{he?}) = V5$ is always matched when transition $\delta_3(0, \text{her}) = 8$ is matched. However, transition $\delta_3(0, \text{he?}) = V5$ preserves the matching output of pattern “he.” Therefore, the matching output corresponding to the second inspecting character can be determined by node V5 alone when the two transitions are matched simultaneously. As another example, when state 9 is activated, pseudostate V1 is also activated, where the first matching outputs of node 9 and pseudonode V1 are the same. Therefore, only $\text{st}(V1)$ is sent to input E4 of priority multiplexer PMUX1. Therefore, when multiple transitions in the same level are matched simultaneously, the matching output is determined by the pseudonode appended for preserving the matching output for the corresponding inspecting character. The relationship between the pseudonodes and the actual nodes can be grouped as follows: V1-V5-2, V2-V6-5, V3-V7-7, and V4-V8-9. Each group member belongs to the matching outputs corresponding to the first through third inspecting characters, respectively. According to this figure, the control inputs of PMUX1 and PMUX2 are all status functions of the pseudonodes. Consequently, with an increasing number of characters to be inspected in parallel, only the priority multiplexers required for determining the matching outputs must be increased accordingly while the inputs of each priority multiplexer are the same.

The earlier description can be generalized to the case inspecting k -characters in parallel, in which the derived k -character transitions can be grouped to k disjoint k -character AC-NFAs. Notably, each AC-NFA is responsible for dealing with a misalignment case. In addition, k priority multiplexers are required to determine the final k matching outputs, and each multiplexer has n_{op} inputs.

Next, this work describes the logic circuit for implementing the multicharacter transitions where the character matching function is implemented by decoders with combinational logics instead of comparators [Clark and Schimmel 2004]. Moreover, the approach of Sidhu and Prasanna [2001] that implements the comparators by the Look-Up Tables (LUTs) of FPGAs can be used as well. Figure 7 illustrates an example of the logic circuit of four transitions. In the upper portion of this figure, devices DEC1, DEC2, and DEC3 are 8-to-256 decoders, with each one used to decode one input character to 256 signals. Signals $\text{dec1}(i)$, $\text{dec2}(i)$, and $\text{dec3}(i)$ represent the i -th decoded signals of input characters C1 through C3, respectively, where i is an integer between 0 and 255. For example, when input character C1 is “e,” which corresponds to ASCII code 101, the signal $\text{dec1}(101)$ is true. In this figure, the notation $\text{dec1}(\text{“e”})$ instead of $\text{dec1}(101)$ is used for clarity.

Fig. 7. Implementation of transitions.

In addition to depicting the transitions used as the example in the lower left portion, Figure 7 also shows the corresponding logic circuit in the lower right portion. Where signal $st(s)$ denotes the activating signal for the node s , in which s is 1, 9, V1, or V7. Consider a situation in which S1 is true and $dec1("e")$, $dec2("r")$, and $dec3("r")$ are true as well. In this situation, signal $st(9)$ is true, implying that transition $\delta_3(1, \text{ers}) = 9$ is matched, and node 9 is activated in the next clock.

4.4. Example of Matching Operation

This work demonstrates the effectiveness of the proposed approach by using an example to explain the matching operation. Figure 8 shows the complete block diagram of the 3-character string-matching engine, and Figure 9 shows the matching example. In every matching cycle, three characters are input via C1, C2, and C3 in parallel; in addition, three corresponding matching outputs are generated from OP1, OP2, and OP3 after one clock cycle. Three registers are connected after PMUX1, PMUX2, and PMUX3 to save the matching outputs. In this example, the input string to be matched is "ushishers," which is divided into three 3-character substrings "ush," "ish," and "ers" and then processed in three consecutive matching cycles, respectively. This figure displays only the triggered transitions; the transitions with higher priorities are arranged in the upper portion. The matching results of each matching cycle are depicted in the bottom portion of this figure, which are represented by state numbers and strings both for clarity. Before the matching procedure, all of the states are initialized. In the first matching cycle, according to the input characters "ush," the matched transitions are $\delta_3(0, ?sh) = 4$ and $\delta_3(0, ??h) = 1$. Moreover, both of the matching outputs determined by these two triggered transitions are empty strings.

In the second matching cycle, according to the input characters "ish" and the next states determined in the previous cycle, the matched transitions are $\delta_3(1, is?) = V7$, $\delta_3(0, ?sh) = 4$, and $\delta_3(0, ??h) = 1$. The matching outputs of node V7 are (0,7,-), whereas nodes 1 and 4 have no matched output. Therefore, only the second matching result OP2 is nonempty, which is state 7 or "his" according to node V7.

Fig. 8. Block diagram of a matching engine.

Fig. 9. Example of matching operations.

In the third matching cycle, according to the input characters “ers” and the next states determined in the previous matching cycle, the matched transitions are $\delta_3(4, e??) = V2$, $\delta_3(1, ers) = 9$ and $\delta_3(1, e??) = V1$, and $\delta_3(0, ??s) = 3$. The matching results of pseudonodes V1 and V2 are (2,-,-) and (5,-,-), respectively, in which both V1 and V2 have matching outputs that correspond to the first input character. Since node V2 (which is in level 5) has a higher priority than node V1 (which is in level 4), the final matching result OP1 corresponding to the first input character is determined as

Fig. 10. Waveform of matching operations.

state 5 or {she he}. Only node 9 has a valid third matching result; thus, the matching result OP3 that corresponds to the third character is 9 or "hers."

The proposed approach is implemented in FPGAs. Figure 10 shows the waveform of the matching operations with the input text "ushishers." Before the beginning of every cycle, the inspecting characters are input from C1, C2, and C3. The matching results are then output to OP1, OP2, and OP3 after the next rising edges of the clock cycle. In cycle 1, the first three characters "u," "s," and "h" are input, and the matching results are all zero. In cycle 2, the inspecting characters are "i," "s," and "h," and only OP2 has a matched output (i.e., state 7) after the rising edge of cycle 2. In cycle 3, the final three characters "e," "r," and "s" are processed, and OP1 and OP3 have the matched outputs that are states 5 and 9, respectively, after the next rising edge.

5. EVALUATION AND COMPARISON

This section first implements the proposed architecture on FPGA devices to evaluate the feasibility of using hardware resources and estimate achievable throughput. Our results are then compared with those of previous work.

5.1. Evaluation of Implementations

The keywords for evaluation are extracted from the index of the King James Bible and the rules of Snort. The implementations of 1,000 and 2,000 Bible keywords with $k = 4, 8, 12,$ and 16 are evaluated first. For the case of 1,000 Bible keywords, the total length is 7,400 characters and the average length is 7.4 characters; in addition, the AC-trie built on these keywords has 3,982 states, in which 1,125 states are output states. Since the number of goto functions is equal to the number of the states in an AC-trie, the number of the transitions of the k -character AC-NFA is $r_k = 3982 + 1125 * (k-1)$. For the other case of 2,000 keywords, the total length is 15,414 characters and the average length is 7.7 characters; in addition, the built AC-trie has 8,681 states, in which 2,391 states are output states. Consequently, the number of transitions is $r_k = 8681 + 2391 * (k-1)$ for the k -character AC-NFA of 2,000 keywords.

A significant amount of research on IDS adopts the Snort rules to evaluate the performance of string matching. Therefore, in this work, Snort keywords are also evaluated for making a comparison with the results of related works. During the evaluation of 1,000 keywords extracted from Snort rules, the total length is 13,566 characters and

Table I. Evaluation with 1,000 Bible Keywords

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	3,982	7,357	11,857	16,357	20,857
Logic utilization	2%	5%	10%	16%	21%
ALUTs	3,583 (<1%)	18,113 (4%)	34,588 (8%)	53,579 (13%)	70,100 (16%)
Registers	3,106 (<1%)	3,142 (<1%)	3,190 (<1%)	3,238 (<1%)	3,286 (<1%)
Fmax	190.33MHz	180.18MHz	162.15MHz	162.07MHz	167.36MHz
Throughput	1.5Gbps	5.8Gbps	10.4Gbps	15.6Gbps	21.4Gbps
LE/char	1.13	3.59	6.38	9.60	12.40

Table II. Evaluation with 2,000 Bible Keywords

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	8,681	15,854	25,418	34,982	44,546
Logic utilization	5%	11%	23%	33%	43%
ALUTs	8,038 (2%)	38,805 (9%)	76,427 (18%)	111,801 (26%)	147,626 (35%)
Registers	6,902 (2%)	6,944 (2%)	7,000 (2%)	7,056 (2%)	7,112 (2%)
Fmax	164.15MHz	142.8MHz	138.52MHz	135.72MHz	137.34MHz
Throughput	1.3Gbps	4.6Gbps	8.9Gbps	13.0Gbps	17.6Gbps
LE/char	1.21	3.71	6.77	9.64	12.55

Table III. Evaluation with 1,000 Snort Keywords

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	10,157	13,547	18,067	22,587	27,107
Logic utilization	4%	8%	16%	24%	31%
ALUTs	6,209 (<1%)	28,589 (7%)	53,386 (13%)	78,715 (19%)	102,030 (24%)
Registers	9,193 (<1%)	9,235 (2%)	9,291 (2%)	9,347 (2%)	9,403 (2%)
Fmax	180.28MHz	172.47MHz	130.17MHz	143.39MHz	131.27MHz
Throughput	1.4Gbps	5.5Gbps	8.3Gbps	13.8Gbps	16.8Gbps
LE/char	1.42	3.49	5.78	8.11	10.27

the average length is 13.6 characters; in addition, the built AC-trie contains 10,157 states and 1,130 output states. Consequently, the number of transitions for k -character AC-NFA is $r_k = 10157 + 1130 * (k-1)$.

We developed programs according to our proposed algorithm to derive the multi-character transitions from the keywords and convert the derived transitions to VHDL codes. The generated VHDL codes were compiled and synthesized by Altera's development tool Quartus II 9.1. The hardware function was verified by ModelSim-Altera 6.5b software. The building environment was equipped with an AMD FX-8150 Processor running at 3.4GHz and 16GB RAM. The device that we selected for evaluating the proposed architecture is an Altera's Stratix IV family FPGA EP4SE530F43C2 that has 424,960 adaptive lookup tables (ALUTs) and 424,960 registers, where an ALUT is a logic unit used in the Altera's FPGA devices. According to the datasheet provided by Altera Corporation [2013], one ALUT is equivalent to about 1.25 logic elements (LEs) for the Stratix II device and later device families. The achievable throughput is derived by multiplying the data width with the maximum frequency (F_{max}) reported by the development tool. The hardware resource required for a character is represented by LE/char, which is calculated as follows:

$$(\# \text{ of ALUTs} + \# \text{ of Registers}) * 1.25 / (\text{total characters})$$

Tables I and II summarize the evaluation results for 1,000 and 2,000 Bible keywords, respectively. In addition, Table III lists the evaluation results for 1,000 Snort keywords. Figure 11 shows the curves of the hardware costs, maximum frequencies, and derived

Fig. 11. Resource requirement, maximum frequencies, and throughputs for the implementations with different k .

throughputs for the implementations with respect to different k values. The curves reveal that the throughput and hardware costs are linearly proportional with respect to k , whereas the maximum frequencies (F_{max}) are slightly decayed as k increases.

Tables I and II indicate that the throughputs grow approximately 14 times, and the used ALUTs grow approximately 18 times for $k = 16$ with respect to $k = 1$ for both cases. In the case of 1,000 Snort keywords, the throughputs grow approximately 12 times and the used ALUTs grow approximately 16 times for $k = 16$ with respect to $k = 1$, as shown in Table III. In these implementations, the used ALUTs grow nearly linearly with respect to k , whereas the used registers remain nearly the same since the states do not increase as k increases and the pseudostates are not saved.

Analysis results indicate that the maximum frequency F_{max} is degraded slightly as k increases. The maximum frequencies F_{max} in the case of 2,000 Bible keywords are always lower than in the case of 1,000 Bible keywords for different k values, even though the former has fewer transitions. This phenomenon might be owing to the fact that the critical path of the circuit is dominated by complexity of the output selection circuit that depends on the number of the output states (n_{op}). Routing between the output states and the output selection circuit becomes more complex as n_{op} increases, whereas the routing complexity increases slightly as k increases. Moreover, maximum frequencies F_{max} in the case of 1,000 Snort keywords are lower than that in the case of 1,000 Bible keywords. This difference might be owing to the fact that the average length of the Snort keyword is nearly twice that of the average length of the Bible keyword.

In these implementations, the priority multiplexers consist of multiple levels of multiplexers, and the critical path is the delay produced by $\log_2 n_{op}$ levels of multiplexers for n_{op} output states, explaining why the delay of the priority multiplexers is longer for more inputs. In the proposed approach, the number of inputs for a priority multiplexer is equal to the number of output states. In the case of

Table IV. Evaluation with 1,000 Snort Keywords for Matching-Flag Output

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	10,157	13,547	18,067	22,587	27,107
Logic utilization	7%	9%	16%	24%	33%
ALUTs	1,701 (1%)	10,105 (7%)	15,372 (11%)	22,751 (16%)	30,226 (21%)
Registers	8,682 (6%)	8,195 (6%)	7,804 (6%)	7,841 (6%)	7,860 (6%)
Fmax	333.44MHz	280.5MHz	245.94MHz	203.79MHz	167.11MHz
Throughput	2.7Gbps	9.0Gbps	15.7Gbps	20.0Gbps	21.4Gbps
LE/char	0.96	1.69	2.14	2.82	3.51

1,000 Bible keywords, 1,125 output states exist and the critical path of the priority multiplexer is 11 levels of multiplexers according to $\log_2 1125 = 10.13$. In the case of 2,000 Bible keywords, 3,982 output states exist and the critical path of the priority multiplexer is 12 levels of multiplexers according to $\log_2 2391 = 11.22$. In the case of 1,000 Snort keywords, 1,130 output states exist and the critical path of the priority multiplexer is 10 levels of multiplexers according to $\log_2 1130 = 10.14$.

In some applications (e.g., the string-matching circuit for IDS described in the article of Katashita et al. [2006a, 2006b], in which only the matching output for each pattern is required), the priority multiplexers can be replaced with OR-gates. This work also examines how the priority multiplexers influence performance by undertaking additional experiments with the architecture that determines the matching outputs by OR-gates instead of priority multiplexers. During this evaluation, the implementation is built on a smaller FPGA EP4SGX180DF29C2X, which has 140,600 ALUTs and 140,600 registers. Table IV summarizes the evaluation results with only matching flags as output. Comparing Table III with Table IV reveals that when the matching outputs are determined by priority multiplexers, the required ALUTs increase approximately 3.36 times. Additionally, F_{max} decreases by approximately 38%, implying that the critical path delay increases on average by 1.61 times.

In Katashita et al. [2006a], the hardware resources required for every character are 0.83 LE/char and 4.13 LE/char for the cases of 1-character and 16-character NFAs, respectively. In the proposed approach, the hardware costs are 0.96 LE/char and 3.51 LE/char for the cases of 1-character and 16-character NFAs, respectively, when only matching flags are output, which resembles the results of Katashita et al. When the output stage is implemented by priority multiplexers, the hardware costs increase to 1.42 LE/char and 10.27 LE/char for the cases of 1-character and 16-character NFAs, respectively. Despite the increases in hardware cost and time delay when using priority multiplexers to determine the matching outputs, providing a corresponding matching output represented in a state number for each inspected character should be convenient in most applications.

5.2. Comparison with Other Approaches

Table V compares our result with those of previous works. The performance data of the other approaches are taken from the corresponding literature. Making a fair comparison is relatively difficult, because the hardware and software approaches significantly differ in the way that they achieve the parallelism. Nevertheless, the comparison provides insight into the status of the proposed approach. For software approaches, column *Clock* is the operating clock of the CPU or GPU, and column *Parallelism* is the number of processing cores. For hardware approaches, columns *Clock* and *Parallelism* are the clock rate and the width of the data bus, respectively. The throughput of a hardware implementation is generally derived by multiplying the clock rate by the data width.

This comparison table lists the hardware approaches in the upper rows. The approach of Pao and Wang [2012] achieves the parallelism by using three QSV (quick sampling

Table V. Comparison of Different Approaches

Description	Clock	Parallelism	Throughput
Our proposed approach	137MHz	128 bits	17.6Gbps
Multicharacter	248MHz	128 bits	31.8Gbps
NFA	263MHz	512 bits	134.7Gbps
QSV implementation	230MHz	24 bits	5.5Gbps
Parallel string-matching engine	149MHz	32 bits	4.8Gbps
Multicore CPU (IBM Cell/B.E.)	3.2GHz	8 cores	40.0Gbps
Multicore GPU (Nvidia)	1.3GHz	30 cores	15.6Gbps
Cray XMT	N/A	128 nodes	28.0Gbps
Multicore CPU (Intel Xeon)	2.26GHz	32 cores	34.0Gbps

with on-demand verification) units that runs at 230MHz and has a 24-bit data width. Consequently, its throughput is 5.5Gbps. The approach of Katashita et al. [2006a] is an implementation of multicharacter NFAs that runs at 263MHz and has a 512-bit data width and can achieve a throughput of 134.7Gbps. This table also lists the result with 128-bit data width of the work of Katashita et al. as a reference for comparison. The approach of Tripp [2006] achieves the parallelism by using multiple string-matching engines that can process four characters every clock and run at 149MHz; in addition, the achievable throughput is 4.8Gbps.

The remaining rows of the table are for software approaches. The approach of Scarpazza et al. [2008a] is implemented on an IBM Cell/B.E. processor that has eight synergistic processing elements (SPEs); the throughput of each SPE is 5Gbps and jointly is 40Gbps. The approach of Tumeo et al. [2010] was implemented on an Nvidia GPU Tesla C1060, which works at 1296MHz (shader clock) and has 30 cores; in addition, the throughput can achieve 15.6Gbps. In the work of Tumeo et al. [2012], several architectures are evaluated where this table lists only the result with the highest performance, which is evaluated on a Cray XMT with 128 processors; the resulting throughput is 28Gbps. The approach of Yang and Prasanna [2013] is implemented on a 32-core Intel Manycore Testing Lab machine based on the Intel Xeon X7560 processor, which is an 8-core “Nehalem” running at 2.26GHz; the resulting throughput is 34Gbps.

6. CONCLUSIONS

This work presents a novel AC-NFA approach, capable of avoiding the state transition explosion in a DFA approach and determining a proper matching output from multiple active states via the use of a priority multiplexer. A systematic approach based on concatenation operations and assistant transitions can derive a multicharacter AC-NFA from an AC-trie. The derived multicharacter AC-NFA can solve the alignment problem as well. The proposed architecture is also implemented on FPGA devices for evaluation. Evaluation results indicate that the throughput and the hardware cost grow nearly linearly with respect to the characters inspected in parallel. Moreover, 16-character transitions with 1,000 keywords can be implemented at a 167.36MHz clock with a throughput of approximately 21.4Gbps achieved, whereas an implementation of 16-character transitions with 2,000 keywords can operate at a clock rate of 137.34MHz and a derived throughput of approximately 17.6Gbps.

In summary, the proposed approach for multicharacter transition string matching is simple and intuitive, allowing for its easy implementation for any required number of characters to be inspected in parallel. Importantly, the proposed approach is efficient for hardware implementation, and the growth of the hardware cost is linear to the number of characters to be inspected in parallel.

ACKNOWLEDGMENTS

The authors would like to thank the National Science Council of Taiwan for financially supporting this research under Contract No. NSC 101-2221-E-002-171-MY3.

REFERENCES

- AHO, A. V. AND CORASICK, M. J. 1975. Efficient string matching: An aid to bibliographic search. *Comm. ACM* 18, 6, 333–340.
- ALICHERY, M., MUTHUPRASANNA, M., AND KUMAR, V. 2006. High speed pattern matching for network IDS/IPS. In *Proceedings of the IEEE 14th International Conference on Network Protocols*. IEEE, Los Alamitos, CA, 187–196.
- ALTERA CORPORATION. 2013. Nios II Performance Benchmarks, DS-N28162004-9.0 Data Sheet, July 2013, 101 Innovation Drive, San Jose, CA 95134.
- BLOOM, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7, 422–426.
- CHEN, C.-C. AND WANG, S. D. 2012. A multi-character transition string matching architecture based on Aho-Corasick algorithm. *Int. J. Innovative Comput. Inf. Control* 8, 12, 8367–8386.
- CLARK, C. R. AND SCHIMMEL, D. E. 2004. Scalable pattern matching for high speed networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 249–257.
- DHARMAPURIKAR, S., KRISHNAMURTHY, P., SPROULL, T., AND LOCKWOOD, J. 2003. Deep packet inspection using parallel Bloom filters. In *Proceedings of the 11th IEEE Symposium on High Performance Interconnects*. IEEE, Los Alamitos, CA, 44–51.
- DIMOPOULOS, V., PAPAETHATHIOU, I., AND PNEVMATIKATOS, D. 2007. A memory-efficient reconfigurable Aho-Corasick FSM implementation for intrusion detection systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS'07)*. IEEE, Los Alamitos, CA, 186–193.
- HERATH, D., LAKMALI, C., AND RAGEL, R. 2012. Accelerating string matching for bio-computing applications on multi-core CPUs. In *Proceedings of the 7th IEEE International Conference on Industrial and Information Systems (ICIIS'12)*. 1–6.
- HUA, N., SONG, H., AND LAKSHMAN, T. V. 2009. Variable-stride multi-pattern matching for scalable deep packet inspection. In *Proceedings of IEEE INFOCOM 2009*. IEEE, Los Alamitos, CA, 415–423.
- KATASHITA, T., MAEDA, A., TODA, K., AND YAMAGUCHI, Y. 2006a. A method of generating highly efficient string matching circuit for intrusion detection. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'06)*. IEEE, Los Alamitos, CA, 1–6.
- KATASHITA, T., MAEDA, A., TODA, K., AND YAMAGUCHI, Y. 2006b. Highly efficient string matching circuit for IDS with FPGA. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. IEEE, Los Alamitos, CA, 285–286.
- LIN, W. AND LIU, B. 2008. Pipelined parallel AC-based approach for multi-string matching. In *Proceedings of the IEEE 14th International Conference on Parallel and Distributed Systems*. IEEE, Los Alamitos, CA, 665–672.
- PAO, D., LIN, W., AND LIU, B. 2010. A memory-efficient pipelined implementation of the Aho-Corasick string-matching algorithm. *ACM Trans. Archit. Code Optim.* 7, 1–27.
- PAO, D. AND WANG, X. 2012. Multi-stride string searching for high-speed content inspection. *Comput. J.* 55, 10, 1216–1231.
- RAHMANZADEH, V. AND GHAZNAVI-GHOUSHCHI, M. B. 2009. A multi-Gb/s parallel string matching engine for intrusion detection systems. In H. Sarbazi-Azad, B. Parhami, S.-G. Miremadi, and S. Hessabi, Eds., *Advances in Computer Science and Engineering*. Springer, Berlin, 847–851.
- SALMELA, L., TARHIO, J., AND KYTOJÖKI, J. 2006. Multipattern string matching with q-grams. *ACM J. Exp. Algorithmics* 11, 1.1, 1–19.
- SCARPAZZA, D. P., VILLA, O., AND PETRINI, F. 2008a. Exact multi-pattern string matching on the Cell/B.E. Processor. In *Proceedings of the 5th Conference on Computing Frontiers (CF'08)*. ACM, New York, 33–42.
- SCARPAZZA, D. P., VILLA, O., AND PETRINI, F. 2008b. High-speed string searching against large dictionaries on the Cell/B.E. Processor. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. IEEE, Los Alamitos, CA, 1–12.
- SIDHU, R. AND PRASANNA, V. K. 2001. Fast regular expression matching using FPGAs. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, Los Alamitos, CA, 227–238.

- SUGAWARA, Y., INABA, M., AND HIRAKI, K. 2004. Over 10Gbps string matching mechanism for multi-stream packet scanning systems. In J. Becker, M. Platzner, and S. Vernalde, Eds., *Field Programmable Logic and Application*. Springer, Berlin, 484–493.
- TRIPP, G. 2006. A parallel “string matching engine” for use in high speed network intrusion detection systems. *J. Comput. Virol.* 2, 21–34.
- TUCK, N., SHERWOOD, T., CALDER, B., AND VARGHESE, G. 2004. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*. IEEE, Los Alamitos, CA, 2628–2639.
- TUMEO, A., VILLA, O., AND CHAVARRIA-MIRANDA, D. G. 2012. Aho-Corasick string matching on shared and distributed-memory parallel architectures. *IEEE Trans. Parallel Dist. Syst.* 23, 3, 436–443.
- TUMEO, A., VILLA, O., AND SCIUTO, O. 2010. Efficient pattern matching on GPUs for intrusion detection systems. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*. ACM, New York, 87–88.
- VILLA, O., CHAVARRIA-MIRANDA, D., AND MASCHHOFF, K. 2009. Input-independent, scalable and fast string matching on the Cray XMT. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. IEEE, Los Alamitos, CA, 1–12.
- YAMAGAKI, N., SIDHU, R., AND KAMIYA, S. 2008. High-speed regular expression matching engine using multi-character NFA. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications*. IEEE, Los Alamitos, CA, 131–136.
- YANG, Y.-H. E. AND PRASANNA, V. 2013. Robust and scalable string pattern matching for deep packet inspection on multi-core processors. *IEEE Trans. Parallel Dist. Sys.* 24, 11, 2283–2292.
- YANG, Y.-H. E., PRASANNA, V. K., AND JIANG, C. Q. 2010. Head-body partitioned string matching for deep packet inspection with scalable and attack-resilient performance. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*. IEEE, Washington, DC, 1–11.
- ZHA, X. AND SAHNI, S. 2008. Highly compressed Aho-Corasick automata for efficient intrusion detection. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC'08)*. IEEE, Washington, DC, 298–303.

Received January 2013; revised July 2013; accepted August 2013