Pipelined Architecture for Multi-String Matching

Derek Pao[1], Wei Lin[2,1], and Bin Liu[2]

[1]Department of Electronic Engineering, City University of Hong Kong, Hong Kong
[2]Department of Computer Science and Technology, Tsinghua University, Beijing, PRC

Abstract—We present a pipelined approach to hardware implementation of the Aho-Corasick (AC) algorithm for string matching called P-AC. By incorporating pipelined processing, the state graph is reduced to a character trie that only contains forward edges. Edge reduction in P-AC is very impressive and is guaranteed algorithmically. For a signature set with 4434 strings extracted from the Snort rule set, the memory cost of P-AC is only 21.5 bits/char. The simplicity of the pipeline control plus the availability of 2-port memories allow us to implement two pipelines sharing the set of lookup tables on the same device. By doing so, the system throughput can be doubled with little overhead. The throughput of our method is up to 8.8 Gbps when the system is implemented using 550MHz FPGA.

Index Terms—string matching, deterministic finite automaton, pipelined processing, network intrusion detection.

1 Introduction

STRING matching has been extensively studied in the past 30 years. A string Y of length n is a sequence of characters $c_1c_2\ldots c_n$. Let $\Sigma = \{Y_1, Y_2, \ldots Y_N\}$ be a finite set of strings called *keywords* or *signatures*, and let I be an arbitrary string. The string matching problem is to locate and identify all substrings of I which are signatures in Σ. There have been renewed interests in hardware-assisted high-speed string matcher prompted by the evolving applications in real-time packet processing and network intrusion detection.

Many of the proposed hardware solutions are based on the well-known Aho-Corasick (AC) algorithm [1], where the system is modeled as a deterministic finite automaton (DFA). The AC algorithm solves the string matching problem in time linearly proportional to the length of the input stream. However, the memory requirement is prohibitive in a straightforward hardware implementation. In this letter, we present a pipelined processing approach to the implementation of AC algorithm, called P-AC. The control logic of P-AC is simple and elegant. The memory cost is less than 30% of the best known AC-based methods. Our contributions are twofold. First, edge reduction in the state graph is guaranteed algorithmically in P-AC. Previously published AC-based methods are heuristic-based, such as bit-map encoding and path compression [13], bit-slice implementation [12], categorizing alphabets based on frequency count [5], and signature set partitioning [6]. Performance of these methods is sensitive to the size and statistical properties of the signature set. Second, we present a comprehensive approach to aggregate partial-matches for the processing of long signatures. We show that by utilizing the timing information of the pipeline, only a small amount of history information needs to be maintained by the system.

We shall restrict our discussion to the processing of simple strings in this letter. Extensions to handle strings defined by regular expressions will be discussed in a full paper. A brief review of related work is given in Section 2. The proposed pipelined architecture will be presented in Section 3. Section 4 is the performance evaluation and Section 5 is the conclusion.

Manuscript submitted: 17-Jan-2008. Manuscript accepted: 06-Feb-2008. Revised-manuscript received: 13-Feb-2008. Final manuscript received: 09-Apr-2008.

2 Background and Previous Work

In the basic AC automaton, the system starts from an initial state, and looks up the transition rule table using the current state and input character to determine the next state. An entry in the transition rule table is a 3-tuple E=(u=*current state*, i=*input symbol*, v=*next state*). A match-result is generated when the system reaches an output state.

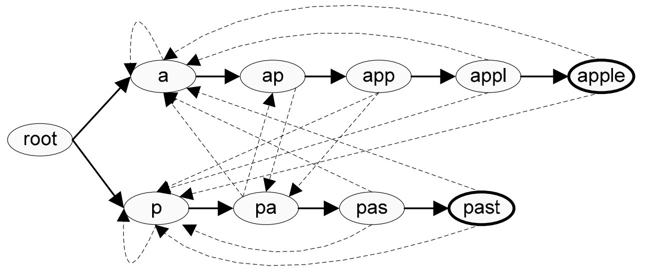


Figure 1. State graph for Σ ={apple, past}.

To facilitate discussion, we shall first introduce some terminology. Figure 1 shows the state graph for a set of two strings Σ={apple, past}. Each node in the state graph represents a distinct string value as shown in the node label. To improve readability, transitions to the root are excluded. The input symbol of a transition is equal to the last character of the string represented by the corresponding destination node. A node in the state graph can be assigned a level number according to the length of the string that it represents. The level number of the *root* node is equal to zero because its state value corresponds to the empty string. We denote a node on level 1 as L1 node, and so on. An edge E=(u, i, v) is called a *forward edge* if the level number of v is equal to 1 plus the level number of u. Forward edges are shown with solid lines in Fig. 1. The remaining edges are called *cross edges*, and they are shown in dashed lines.

Tuck et al. [13] proposed a bit-map encoding and path compression technique to reduce the memory cost of the transition rule table. Tan and Sherwood [12] proposed to use bit-split finite state machines (FSMs), where each bit-split FSM processed one bit of an input byte. The signatures are divided into groups of 16, such that the partial-match of a node can be represented by a bit-vector with 16 bits. Eight bit-split FSMs are built for each group of 16 signatures. Hence, the system contains $N/2$ FSMs. There are practical difficulties and substantial overheads in implementing a large number of FSMs in hardware. Dimopoulos et al. [5] proposed to divide the 256 alphabets into *frequent* and *infrequent* characters based on their frequency counts in the signature set. A full state graph is constructed for frequent characters, where the transition rule table for infrequent characters is implemented using CAM. Lunteren [6] observed that if the system supported prioritized tabl lookup, all the edges pointing to a node v on L1 can be replaced by a single entry (*, i, v) in the transition rule table, where * is a wildcard. Similarly, all edges pointing to the *root* are replaced by a single entry (*, *, *root*). The number of edges can be further reduced to about 1.5 edges per character by dividing the signature set into multiple groups. But the performance of the partitioning scheme depends largely on the size and statistical properties of the signature set. Alicherry et al. [2] used ternary CAM (TCAM) to implement the transition rule table. In their method, state transitions are based on multiple (typically 2 to 4) input characters. There are two advantages in using multi-character based transitions. First the size of the transition rule table is significantly reduced, and second the throughput can be increased. However, improvements in speed and memory cost are partially offset by the slower clock rate (maximum 266 MHz) and the higher cost of TCAM.

There are also proposals based on other techniques, such as hashing [3, 7, 11], Bloom filters [4, 9, 10], and CAM [11, 14]. A general drawback of these methods is that they can only handle strings of up to certain length. Long signatures are divided into multiple segments. Complex auxiliary data structures [14] and/or hardwired aggregation logic [11] are required to process the partial-matches. The aggregation methods in some of the above studies are over-simplified. For example, in [7, 11] a long signature can only be divided into 2 segments; in [3] it does not consider the case where a segment can be the first segment of one signature, and the middle or last segment of some other signature(s) at the same time.

3 Pipelined Architecture

3.1 Basic Idea

First we shall consider signatures that are case insensitive. The processing of case sensitive signatures will be discussed in Section 3.3. In the basic AC automaton, the longest matching substring for the current input is represented by the value of the current state. Suppose the input stream is "appastxyz". The DFA of Fig. 1 will visit nodes <a>, <ap>, and <app> in the first 3 cycles. In the 4th cycle, the input character 'a' does not match the input symbol of any forward edges originating from <app>. Hence, the DFA will follow a cross edge and change to state <pa>.

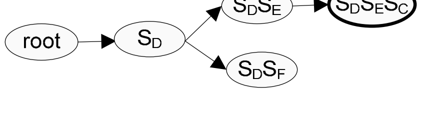
In the proposed pipelined architecture, a new thread is initiated to trace along the automaton starting from the current input character in each cycle. By doing so, the system only needs to store the forward edges in the transition rule table. There is zero or one active state in each pipeline stage. The active state in stage i represents a matching substring of length i that ends at the last input character. In each cycle, the input character is sent to all pipeline stages. The local transition rule table of stage i stores the forward edges originating from a node on L_i to a destination node on L_{i+1}. Stage i will simply look up its local table using its local active state and the input character, and pass the result to the next stage. Table 1 shows the active states of the first 8 cycles for the sample input stream "appastxyz". Note that the active state of stage 0 is always equal to <root>. When a thread cannot proceed further with the current input, it is terminated. In cycle 4, the thread of stage 3 is terminated since the input character 'a' does not match any of the forward edges originating from node <app>. But another thread will pick up the longest matching substring "pa" without involving cross edges. A match-result for the string "past" will be generated in cycle 7.

Table 1. Active states for the sample input "appastxyz".

		Active state of pipeline stages					
cycle	input	stage 0	stage 1	stage 2	stage 3	stage 4	stage 5
1	'a'	<root>					
2	'p'	<root>	<a>				
3	'p'	<root>	<p>	<ap>			
4	'a'	<root>	<p>		<app>		
5	's'	<root>	<a>	<pa>			
6	't'	<root>			<pas>		
7	'x'	<root>				**<past>**	
8	'y'	<root>					

3.2 Processing of Long Signatures

In general, signatures can be longer than the hardware pipeline. Some refinements to the pipeline are necessary in order to handle long strings. Assume the pipeline has k+1 stages numbered from 0 to k, where the last stage is only used to buffer the search result of stage k –1. Strings longer than k characters are divided into segments of length k, except for the last segment whose length can be less than k. Consider a signature set with 4 strings Σ = {and, test, instructions, instrument}. Assume k is equal to 4, there are 7 segmented strings {nt, and, ions, inst, ruct, rume, test}. Each segment is assigned a unique segment ID and a Boolean flag L. L is equal to 1 if the segment is part of a long string. Detection of a segment with L equals to 1 represents a partial-match of a long signature. Partial-matches are aggregated by a DFA constructed using the ID of full-length segments as shown in Fig. 2. The method of [6] is applied to reduce the number of cross edges that point to the root and L1 nodes.


$$\begin{tikzcd}
 & & S_DS_E \arrow[r] & S_DS_ES_C \\
\text{root} \arrow[r] & S_D \arrow[ur] \arrow[dr] & & \\
 & & S_DS_F &
\end{tikzcd}$$

Segment	Seg. ID	L
nt	S_A	1
and	S_B	0
ions	S_C	1
inst	S_D	1
ruct	S_E	1
rume	S_F	1
test	S_G	0

Figure 2. DFA for aggregation of partial-matches.

Organization of the pipeline system is depicted in Fig. 3. The local transition tables (LT) can be implemented using hardware hashing. Match-results for strings with no more than *k* characters are generated by the pipeline unit directly (the output paths of the pipeline unit are not shown in the diagram to enhance readability). For long strings with more than *k* characters, the match-result will be generated by the aggregation unit (AU).

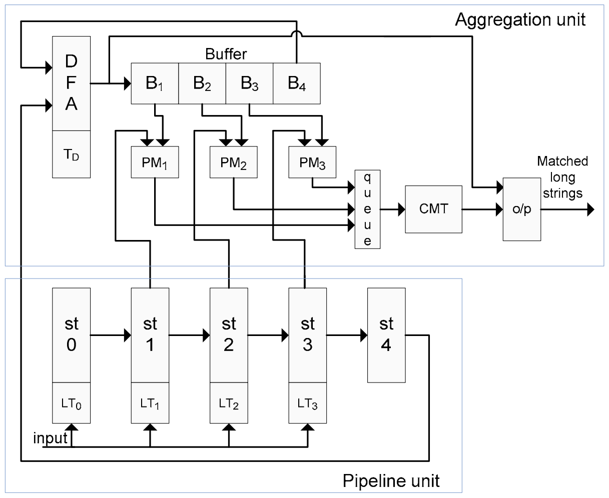


Figure 3. Organization of the pipeline system for *k*=4.

When the pipeline unit detects a full-length segment, the corresponding segment ID will be passed to the DFA unit of the AU if the L bit of the segment is equal to 1. The AU maintains a buffer (shift register) that stores the previously aggregated state values. Contents of the buffer are shifted to the right by one slot in each cycle. When a new segment ID is received, the state value corresponding to the preceding segment detected at *k* cycles ago, if any, will be stored in the last buffer slot B_k. The DFA unit uses the state value stored in B_k and the new segment ID to look up the transition rule table T_D to determine the next state. If there is no matching rule found, the <root> is taken as the default next state. The lookup result will be shifted into the buffer at the start of the next cycle. If the pipeline unit detects the last segment of a long string, where the segment length is less than *k*, the segment ID will be passed to the corresponding partial-match unit (PM_i) of the AU. If the state value stored in buffer B_i is not equal to <root>, PM_i sends a lookup request to the conditional match table (CMT); otherwise PM_i will simply discard the segment ID. Tables 2 and 3 show the transition rule table for the DFA and the CMT.

Let's consider an example with the input data stream equals to "test instrument …". In cycle 5 the last stage of the pipeline detects the string "test". In cycle 10, the pipeline unit detects the segment "inst" and passes its segment ID S_D to the AU. The state value <S_D> will then be shifted into B_1 at the start of the next cycle. In cycle 14, the pipeline detects the segment "rume" and sends the segment ID S_F to the AU. At this moment, the state value <S_D> would have been shifted down to buffer slot B_4. The DFA unit looks up table T_D using the state value <S_D> and the segment ID S_F to find the next state <S_DS_F>. In cycle 16, the segment "nt" is detected at stage 2 of the pipeline unit. At this moment, the state value <S_DS_F> is stored in buffer slot B_2. On receiving the segment ID S_A for "nt", PM_2 sends a lookup request to the CMT and finds the matched string "instrument".

Table 2. Transition rule table for the DFA.

Current state	Input	Next state	Matched string
*	S_D	S_D	
S_D	S_E	S_DS_E	
S_D	S_F	S_DS_F	
S_DS_E	S_C	$S_DS_ES_C$	"instructions"

Table 3. Conditional match table.

State	Input	Matched string
S_DS_F	S_A	"instrument"

In principle the pipeline unit may find a matched segment in each stage at each cycle. As a result, concurrent access to the CMT may be required. When there are multiple access requests, the requests are queued and served in FIFO order. However, concurrent access to the CMT are unlikely to happen except when the signature set contains strings of different lengths that are composed of sequences of the same character or some short repetitive patterns, e.g. "aaaaa", "aaaaaa", "aaaaaaa", and the input stream contains a long sequence of the letter 'a'. To resolve this problem, each PM caches the last lookup request to the CMT. If the new lookup request is the same as the cached request, the lookup result will be provided by the cache directly.

3.3 Handling Case Sensitive Signatures

We adopt the match and verify strategy to handle case sensitivity. All the signatures and the input data stream are converted to lower case letters. Case sensitivity is verified after a matching string has been found. In the ASCII code, the 5-th bit for a lower case letter is equal to 1, where the 5-th bit for an upper case letter is equal to 0. We can modify the hardware in such a way that the original value of the 5-th bit of each byte of input data is extracted and stored in a bit-vector. There is a control bit associated with the match-result that indicates whether the matched string is case sensitive or not. If the matched string is case sensitive, the system compares the extracted bit-vector with the corresponding bit-vector of the signature to confirm the match result.

4 Performance Evaluation

We extract 4434 distinct signatures from the Snort rule set [8]. The signatures are converted into lower case letters. The average signature length is about 19 characters, and the total character count is 84015. About 98% of the signatures have no more than 64 characters.

Our evaluation indicates that better memory efficiency is obtained when the segment length is in the range of 4 to

6. For *k*=4, the number of entries in the lookup tables for the pipeline unit, the DFA unit, and CMT are 22031, 19289, and 3076, respectively. The width of the state ID for the pipeline unit and the DFA unit are 15 bits and 14 bits, respectively. The amount of memory space occupied by the entries of the lookup tables is 220KB. A performance comparison with 4 other AC-based methods is given in Table 4. The memory cost of P-AC, BFPM [6], and the TCAM-based method of [2] are evaluated using the same signature set. When the given signature set is divided into eight groups in BPFM, the number of edges per character is about 2.5. With 36-bit lookup table entries, the memory cost for BFPM is about 90 bits/char. In [2], we assume state transitions are based on 2 input characters per cycle. The hardware cost of bit-split FSM [12] and split-AC [5] are quoted from [5]. Both bit-split FSM and split-AC require complex control logic. For the split-AC method, there can be tradeoff between memory cost and control logic. The chip area for a FPGA logic cell is approximately the same as 12 bytes of memory [11]. Hence, the actual hardware cost for the 2 sets of implementation parameters shown in Table 4 are more or less the same.

Table 4. Comparison with AC-based methods

Method	Signature Set (chars)	Memory (per char)	Control logic	Speed (char/cycle)
bit-split FSM	12.8K	186 bits	complex	1
split-AC	24K	65 bits	60061 LUTs	1 (slower clk)
		189 bits	12341 LUTs	
BFPM (8 groups)	84K	90 bits	simple	1
TCAM (2 char/cycle)	84K	56 bits TCAM + 27 bits SRAM	simple	2 to 4 (max 266MHz clk)
P-AC	84K	21.5 bits	simple	2 (with dual pipelines)

The control logic for P-AC is simple. Pipelining of output results from one stage to the next stage can be realized by simple clocked-register. There are *k*+3 memories in the system (*k* memories for the pipeline unit, 2 memories for the DFA units and 1 memory for the CMT). The DFA unit requires 2 memories because there are 2 types of edges, one type with the current state equals to don't care and the second type with current state not equal to don't care. LT_0 is a 256-entry table indexed by the input character. A hash function generator and a comparator will be required for each of the remaining lookup tables. With *k*=4, the length of the hash key is up to 29 bits for the signature set used in this study. We estimate that the control logic for the lookup tables can be implemented using about 1000 LUTs.

Two-port memories are available in commercial FPGAs. Hence, two pipelines can be built on the same device sharing the set of lookup tables. By doing so, the system throughput can be doubled with little overhead. Using the Xilinx Virtex-5 FPGA that operates at 550 MHz, the throughput of P-AC is up to 8.8 Gbps. Higher throughput can be possible by using ASIC implementation or faster FPGAs from Achronix Semicondutor that can operate at 1.6 to 2.2 GHz.

5 Conclusion

A pipelined approach for hardware implementation of the Aho-Corasick algorithm called P-AC is presented. The system maintains multiple threads that traverse the automaton concurrently. Only forward edges of the state graph need to be stored in the lookup tables. In contrast to previously published heuristic-based methods, edge reduction in P-AC is guaranteed algorithmically. This is an advantage that ensures scalability of the method in handling fast expanding signature sets of network intrusion detection systems. For a signature set with 4434 strings extracted from Snort, the memory cost of P-AC is as low as 21.5 bits/char, which is less than 30% of the best known AC-based methods. Simplicity and elegance of the pipeline control allows the system to operate at high clock rate. In addition, if 2-port memories are available, we can implement 2 pipelines on the same device that share the lookup tables. As a result, the system throughput can be doubled with little overhead. The throughput of P-AC is up to 8.8 Gbps when the system is implemented using 550 MHz FPGA.

Acknowledgment

This work is partially supported by NSFC (No. 60573121, 60625201), and China-973 program (No. 2007CB310701). The authors like to thank Dr. H. F. Li, Dr. Clement Lam, Dr. Hongbin Lu, editors of CAL, and the anonymous reviewers for their constructive comments.

References

[1] A. V. Aho, M. J. Corasick, "Efficient string matching: an aid to bibliographic search", Comm. of the ACM, Vol. 18, No. 6, pp. 333-340, 1975.
[2] M. Alicherry, M. Muthuprasanna, V. Kumar, "High speed matching for network IDS/IPS", IEEE ICNP, pp. 187-196, 2006.
[3] Y. H. Cho, W. H. Mangione-Smith, "A pattern matching co-processor for network security", IEEE DAC, pp. 234-239, 2005.
[4] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, J. W. Lockwood, "Deep packet inspection using parallel Bloom filters", IEEE Micro, pp. 52-61, Jan.-Feb. 2004.
[5] V. Dimopoulos, I. Papaefstathiou, D. Pnevmatikatos, "A memory-efficient reconfigurable Aho-Corasick FSM implementation for intrusion detection systems", IEEE IC-SAMOS, pp. 186-193, 2007.
[6] J. van Lunteren, "High-performance pattern-matching for intrusion detection", IEEE INFOCOM, pp. 1-13, 2006.
[7] G. Papadopoulos, D. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching ", IEEE Int. Conf. on Field Programmable Logic and Applications, pp. 39-44, 2005.
[8] SNORT network intrusion detection system, http://www.snort.org
[9] H. Song, T. Sproull, M. Attig, and J. Lockwood, "SNORT offloader: a reconfigurable hardware NIDS filter", IEEE Int. Conf. on Field Programmable Logic and Applications, pp. 493-498, 2005.
[10] H. Song, J. W. Lockwood, "Multi-pattern signature matching for hardware network intrusion detection systems", IEEE GLOBECOM, pp. 1686-1690, 2005.
[11] I. Sourdis, D. Pnevmatikatos, S. Vassiliadis, "Scalable multigigabit pattern matching for packet inspection", IEEE Trans. on VLSI Systems, Vol. 16, Issue 2, pp. 156-166, Feb. 2008.
[12] L. Tan, T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention", IEEE ISCA, pp. 112-122, 2005.
[13] N. Tuck, T. Sherwood, B. Calder, G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection", IEEE INFOCOM, pp. 2628-2639, 2004.
[14] F. Yu, R. H. Katz, T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM", IEEE ICNP, pp. 174-183, 2004.