A Memory-Efficient Pipelined Implementation of the Aho-Corasick String-Matching Algorithm

DEREK PAO
City University of Hong Kong
and
WEI LIN, and BIN LIU
Tsinghua University

With rapid advancement in Internet technology and usages, some emerging applications in data communications and network security require matching of huge volume of data against large signature sets with thousands of strings in real time. In this article, we present a memory-efficient hardware implementation of the well-known Aho-Corasick (AC) string-matching algorithm using a pipelining approach called P-AC. An attractive feature of the AC algorithm is that it can solve the string-matching problem in time linearly proportional to the length of the input stream, and the computation time is independent of the number of strings in the signature set. A major disadvantage of the AC algorithm is the high memory cost required to store the transition rules of the underlying deterministic finite automaton. By incorporating pipelined processing, the state graph is reduced to a character trie that only contains forward edges. Together with an intelligent implementation of look-up tables, the memory cost of P-AC is only about 18 bits per character for a signature set containing 6,166 strings extracted from Snort. The control structure of P-AC is simple and elegant. The cost of the control logic is very low. With the availability of dual-port memories in FPGA devices, we can double the system throughput by duplicating the control logic such that the system can process two data streams concurrently. Since our method is memory-based, incremental changes to the signature set can be accommodated by updating the look-up tables without reconfiguring the FPGA circuitry.

This work was supported by the Hong Kong University Grant Council GRF research grant no. 9041500, the NSF China (60625201, 60873250) and 973 project (2007CB310701), and Tsinghua University Initiative Scientific Research Program.
This work is an extension of a previously published short paper by the same authors, "Pipelined architecture for multi-string matching," IEEE Computer Architecture Letters, 7, 2, 33-36.
Authors' addresses: D. Pao, Department of Electronic Engineering, City University of Hong Kong, Hong Kong. W. Lin and B. Liu, Department of Computer Science and Technology, Tsinghua University, Beijing, PRC; email:d.pao@cityu.edu.hk.

DOI 10.1145/1839667.1839672 http://doi.acm.org/10.1145/1839667.1839672

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; I.5.5 [**Pattern Recognition**]: Implementation

General Terms: Algorithms, Design, Performance, Security

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

ACM Reference Format:

Pao, D., Lin, W., and Liu, B. 2010. A memory-efficient pipelined implementation of the aho-corasick string-matching algorithm. ACM Trans. Architec. Code Optim. 7, 2, Article 10 (September 2010), 27 pages.
DOI = 10.1145/1839667.1839672 http://doi.acm.org/10.1145/1839667.1839672

1. INTRODUCTION

String matching has been studied extensively in the past 30 years. A string Y of length s is a sequence of characters $c_1c_2 \ldots c_s$. Let $\Gamma = \{Y_1, Y_2, \ldots Y_n\}$ be a finite set of strings called *keywords* or *signatures*, and let I be an arbitrary string (the input stream to the string-matching engine). The string-matching problem is to locate and identify all the substrings of I which are signatures in Γ. Pioneering string-matching algorithms include the methods of Aho and Corasick [1975], Knuth et al. [1977], and Wu and Manber [1992]. String matching has found widespread applications in text editors, word processing, bibliographic search, and pattern recognition. With rapid advancements in Internet technology and usage, some emerging applications in data communications and network security require the matching of a huge volume of data against large signature sets with thousands of strings. For example, the Snort intrusion detection system contains more than 6,000 strings in its rule set. These new applications sparkle renewed interests in the design of high-speed string-matching engines using FPGA or ASIC.

Sophisticated hardware implementation techniques have been reported in the literature. Baker and Prasanna [2005] presented a hardware implementation of the Knuth-Morris-Pratt (KMP) algorithm [Knuth et al. 1977]. Since the KMP algorithm is designed to match the input stream against a single string, one matching unit is required per string, and the hardware system is composed of a linear array of matching units. The methods of Clark and Schimmel [2004], Baker and Prasanna [2006], and Sourdis and Pnevmatikatos [2004] are based on pre-decoded characters with hardwired logic circuits. The system is optimized with respect to the given set of signatures and the characteristics of the FPGA devices. The FPGA is reconfigured when there are changes to the signature set. However, the long latency required in offline generation of the optimized hardwired circuits is considered as a major disadvantage in a network intrusion detection system that demands fast responses to hostile conditions. Other implementation techniques based on hashing [Cho and Mangione-Smith 2005; Papadopoulos and Pnevmatikatos 2005; Sourdis et al. 2008]; Bloom filters [Dharmapurikar et al. 2005; Dharmapurikar and Lockwood 2006; Lu et al. 2006]; and ternary content addressable memory (TCAM) [Yu et al. 2004] have also been proposed.

Hardware-assisted string-matching engines based on the AC algorithm have gained much popularity in recent years. An attractive feature of the AC algorithm is that it can solve the string-matching problem in time linearly proportional to the length of the input stream. All the strings in the signature set are integrated into a single deterministic finite automaton (DFA) such that the processing time is independent of the size of the signature set. A major disadvantage of the AC algorithm is the high memory cost required to store the transition rules of the DFA. Consider a signature set with n strings and an average length L. The maximum number of states in the AC state graph is equal to $n \times L$. There are 256 transition edges from each state. If $n = 5{,}000$ and $L = 20$, there are up to 25.6 million transition edges in the state graph. Implementing such a large transition rule table in embedded devices is not feasible.

Space–time trade-off in string matching using DFA and nondeterministic finite automaton (NFA) has been studied extensively. The advantage of DFA is that the processing time is deterministic. To process one input character, the automaton only needs to perform one table look-up and makes one state transition. The disadvantage of DFA is the high memory cost. On the other hand, NFA is well known for its superior space efficiency over DFA. The state graph maintained by the NFA is reduced to a tree. The disadvantage of using NFA is that the automaton can make multiple state transitions for each input character. If the NFA is implemented using a memory-based approach, multiple table look-up operations are required for each input character. To avoid the speed penalty, researchers map the finite automata to hardwired logic circuits. However, hardwired circuits lack flexibility. Whenever the signature set is updated, the circuits need to be recompiled.

Previous studies had either focus on (i) reducing the memory space in the DFA using various techniques, such as compression, encoding, and transition edge reduction, or (ii) simplification of the logic for implementing NFA with hardwired circuits. In this article, we present a memory-based pipelined processing approach that possesses the advantages of DFA and NFA such that it can achieve the computation efficiency of DFA and space efficiency of NFA. The work presented in this article is an extension of our previously proposed pipelined architecture for string matching [Pao et al. 2008]. We observe a very subtle property of the NFA for the string-matching problem. Each state in the state graph represents a distinct substring of a pattern in the signature set. If we assign a level number to each node in the state graph according to the length of the substring represented by the corresponding state, we will find that there can be, at most, one active state on each level at any given time. We show that by making use of this property, the pipelined architecture can achieve deterministic processing speed and state graph reduction at the same time.

Conceptually, each pipeline stage will make one state transition for each input character. In order to achieve a processing speed of one character per cycle, each table look-up operation should be completed in one cycle. The conventional approach to implement large look-up tables (LUTs) is based on hashing. However, collisions are not avoidable for dynamic signature set. When collisions

occur, a table look-up operation may require multiple memory accesses. As a result, the throughput of the system will be degraded. In this article, we present a space-efficient method to implement the required LUTs such that a look-up operation is guaranteed to be completed in one cycle.

The performance of the pipelined architecture is evaluated using a signature set with 6,166 static strings extracted from the Snort database downloaded in April 2008. The normalized storage cost of our method is only 18 bits per character (bits/char) in the signature set. Moreover, the control logic of the proposed method is very simple. If multiport memory modules are available, we can duplicate the control logic and implement multiple pipelines that share one copy of the LUTs. Typical FPGAs are equipped with dual-port memories. Hence, the system can process two independent input streams concurrently and double the throughput with very little overhead.

The organization of this article is as follows. In Section 2, we define some terminology and give a formal characterization of the AC automaton. We identify the major issues in hardware implementation and give a detailed review of related work. The proposed pipeline system is presented in Section 3. We discuss the motivation and explain why the proposed pipelined processing approach can guarantee the elimination of backward transitions in the DFA. In Section 4, we present how to implement the LUTs using an approach called *direct indexing plus bit-selection* (DIBS). The success of DIBS depends on an intelligent strategy in assigning numeric identifiers (IDs) to states in the DFAs, and string segments. In Section 5, we present performance evaluation and comparison with previously published AC-based methods. Section 6 concludes the article.

2. CHARACTERIZATION OF THE AC AUTOMATON AND RELATED WORK

2.1 The AC Algorithm

In the basic AC algorithm, the system is modeled as a DFA. Let $M(\Gamma) = (Q, \Sigma, q_0, \delta, F)$ denote the AC automaton for a signature set Γ, where Q is the set of states, Σ is the set of alphabets, q_0 is the initial state, δ is the transition function, and $F \subseteq Q$ is the set of output states. The transition function δ is a mapping $Q \times \Sigma \rightarrow Q$. The automaton $M(\Gamma)$ can be visualized as a state graph $G = (Q, E)$, where Q is the set of nodes and E is the set of edges. The set of edges E is defined by the transition function δ, that is, $E = \{(u, x, v) | u \in Q \wedge x \in \Sigma \wedge v = \delta(u, x)\}$, where u is the current state, x is the input symbol, and v is the next state. The system starts from the initial state q_0. It makes a state transition in each cycle based on the current state and the input character. A match-result is generated when the system reaches an output state. In this article, the pair of terms *state* and *node*, and *edge* and *transition* are used interchangeably.

Figure 1 shows the AC state graph for a set of two strings $\Gamma = \{\text{apple, past}\}$. Each node u in the state graph represents a distinct string value U as shown in the node label, where U is a prefix of some string $Y \in \Gamma$. The initial state q_0 corresponds to the empty string. To improve readability, backward transitions to q_0 and the input symbols of the transitions are not shown. The input symbol

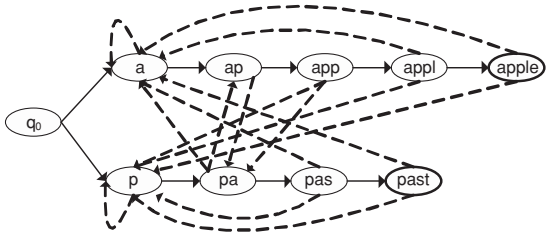

Fig. 1. AC state graph for $\Gamma = \{\text{apple, past}\}$. Backward transitions to q_0 are not shown.

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 q_0 is equal to 0 because its state value corresponds to the empty string. Let N_i denote the set of nodes on level i of the state graph. An edge $e = (u, x, v)$ is called a *forward edge* if $u \in N_i$ and $v \in N_{i+1}$. Forward edges are shown in solid lines in Figure 1. The remaining edges are called *cross-edges*, and they are shown in dashed lines. The set of cross-edges can be divided into two subgroups, namely the *failure* edges and *nonfailure* edges. A cross-edge $e = (u, x, v)$ is a nonfailure edge if $v \neq q_0$; otherwise, e is a failure edge. Let E_f, E_{cn}, and E_{cf} denote the set of forward edges, nonfailure cross edges, and failure edges, respectively. We have $E = E_f \cup E_{cn} \cup E_{cf}$.

We characterize the three types of edges defined earlier in the text. Let u and v be two distinct states in Q, and U and V denote the strings represented by u and v, respectively. A forward edge $e = (u, x, v) \in E_f$ satisfies the following properties.

(i) U and V are prefixes of some string(s) in Γ.
(ii) $u \in N_i$ and $v \in N_{i+1}$ for $i \geq 0$.
(iii) $U \cdot x = V$, where $\cdot$ is the concatenation operation.

A nonfailure cross edge e= $(u, x, v) \in E_{cn}$ satisfies the following properties.

(i) $u \in N_i$, $v \in N_j$ and $i \geq j > 0$.
(ii) There does not exist any forward edge e= $(u, x, w) \in E_f$ for some $w \in Q$.
(iii) There exists a substring y (y can be the empty string) such that y is a suffix of U and $y \cdot x = V$.
(iv) There does not exist another suffix z of U such that the length of z is longer than the length of y, and $z \cdot x = V$' for some node v' $\in Q$.

The set of alphabets is divided into three disjoint partitions with respect to a node u. Let $\Sigma_f(u) = \{x | x \in \Sigma \wedge \exists e = (u, x, v) \in E_f\}$, $\Sigma_{cn}(u) = \{x | x \in \Sigma \wedge \exists e = (u, x, v) \in E_{cn}\}$, and $\Sigma_{cf}(u) = \Sigma - \Sigma_f(u) - \Sigma_{cn}(u)$. For each $x \in \Sigma_{cf}(u)$, there exists a failure edge $e = (u, x, q_0) \in E_{cf}$.

Let n be the number of strings in Γ, L be the average string length, and Σ be the set of 8-bit ASCII characters. There are $256 \times \beta \times n \times L$ edges in $M(\Gamma)$, where the value of β is around 0.7 to 0.8. For $N = 5{,}000$ and $L = 20$, there can be more than 20 million edges in the AC state graph. The storage cost of a

straightforward implementation of the AC algorithm is up to 100MB, that is, 1KB per character. Such a high memory cost is prohibitive in ASIC or FPGA implementation.

Memory cost reduction is the major concern in the studies of hardware implementation of AC algorithm. There are proposals based on LUT compression, encoding, bit-slice implementation, rule set partitioning, and state graph reduction. In general, less than 1% of the transition edges in the AC state graph are forward edges. Hence, if we can eliminate the cross edges in the state graph, very substantial savings in memory cost can be achieved. In Section 2.2, we give an overview of previous approaches for state graph reduction, and other approaches are discussed in Section 2.3.

2.2 Related Work on State Graph Reduction

The Snort rule set contains several thousand signatures. Aldwairi et al. [2005] proposed to divide the signature set into multiple groups according to the IP addresses, protocols, and port numbers in the Snort rule header. The grouping of strings may not be disjoint, since many of the Snort rule headers contain wildcard fields. A separate DFA is built for each group of signatures. Some reduction in memory space can be achieved using this approach. For a signature set with 1,542 strings, the memory cost is 3.1MB, that is, about 100 bytes per character in the signature set.

A breakthrough in edge reduction method can be found in Lunteren [2006]. Recall that in a state transition $e = (u, x, v)$ the substring represented by the destination node v is $V = U \cdot x$. If $v \in N_1$, then U is the empty string. In this case, the state value of v is independent of the originating state u. If $v \in N_j$ for $j > 1$, U is not equal to the empty string. This means that the state value of v depends on the state value of u. Observing this property, all transitions to a node $v \in N_1$ can be replaced by a single entry (*, x, v) in the transition rule table, where * represents the wildcard. Similarly, all failure edges can be replaced by a default transition (*, *, q_0). The default transition (*, *, q_0) is assigned priority 0, transitions of type (*, x, v) with $v \in N_1$ are assigned priority 1, and transitions of type (u, x, v) with $u \neq *$ and $v \in N_j$ for $j \geq 2$ are assigned priority 2. When looking up the transition rule table with current state u and input character x, up to three matching transition rules can be found. The matching transition rule with the highest priority will be applied. In the AC state graph for a typical character-based signature set, the number of failure edges and cross edges pointing to nodes in N_1 account for about 92% of all edges. Lunteren also proposed a heuristic scheme to divide the signature set into multiple disjoint groups and a separate automaton is constructed for each group of signatures. The total number of edges can be further reduced to about 1.5 edges per character for a signature set with about 2,000 strings, where an edge requires 36 bits of storage. The performance of the partition scheme is, however, sensitive to the insertion order, size, and statistical property of the signature set.

Alicherry et al. [2006] proposed an edge reduction strategy similar to the idea of Lunteren at more or less the same time. They used TCAM to implement

the transition rule table. TCAM has been widely used in Internet router to implement various table look-up and address translation operations, such as IP address look-up and packet classification [Pao et al. 2006; Zheng et al. 2006]. TCAM supports prioritized table look-up operation and can handle case-sensitive and case-insensitive strings efficiently by setting the TCAM bit corresponding to the 5th bit of the input ASCII character to*. Alicherry et al. further pointed out that if the state ID of the destination node w of a transition rule (u, x, w) is suffixed by the input character x, then all transition edges pointing to a node $v \in N_2$ can be replaced by a single entry in the LUT $(*x, y, v)$ where the string represented by v is $V =$ "xy". Thus, the size of the LUT can be reduced to less than 3% of the original AC state graph. In principle, this approach can be extended to nodes on level 3 and so on, but the field length of the state ID will be increased proportionally.

Alicherry et al. [2006] also showed that instead of using the previously described state ID assignment strategy, edge reduction can also be achieved by using multicharacter-based state transitions. Typically, the system will make a state transition based on two or three input characters. Assume the system processes three characters per cycle. A signature is divided into chunks (segments) of three characters. An alphabet in a state transition is replaced by a chunk of three characters. Hence, the effective number of alphabets in the AC automaton is substantially increased, and the effective length (in terms of the number of segments) of a signature is reduced by a factor of 3. In addition to the state graph reduction, the throughput can be increased. A general problem with multicharacter-based state transition is the alignment problem. When the system reads in a chunk of three characters, it may not be aligned with the corresponding segment of the signature. One way to resolve the alignment problem is to introduce additional shallow states in the state graph. In general, the number of shallow states is proportional to the number of signatures. Another problem is the handling of signatures whose length are not multiple of 3. This problem can be resolved by the match-and-verify approach. A signature is truncated to length that is multiple of 3. When a match-result is reported, a postmatching module will verify the last one or two characters of the original signature to confirm the match-result.

The recent paper by Song et al. [2008] falls into the same framework that aims at eliminating edges that point to nodes on levels 0 to 2. Song's method, called *cached DFA* (CDFA), maintains two active states, namely the conventional current state u and a *cached* state c. The cached state is the destination state v obtained by following a transition (q_0, y, v) with the last input character y. In each cycle, the system looks up the transition rule table using the current state u and the cached state c. It may find two matching transition rules $e_1 = (u, x, v_1)$ and $e_2 = (c, x, v_2)$ for the given input character x. Transition rule e_1 is given higher priority over e_2. If no matching transition rule is found, then q_0 is taken as the default next state. By using this approach, all failure edges and cross edges pointing to nodes on levels 1 and 2 can be removed from the transition rule table. Song et al. also presented an innovative way to organize the transition rule table. Each node in the AC state graph is assigned a state ID and a color code. The transition rule table is organized as three LUTs called

TRM-0, TRM-1, and ITT. TRM-0 is used to determine the next value of the cached state. The state ID is used as the address to access the TRM-1, and the color code is used to access the ITT. Each entry in the ITT has 256 columns, indexed by the input character. A state transition involves two steps. In the first step, the system uses the color code of the current state and the input character x to look-up the ITT to obtain a potential next state v. In the second step, the system looks up TRM-1 using the state value v. Each entry in TRM-1 stores the ordered pair (x, *color code*), where x is the input character of the transition edges that point to node v. The next state value v is accepted if the input character x matches the value stored in the corresponding entry in TRM-1. Two nodes in the state graph can share a color code if their possible transitions are conflict-free. A large signature set is divided into multiple disjoint subsets of smaller sizes so as to get a better result in ITT optimization. Each entry in the ITT occupies 512 bytes or more. Hence, the performance is very sensitive to the statistical property of the signature set. Song's method requires two memory accesses to process one input character. The system can be sped up by pipelining and interleaving the processing of two input streams such that an overall throughput of one character per cycle can be maintained.

2.3 Other Approaches to Reduce the Transition Rule Table

Tuck et al. [2004] proposed a bit-map encoding and path compression technique to reduce the memory cost of the transition rule table. A fundamental limitation of Tuck's approach is that if the number of transitions in the state graph remains unchanged, the achievable reduction of memory cost is limited. Tan et al. [2006] proposed to use bit-split finite state machines (FSMs), where each bit-split FSM processed 1 bit of an input byte. Each state in a bit-split FSM can have only two possible transitions. Hence, a transition rule can be stored in a very compact format. 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. [2007] 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. There can be trade-off in the amount of control logic and memory storage in this method depending on the threshold used to classify frequent alphabets. Generally speaking, the complexity of the control logic of this method is relatively high.

3. PIPELINED IMPLEMENTATION OF THE AC ALGORITHM

State graph reduction is the most effective approach to reduce the hardware implementation cost. Previously published methods can mostly remove cross edges pointing to q_0 and nodes on levels 1 and 2. The methods of Lunteren and Song rely heavily on partitioning the signature set into multiple subsets

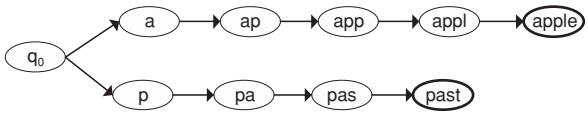

Fig. 2. Reduced state graph maintained by the pipeline system.

in order to get better performance. A separate set of LUTs are required for each partition. Hence, the amount of control logic goes up with the number of partitions.

In this section, we present a pipelined processing approach called P-AC that can remove all cross edges in the AC state graph. The AC automaton is essentially a DFA. There can be only one active state in the system at any one time. The system uses distinct states to represent the longest matching substring found so far when processing the input stream one character at a time. As a result, there are a large number of cross edges in the automaton. On the other hand, if the system is modeled as a NFA allowing multiple active states that represent matching substrings of various length, then all the cross-edges in the automaton can be removed. There is one very important property of the NFA for static strings, that is, there can be, at most, one active state on each level of the state graph. The proposed pipeline architecture exploits this characteristic to achieve edge reduction. The reduced automaton maintained by the pipeline system for the signature set Γ is $M_p(\Gamma) = (Q,\Sigma,q_0,\delta_p,F)$, where $\delta_p(u, x) = v$ if $v = \delta(u, x) \wedge u \in N_i \wedge v \in N_{i+1}$; otherwise, $\delta_p(u, x) = \phi$. The set of edges corresponding to the reduced transition function δ_p is $E_p = \{(u, x, v)|v = \delta_p(u, x) \wedge v \neq \phi\}$. Hence, the state graph maintained by the pipeline is reduced to a tree that only contains forward edges, as shown in Figure 2.

In the proposed pipelined architecture, a new thread is initiated to trace along the automaton M_p, starting from the current input character in each cycle. There is zero or one active state maintained 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 LT_i of stage i stores the forward edges originating from a node belong to N_i to a destination node in N_{i+1}, that is, $\mathrm{LT}_i = \{e = (u, x, v)|e \in E_f \wedge u \in N_i \wedge v \in N_{i+1}\}$. In each cycle, stage i looks up its local table using its local active state and the input character, and passes the result to the next stage. Table I shows the active states of the pipeline in the first eight cycles when processing the sample input stream “appastxyz”. Note that the active state of stage 0 is always equal to q_0. 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 of stage 1 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.

Although the motivation of the pipelined implementation is based on the concept of NFA, the hardware pipeline is a deterministic system. The overall system state (state of the pipeline as a whole) is a vector $(s_0, s_1, \ldots, s_k)$, where s_i is the state of the i-th pipeline stage. For an input character x, there will be one

Table I. Active States for the Sample Input "appastxyz"

		Active State of the Pipeline Stages					
Cycle	Input	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
1	'a'	<q_0>					
2	'p'	<q_0>	<a>				
3	'p'	<q_0>	<p>	<ap>			
4	'a'	<q_0>	<p>		<app>		
5	's'	<q_0>	<a>	<pa>			
6	't'	<q_0>			<pas>		
7	'x'	<q_0>				**<past>**	
8	'y'	<q_0>					

and only one next system state, since the state transition of individual pipeline stages are deterministic. One may say that the pipeline system possesses the dual property of DFA and NFA.

3.1 Processing 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 referred to as *long signatures*. A long signature is divided into segments of length k, except for the last segment whose length can be less than k. Segments with length k are called full-length segments.

The string-matching engine is consisted of two units: a pipeline unit and an aggregation unit. The pipeline unit is used to detect pattern segments with up to k characters. A matched segment represents a partial match of a signature. The aggregation unit is responsible for combining the partial-match results to produce the final results.

Consider a signature set with eight strings $\Gamma = \{$dos, pos, disk, disks, passwd, directory, directories, entrance$\}$ shown in Table II. Assume k is equal to 4, strings in the signature set Γ are divided into 12 segments $\Gamma_k = \{$s, y, wd, dos, ies, pos, ance, ctor, dire, disk, entr, pass$\}$. Let the logical segment IDs assigned to the segments in Γ_k be $\mathrm{S}_A, \ldots, \mathrm{S}_L$, respectively. As shown in Figure 3, The automaton of the pipeline system for the set of segmented strings $M_p(\Gamma_k)$ is constructed. A segment detected by $M_p(\Gamma_k)$ can be a short signature having k characters or less, and/or part of a long signature. In the latter case, the detected segment represents a partial-match of a long signature.

Table II shows the set of signatures and the corresponding set of segments with $k = 4$. The assignment of physical IDs will be discussed in Section 4. In this section, our discussion is based on the logical segment IDs. A long signature is represented by a sequence of segment IDs. For example, the segment sequence for "directory" is $\mathrm{S_I S_H S_B}$, where $\mathrm{S_I}$, $\mathrm{S_H}$ and $\mathrm{S_B}$ are the IDs of the segments "dire," "ctor," and "y," respectively. An *aggregation string* for a long signature is defined as follows. Let Y be a long signature, and $S_\mathrm{Y} = S_1 S_2 \ldots S_{\mathrm{t}-1} S_t$ be the corresponding segment sequence. The aggregation string A_Y for signature Y is equal to $S_1 S_2 \ldots S_{t-1} S_t$ if S_t is a full-length segment; otherwise, A_Y is equal to $S_1 S_2 \ldots S_{t-1}$. Let $\boldsymbol{A}$ be the set of aggregation strings with respect to the

Table IIA. Sample Signature Set

Pattern ID	Signature	Length Group	Segment Sequence	Aggregation String
1	disk	—		
2	disks	LS_1	S_JS_A	S_J
3	directory	LS_1	$S_IS_HS_B$	S_IS_H
4	directories	LS_3	$S_IS_HS_D$	S_IS_H
5	entrance	LS_0	S_KS_G	S_KS_G
6	passwd	LS_2	S_LS_C	S_L
7	dos	—		
8	pos	—		

Table IIB. Set of Segments Corresponds to the Sample Signature Set

Segment	Logical Segment ID	Physical Segment ID	Segment	Logical Segment ID	Physical Segment ID
s	S_A	10	ance	S_G	5
y	S_B	11	ctor	S_H	6
wd	S_C	10	dire	S_I	2
ies	S_D	9	disk	S_J	1
dos	S_E	7	entr	S_K	3
pos	S_F	8	pass	S_L	4

signature set Γ, that is, $\boldsymbol{A} = \{A_Y | \ Y \in \Gamma \wedge \text{length}(Y) > k\}$. An AC automaton $M(\boldsymbol{A})$ is constructed for $\boldsymbol{A}$ using Lunteren's approach, as shown in Figure 4. The set of long signatures is divided into k disjoint length groups, LS_0 to LS_{k-1}, where $LS_i = \{Y | Y \in \Gamma \wedge \text{length}(Y) > k \wedge \text{length}(Y) \text{ modulo } k = i\}$. An output state in $M(\boldsymbol{A})$ is unconditional if it corresponds to a long signature in LS_0; otherwise, it is conditional if the output state corresponds to the aggregation string of a long signature in LS_i for $1 \leq i < k$. When the aggregation unit reaches an unconditional output state, a match-result for a long signature in LS_0 is generated. When the aggregation unit reaches a conditional output state, a match-result is generated only if the pipeline unit has found the corresponding partial-length suffix segment.

The organization of the pipeline system is depicted in Figure 5 with $k = 4$. When pipeline stage P_i detects a segment, the segment ID is passed to the corresponding partial-match unit PM_i in the aggregation unit. Pipeline stage P_4 is simply a register that holds the table look-up results of stage P_3. PM_4 is responsible for traversing the automaton $M(\boldsymbol{A})$ for aggregation of partial-matches. Similar to the pipeline unit, the aggregation unit maintains up to k independent threads that traverse $M(\boldsymbol{A})$ concurrently. However, a full-length segment for a given thread may be detected in every k cycles. Hence, the aggregation unit can interleave the processing of the k threads by using a shift register with k buffer slots. As shown in Figure 5, the next state value determined by PM_4 is fed back to the first buffer slot B_1 of the shift register. PM_4 maintains two LUTs (TA_0 and TA_1) for traversing the AC automaton $M(\boldsymbol{A})$. Each PM_i for $1 \leq i < k$ maintains a LUT CM_i for verifying the match of a long signature in LS_i.

The number of states in $M(\boldsymbol{A})$ is determined by the number of full-length segments, which is roughly equal to nL/k, where n is the number of strings, L

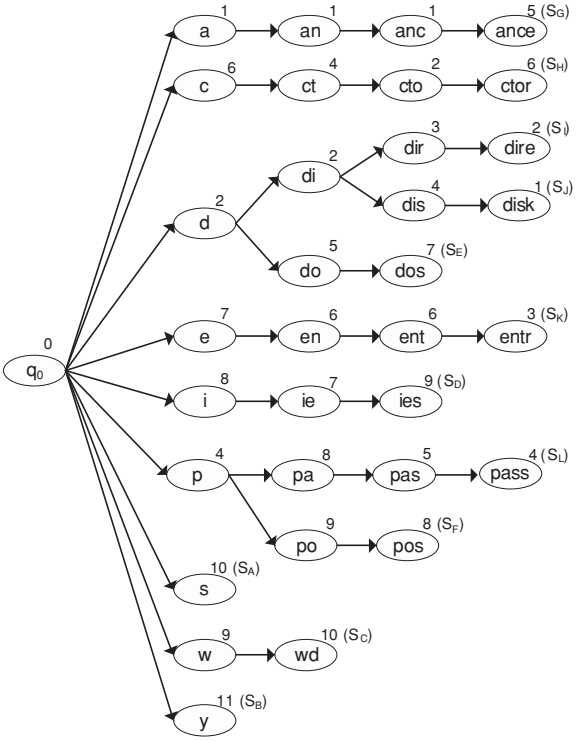

Fig. 3. $M_p(\Gamma_k)$ for the sample signature set. The number next to a node represents the assigned physical state ID (physical segment ID), and the logical segment ID is placed inside the bracket.

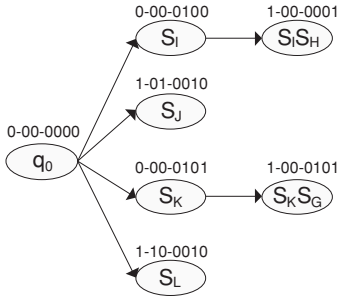

Fig. 4. State graph for the aggregation of partial-matches. The binary number next to a node corresponds to the physical state ID.

is the average string length, and k is the length of a full-segment. The alphabet set of $M(\boldsymbol{A})$ is made up of the segment IDs. For k equals to 4 or larger, the number of distinct full-length segments (i.e., the size of the alphabet set) is comparable to the number of states in $M(\boldsymbol{A})$. In the extreme case where the size of the alphabet set is equal to the number of states (i.e., all the full-length segments are distinct), $M(\boldsymbol{A})$ is reduced to a tree. There will not be any cross edges in the state graph because there does not exist any segment α such that α appears in two strings. In a practical scenario, 40% to 60% of the full-length

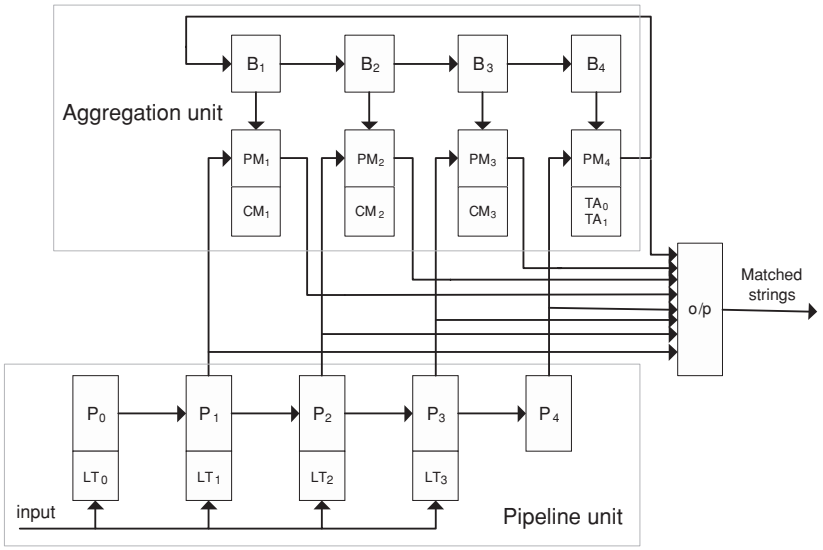

Fig. 5. Organization of the pipeline system.

Table III. Logical Organization of Look-Up Tables TA_0, TA_1, CM_1, CM_2 and CM_3

TA_0		
Input	Next state	Pattern
S_I	<S_I>	—
S_J	<S_J>	"disk"
S_K	<S_K>	—
S_L	<S_L>	—

TA_1			
Current state	Input	Next state	Pattern
<S_I>	S_H	<S_IS_H>	—
<S_K>	S_G	<S_KS_G>	"entrance"

CM_1			CM_2			CM_3		
State	Input	Pattern	State	Input	Pattern	State	Input	Pattern
<S_J>	S_A	"disks"	<S_L>	S_C	"passwd"	<S_IS_H>	S_D	"directories"
<S_IS_H>	S_B	"directory"						

segments are distinct for text-based signature set. The expected number of cross edges is no more than 50% of the number of states in the state graph.

The logical organization of Tables TA_0 and TA_1, and CM_1 through CM_3 for the sample signature set is shown in Table III. Assume the input stream $I =$ "directory ... " Initially, the state value of B_1 to B_4, and P_0 to P_4 are equal to q_0. In cycle 5, P_4 detects the segment "dire" and the segment ID S_I is sent to PM_4. PM_4 looks up TA_0 to determine the next state <S_I>. In cycle 9, P_4 will detect the segment "ctor" and sends the segment ID S_H to PM_4. At this time, the state value stored in B4 is equal to <S_I>. PM_4 finds a matching entry in TA_1 and determines the next state to be <S_IS_H>. In the next cycle, the state value <S_IS_H> will be shifted into B_1, and P_1 will reach state <y>. PM_1 will then use the state value of B_1 and the segment ID S_B received from P_1 to look up its local CM_1 table and generates a match-result for the string "directory."

3.2 Handling Case-Sensitive and Case-Insensitive Signatures

The complexity of the state graph will be largely increased if case sensitive strings and case insensitive strings are included in the same DFA. There are two general approaches to handle case sensitivity. In the first approach, two independent DFAs are constructed for the case-sensitive signatures and the case-insensitive signatures, respectively. Strings in the case-insensitive signature set are converted to lower case letters. The two DFAs are run in parallel. Uppercase letters in the input stream are converted to lowercase before passing to the input interface of the DFA for case-insensitive signatures.

The second approach is based on the match-and-verify strategy [Lu et al. 2007]. All the signatures and the input data stream are converted to lowercase letters. Case sensitivity is verified after a matching string has been found. In the ASCII code, the 5-th bit for a lowercase letter is equal to 1, where the 5th bit for an uppercase letter is equal to 0. The original value of the 5th 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.

The first approach offers an implicit partitioning of the signature set, dividing the signatures into two subsets. In general, signature set partitioning may help to reduce the total memory cost, but the cost of the control logic will be doubled, since the system needs to operate two independent DFAs. The second approach will incur some postprocessing requirement and additional cost in the control logic. Both approaches are viable in the proposed pipelined architecture while implementation of the first approach can be simpler.

4. IMPLEMENTATION OF THE LUTS

The LUTs can be implemented using hardware hashing [Ramakrishna et al. 1997]. However, if a conventional hardware-hashing scheme is used, we cannot guarantee the hash function is collision-free. When collisions happen in the hash table, the system will take multiple cycles to resolve the collisions and the whole pipeline will be slowed down. In this section, we present a method to implement the LUTs such that single cycle look-up operation can be guaranteed.

The system maintains three types of LUTs, namely the transition rule tables LT_0 to LT_3 for the pipeline unit, the transition rule tables for the aggregation unit TA_0 and TA_1, and the conditional match Tables CM_1 through CM_3. The information maintained in the previously mentioned tables are listed in Table IV.

In general, the set of signatures is stored in an array and an individual signature is referenced by the array index. Hence, if there are n signatures, the pattern ID is within the range of 1 to n. In principle, there is a one-to-one mapping of state ID to pattern ID, that is, one can find out the pattern ID from the state ID via another LUT. However, using this approach will make the implementation of TA_0, TA_1, and CM_1 to CM_3 very difficult.

Table IV. Information Maintained in Different Look-Up Tables

Look-up table	Search key	Output information
LT_0	input character	next state of $M_p(\Gamma_K)$, segment ID, pattern ID
LT_1 to LT_3	current state of Mp(Γ_K), input character	next state of $M_p(\Gamma_K)$, segment ID, pattern ID
TA_0	segment ID	next state of $M(\boldsymbol{A})$, pattern ID
TA_1	current state of $M(\boldsymbol{A})$, segment ID	next state of $M(\boldsymbol{A})$, pattern ID
CM_1 to C_{M3}	state ID of $M(\boldsymbol{A})$, segment ID	pattern ID

In this article, we implement the LUTs using an approach based on *direct indexing plus bit-selection* (DIBS). In order to achieve good memory efficiency, we need to do a careful assignment of state IDs, segment IDs, and pattern IDs. What we want to achieve is to overlay the three number spaces such that the state ID is the same as the segment ID and pattern ID that the node may represent. First, we discuss the implementation of Tables LT_0 through LT_3. One can observe that if the transition rules of $M_p(\Gamma_k)$ are divided into multiple tables, the state IDs of nodes on N_1 can be independent of the state IDs of nodes on N_2. The state ID is used as a base address, and a bit-selection scheme is used to determine an offset value from the input character. The transition rule at the location equal to the sum of the base address and offset is retrieved. If the input character is equal to the expected value, the transition rule is fired and the retrieved next state value is passed to the next pipeline stage. Otherwise, a *null* value (i.e., 0) is passed to the next pipeline stage.

We illustrate our design using the sample signature set shown in Table II. The number next to a node in Figure 3 represents the assigned physical state ID. Table LT_0 can be implemented using a 256-entry array indexed by the input character. Each entry in LT_0 contains three fields, *ns* represents the next state value, the o/p flag is used to indicate if the next state is an output state or not, and *bs* is a bit-select mask vector used to control the bit-select circuit when accessing the LUT of the next pipeline stage. If o/p is equal to 1, then the *ns* value that represents the pattern ID will be sent to the output interface in the next cycle. The organization of the control logic for a LUT is depicted in Figure 6. Suppose the *bs* mask has 8 bits, $b_7 \ldots b_1 b_0$. If all the bits in the *bs* mask are equal to 0, then the offset value generated by the bit-select circuit is equal to 0. Assume g bits of the *bs* mask, b_{s1}, b_{s2}, ... b_{sg}, are equal to 1, where $s1 < s2 \ldots < sg \leq 7$. Let the value of the input character be $i_7 \ldots i_1 i_0$. The offset value produced by the bit-select circuit is equal to $0..0 i_{sg} \ldots i_{s2} i_{s1}$.

The contents of LT_0 through LT_3 for the sample signature set are depicted in Figure 7. Note that the address assignment for the transition rules in Tables LT_1 through LT_3 are based on the assigned physical state IDs shown in Figure 3. For example, physical state ID of node <a> is equal to 1 and the transition edge (<a>, 'n', <an>) is stored at location 1 in LT_1. Transitions (<d>, 'i', <di>) and (<d>, 'o', <do>) are stored at locations 2 and 3 in LT_1, where the physical state ID of <d> is equal to 2. Similarly, physical state ID of <do> is equal to 5 and the transition edge (<do>, 's' <dos>) is stored at location 5 in LT_2. Table LT_i stores the transition edges that are originating from nodes on level i. Since Tables LT_0 through LT_3 are implemented on separate physical memory

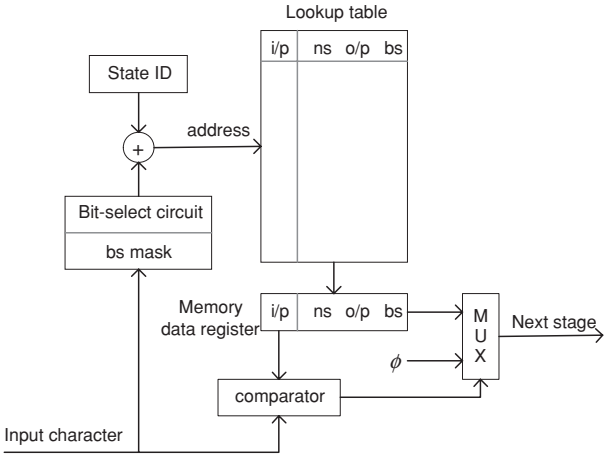

Fig. 6. Control logic for a LUT using DIBS.

	LT_0		
	ns	o/p	bs
'a'	1	0	0000
'b'	*0*	*0*	*0000*
'c'	6	0	0000
'd'	2	0	00**1**0
'e'	7	0	0000
'i'	8	0	0000
'p'	4	0	00**1**0
's'	10	0	0000
'w'	9	0	0000
'x'	*0*	*0*	*0000*
'y'	11	0	0000

	LT_1			
	i/p	ns	o/p	bs
0	*	*0*	*0*	*0000*
1	n	1	0	0000
2	i	2	0	000**1**
3	o	5	0	0000
4	a	8	0	0000
5	o	9	0	0000
6	t	4	0	0000
7	n	6	0	0000
8	e	7	0	0000
9	d	10	0	0000
10	*	*0*	*0*	*0000*
11	*	*0*	*0*	*0000*
12				

LT_2			
i/p	ns	o/p	bs
*	*0*	*0*	*0000*
c	1	0	0000
r	3	0	0000
s	4	0	0000
o	2	0	0000
s	7	**1**	0000
t	6	0	0000
s	9	0	0000
s	5	0	0000
s	8	**1**	0000
*	*0*	*0*	*0000*

LT_3		
i/p	ns	o/p
*	*0*	*0*
e	5	0
r	6	0
e	2	0
k	1	**1**
s	4	0
r	3	0
*	*0*	*0*
*	*0*	*0*
*	*0*	*0*

Fig. 7. LUTs LT_0 through LT_3 for the sample signature set.

modules, the physical state IDs for nodes on different levels are totally independent. However, physical state IDs for nodes on the same level must be distinct. In general, terminal states on levels 1 to 3 are assigned state IDs on the higher end. By doing so, the *null* transition rules, for example, rules 10 and 11 of LT_1, rules 10 of LT_2, and rules 7 to 9 of LT_3, need not be stored in the physical LUTs. This will help to reduce the physical size of the LUTs.

Only the least significant 4 bits of the *bs* mask are shown in Figure 7, and the most significant 4 bits of the *bs* mask are always equal to 0 in this example. If *ns* is equal to 0, it represents the *null* value. Note that LT_3 does not contain the *bs* mask, since no table look-up operation is required in stage P_4. Let's consider the operation of stage P_1. In each cycle, the current state is set to the *ns* value received from P_0. If the received o/p flag is set to 1, then the value of

current state is sent to the output module. Assume the *ns* value received from P_0 is equal to 2 (corresponds to state <d>). The fanout of node <d> in $M_p((\Gamma_k)$ is equal to 2. In this case, a memory block of 2 entries is allocated in LT_1 to store the forward edges originating from node <d>. If the state ID of <d> is equal to 2 and the size of the memory block allocated to the node is equal to 2, then the state ID value 3 will not be available for other node on same level in the state graph. That is to say the state space allocated to node <d> contains two points {2, 3}. The input characters of the two forward edges originating from <d> are equal to 'i' (ASCII code = x69) and 'o' (ASCII code = x6F), respectively. These two characters can be distinguished by the second least significant bit. Hence, the lower 4 bits of the *bs* mask is equal to 0010. On receiving an input character x, it extracts the second least significant bit of x to form the offset value.

The size of the memory block allocated to a node is always a power of 2. For a node u with fanout greater than 1, a heuristic algorithm is used to compute the bit-select mask with the smallest number of bits set to 1. If there are g bits in *bs* that are equal to 1, then the size of the memory block allocated to node u in the LUT is equal to 2^g. The state value of a node is always an integral multiple of the size of the memory block allocated to it. Hence, the memory address used to access the local LUT is obtained by performing a logical-OR operation of the state ID with the offset value produced by the bit-select circuit. If the input character x is equal to the expected input retrieved from the LUT, the retrieved ns, o/p, and *bs* are passed to the next pipeline stage; otherwise, *null* values are passed to the next pipeline stage.

The assignment of state ID in $M_p(\Gamma_k)$ is to facilitate efficient implementation of TA_0, TA_1, and CM_1 to CM_3. Discussion on the state ID assignment strategy will be delayed after the presentation of the design of the other LUTs. There are a few important properties of $M(\boldsymbol{A})$ one must be aware of in deriving a memory-efficient implementation of the LUTs. First, the fanout of q_0 is proportional to the number of strings in the signature set (i.e., in the range of thousands). Except q_0, most of the other states have fanout equal to 0 or 1. Only a small number of states may have higher fanout. Second, recall that the automaton of the aggregation unit $M(\boldsymbol{A})$ is implemented using Lunteren's approach. The system maintains two transition rule tables, TA_0 and TA_1. These two tables are searched in parallel. If match-results are found in both tables for a given input (the segment ID received from the pipeline unit), priority will be given to the match-result of TA_1. All failure edges pointing back to q_0 are implicit in the state graph. That is to say, if the system searches the two tables and does not find any matching entry, then q_0 is taken as the default next state. There are a relatively large number of terminal nodes in $M(\boldsymbol{A})$. No transition rules for these terminal nodes need to be stored in TA_1 because the only possible next state is q_0, but the state ID of a terminal node may be used to reference the CM tables.

TA_0 can be implemented using the same approach of LT_0 provided that the segment IDs involved in all the forward edges of the initial state of $M(\boldsymbol{A})$ are numbered in the range 1 to $f + \Delta$, where f is the fanout of q_0, and Δ is equal to the number of signatures with length equal to 4 and the corresponding four-character signature is not the first segment of a long signature. By doing

Table V. Format of State ID for $M(\boldsymbol{A})$

Terminal State	Conditional Output State	State ID			Access table				Remark
		T	CT	AD	CM_1	CM_2	CM_3	TA_1	
yes	yes	1	00	$00a_9 \cdots a_0$	√	√	√	×	
			01	$00a_9 \cdots a_0$	√	×	×	×	
			10	$00a_9 \cdots a_0$	×	√	×	×	
			11	$00a_9 \cdots a_0$	×	×	√	×	
	no		$a_{13}a_{12}$	$a_{11}a_{10}a_9 \cdots a_0$	×	×	×	×	$a_{11}a_{10} \neq 00$
no	yes	1	00	$00a_9 \cdots a_0$	√	√	√	√	
			01	$00a_9 \cdots a_0$	√	×	×	√	
			10	$00a_9 \cdots a_0$	×	√	×	√	
			11	$00a_9 \cdots a_0$	×	×	√	√	
	no	1	$a_{13}a_{12}$	$a_{11}a_{10}a_9 \cdots a_0$	×	×	×	√	$a_{11}a_{10} \neq 00$

so, we can use the segment ID as the array index to access Table TA_0. The remaining LUTs (i.e., TA_1 and CM_1 to CM_3) can be implemented using the DIBS approach. However, there are additional constraints in the assignment of state ID in $M(\boldsymbol{A})$ and the IDs of full-length segments. A state ID can be used as the base address to access TA_1 and some of the CM tables if the node is a conditional output node. However, the size of TA_1 is much larger than the CM tables. We do the state ID assignment in such a way that only the lower-order bits of the state ID will be used to access a CM table. A state in $M(\boldsymbol{A})$ can be related to more than one CM table. We define the CM-group (CMG) attribute of a state in $M(\boldsymbol{A})$ as follows. If state u is not a conditional output state (i.e., not related to any CM table), then its CMG is equal to -1. If u is related to multiple CM tables, then CMG of u is equal to 0. If u is related to exactly one CM table, then CMG of u is equal to the table number (i.e., 1 to 3).

The state ID for $M(\boldsymbol{A})$ contains three fields, T-CT-AD, as shown in Table V. Field T is used to represent if the node is a terminal node or not. The CT field can have four possible values 00, 01, 10, and 11. The field size of AD depends on the number of nodes in $M(\boldsymbol{A})$ and the sizes of the CM tables. Let the length of the AD field be a bits. The size of the largest CM table should be less than 2^{a-2}, and the sum of the number of nodes in $M(\boldsymbol{A})$ and the size of the largest CM table should be less than 2^{a+2}. For example, in the case-insensitive signature set used in the performance evaluation with 3,635 strings, the number of nodes in $M(\boldsymbol{A})$ is 13,100, and the sizes of the CM tables are about 600 to 700 entries. In this case, the AD field will have 12 bits. If the current state is a terminal state, then the next state will always be q_0, and there is no need to look up Table TA_1. If the current state is a nonterminal state, then the values of CT and AD fields will form the base address for accessing Table TA_1. If the two most significant bits of AD are equal to 0, then the least significant 10 bits of the AD field will be used as the base address to access one or more of the CM tables, depending on the value of CT.

We are now ready to present the heuristic strategy for the segment ID assignment. Segments of different lengths are used to access different LUTs. Hence, the segment ID assignment for different length groups can be done independently, except for segments that correspond to short signatures because the IDs of these segments should be the same as their respective pattern IDs.

Let τ_4 be the set of signatures of length 4, and τ_3 be the set of signatures of length less than 4. Segments in τ_4 are assigned physical IDs starting from 1 and onward. Segments in τ_3 are assigned segment ID starting from n and downward subject to the address block size constraint. In the previous example, the physical segment ID of "disk" is equal to 1, and the segment ID of "dos" and "pos" are equal to 7 and 8, respectively.

The ID assignment for full-length segments is most critical because it will affect the memory efficiency of Table TA_1, the largest table in the system. We first construct $M(\boldsymbol{A})$ using the logical segment IDs. The logical segment IDs involved in TA_0 are then collected. For each segment in this group that has not been assigned a physical segment ID, it is assigned the smallest ID value that is available. Next, we determine the list of nodes in $M(\boldsymbol{A})$ that have fanout greater than 1, except q_0. The list is ordered by the fanout value in decreasing order. For each node in this list, we determine the group of logical segment IDs associated with that node. We try to assign consecutive physical IDs to the segments in the same group. By using this assignment strategy, the number of bits required in the bit-selection operation will be minimized.

The ID assignment for segments with length 1 to 3 is less critical because it will only affect the CM tables whose sizes are relatively small. In addition, the number of short segments associated with a conditional output node in $M(\boldsymbol{A})$ is usually very small. The assignment of physical ID values to segments shorter than 4 is mainly constrained by the construction of Tables LT_1 through LT_3.

After the assignment of the physical segment ID, the bit-select mask vector and the required block size of each node in the state graph will be computed. The amount of state space (address block size) allocated to a node in $M(\boldsymbol{A})$ is equal to the maximum of the required block size for the Tables TA_1 and CM_1 through CM_3. An address allocation vector is maintained for each of the four tables. If an address block $[a, b]$ is allocated to a node u with $CMG = 0$, then the corresponding address range of the allocation vector of all the four tables is marked. If an address block $[a, b]$ is allocated to a node u with $CMG = j$ where $j > 0$, then only the allocation vector of TA_1 and CM_j will be marked. The assignment of state ID to nodes in the state graph is carried out in the following order.

(i) internal node, output state, CMG = 0
(ii) terminal node, output state, CMG = 0
(iii) internal node, CMG = 0
(iv) terminal node, CMG = 0
(v) internal node, output state, CMG > 0
(vi) terminal node, CMG > 0
(vii) internal node, output state, CMG < 0
(viii) internal node, nonoutput state, CMG < 0
(ix) terminal node, output state, CMG < 0

The lower order $\lfloor \log n \rfloor$ bits of the ID of an output node correspond to the associated pattern ID. Hence, the lower order $\lfloor \log n \rfloor$ bits of the state ID for

Table VI. Contents of TA_0, TA_1, and the CM Tables for the Sample Signature Set

TA_0				TA_1					
Address (segment ID)	Next state	*bs*	CMG	Address (binary)	Segment ID	Next state	o/p	*bs*	CMG
1 (S_J)	1-01-0010	0..0	1	00-0000	*	0-00-0000	0	0..0	−1
2 (S_I)	0-00-0100	0..0	−1	00-0001	empty	empty	—	—	—
3 (S_K)	0-00-0101	0..0	−1	00-0010	empty	empty	—	—	—
4 (S_L)	1-10-0010	0..0	2	00-0100	6 (S_H)	1-00-0001		0..0	0
				00-0101	5 (S_G)	1-00-0101	1	0..0	−1

	CM_1			CM_2			CM_3		
Address (binary)	Segment ID	Pattern ID	*bs*	Segment ID	Pattern ID	*bs*	Segment ID	Pattern ID	*bs*
01	11 (S_B)	3	0..0	—	—	0..0	9 (S_D)	4	0..0
10	10 (S_A)	2	0..0	10 (S_C)	6	0..0	—	—	0..0

an output node must be distinct. This constraint is not difficult to fulfill, since less than 10% of the nodes in $M(\boldsymbol{A})$ are output nodes. An allowable physical segment ID and state ID assignment for the sample signature set is shown in Table VI. The last entry in TA_1 corresponds to a transition to an output state in $M(\boldsymbol{A})$. The last 4 bits of the state ID represent the pattern ID, that is, the pattern ID for "entrance" is equal to 5 (or 0101 in binary). Note that the CMG field is only used in the segment ID assignment process, it is not required in the physical LUTs. The bit-select mask values for all table entries are equal to '0' in this example. The *bs* mask stored in TA_0 and TA_1 is used to access table TA_1 to determine the next state in $M(\boldsymbol{A})$. The *bs* mask used to access a CM table is stored in the corresponding CM table. A CM table is divided into two physical partitions, CM_{bs} and CM_{data}. The CM_{bs} partition stores the *bs* masks and the CM_{data} partition stores the segment ID and pattern ID pairs. The look-up operation of the CM table involves two steps. First, the system reads the *bs* mask from CM_{bs}. Second, the system searches CM_{data} with the segment ID and the *bs* mask obtained in step 1. The access to CM_{bs} and CM_{data} can be pipelined without affecting the operation of the other parts of the system.

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed method using a signature set extracted from the Snort rule set in April 2008. All static strings defined in the rule set are included in the performance evaluation. In the extracted signature set, 1,221 strings contain binary data. The binary data accounts for 14% of the total byte count of the signature set. Patterns that contain only binary data are placed in the case-sensitive set. Membership of patterns that contains both text and binary data depends on the case-sensitivity requirement on the text component. The statistical information of the case-sensitive and case-insensitive signature set are summarized in Table VII.

We evaluated the total number of rules in all LUTs for different values of k, as shown in Table VIII, and found that the smallest number of rules is obtained when k is equal to 4. Hence, our detailed performance evaluation of P-AC is based on $k = 4$.

Table VII. Statistics of the Snort Signature Set

	Case Insensitive	Case Sensitive
Number of signatures	3,635	2,531
Maximum length	109	122
Average length (character)	19.5	11.6
Total number of characters	70,732	29,229

Table VIII. Total Number of Entries in All Look-Up Tables for Different Pipeline Lengths

Signature set	$k = 3$	$\boldsymbol{k = 4}$	$k = 5$	$k = 6$
Case insensitive	34,539	**32,505**	33,044	34,526
Case sensitive	19,426	**17,517**	18,089	18,967

In this section, we assume two separate hardware pipelines are built to process case-sensitive signatures and case-insensitive signatures independently. For the processing of case-insensitive signatures, all the strings are converted into lowercase letters. Before the input stream is matched against the case-insensitive signatures, all letters in the input stream are also converted to lowercase. In the construction of Tables LT_1 through LT_3, four parallel memory modules are used so that better memory efficiency can be achieved by the bit-selection method. Consider an extreme case where the fanout of a node is equal to 4. Let the input characters of the four transition rules be x01, x02, x04 and x08. If only one memory module is used, we need to allocate an address block of size 8 to the node, that is, the bit-select mask is 0000-0111. If four parallel memory modules are used, then the node can be assigned an address block of size 1, since the four transition rules can be stored at the same address on different memory modules. In general, the use of multiple memory modules can reduce the number of bits required in the bit-select mask and improve the memory efficiency. In this study, Tables LT_1, LT_2, and LT_3 are implemented using four parallel memory modules. The four memory modules can have different sizes, as shown in Figure 8. The address (state ID) allocation for nonoutput nodes follows the strategy shown in Figure 8. For output nodes, the state ID allocation is constrained by the requirement that the state ID should be equal to the pattern ID. This constraint can be fulfilled quite easily, since only a few percents of strings in the signature set have length shorter than or equal to 4.

In principle, using two or more memory modules can also help to improve the memory efficiency of TA_1. However, in the Altera Startix II FPGA devices used in this study, there are a small number of large memory modules with 512Kbits capacity, and a larger number of memory modules with smaller capacity, for example, 512 bits and 4Kbits. The size of TA_1 fits into a 512Kbits SRAM module, so the use of parallel memory modules will not offer any real improvement. Table IX shows the number of transition rules and the physical size of each LUT. We can see that using the DIBS method to implement the LUTs incurs an overall overhead of 25% and 34% for the case-insensitive and case-sensitive signature set, respectively.

Table X lists the field widths of the LUTs. The overall amount of memory required for the LUTs for the case-insensitive and case-sensitive signature sets

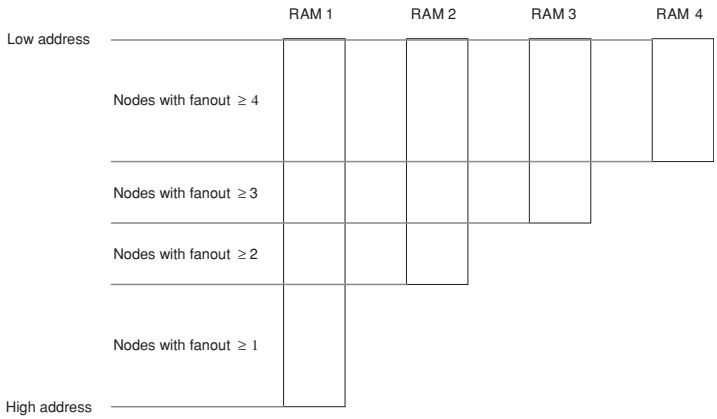

Fig. 8. Organization of LUT with four parallel memory modules.

Table IX. Number of Rules and Physical Size of Look-Up Tables in P-AC

Case-Insensitive Signature Set										
Table	LT_0	LT_1	LT_2	LT_3	TA_0	TA_1	CM_1	CM_2	CM_3	Total
Transition rules	75	1,243	5,131	6,984	1,697	13,662	718	671	627	30,808
Physical table size	256	2,112	6,913	7,599	1,896	17,614	742	692	657	38,481

Case-Sensitive Signature Set										
Table	LT_0	LT_1	LT_2	LT_3	TA_0	TA_1	CM_1	CM_2	CM_3	Total
Transition rules	208	1,694	3,516	4,109	1,323	4,007	461	485	391	16,194
Physical table size	256	3,015	4,528	4,508	1,536	6,415	489	504	413	21,661

are 1.18Mbits and 0.6Mbits, respectively. The overall normalized memory cost of P-AC is about 18 bits/char. The control logic in P-AC is very simple. The comparators and bit-select circuits for one pipeline system can be implemented using less than 1,000 four-input LUTs. (The LUT in this context refers to the basic logic circuit building block in FPGA devices.) Since dual-port memories are available in today's FPGA, we can duplicate the control logic and process two data streams at the same time. Hence, the system throughput can be doubled with little overhead. With two independent pipelines for case-sensitive and case-insensitive signatures, the total cost of the control logic is about 4,000 LUTs.

A comparison with some other AC-based string-matching methods is summarized in Table XI. The memory cost of P-AC, BFPM [Lunteren 2006], and the TCAM-based method of [Alicherry et al. 2006] are evaluated using the same signature set. In the BFPM method, both the case-sensitive and case-insensitive signature sets are divided into eight groups. The number of transition rules per character is about 2.2. With 36-bit transition rules, the memory cost for BFPM is about 79 bits/char. The hardware cost of bit-split FSM is obtained from Jung et al. [2006]. The performance of CDFA is obtained from Song et al. [2008], and the performance of split-AC is obtained from Dimopoulos et al. [2007]. For the split-AC method, there can be trade-off between memory cost and control logic. The chip area for a FPGA logic cell is approximately the same as 12 bytes

Table X. Field Sizes (Number of Bits) of the Look-Up Tables

	Case-Insensitive Signature Set					Case-Sensitive Signature Set				
Table	i/p	*ns*/*pid*	o/p	*bs*	Total	i/p	*ns*/*pid*	o/p	*bs*	Total
LT_0	—	8	1	6	15	—	8	1	8	17
LT_1	8	12	1	6	27	8	11	1	7	27
LT_2	8	13	1	6	28	8	12	1	6	27
LT_3	8	13	1	—	22	8	13	1	—	22
TA_0	—	15	—	9	24	—	12	—	9	21
TA_1	13	15	1	9	38	12	13	1	9	35
CM_1	8	13	—	2	23	8	12	—	3	23
CM_2	12	13	—	2	27	12	12	—	6	30
CM_3	13	13	—	5	31	12	12	—	7	31

Table XI. Comparison of AC-Based Methods

Method	Signature Set (chars)	Memory (per char)	Control Logic (LUT/char)	Speed (char/cycle)
bit-split FSM	16.7K	184 bits	0.27	1 (note 1)
split-AC	24K	65 bits	2.5	1 (slower clock)
		189 bits	0.5	
BFPM (16 groups)	100K	79 bits	data not available	1
TCAM (2 char/cycle)	100K	76 bits TCAM + 41 bits SRAM	not applicable	Up to 2 (max 266MHz clock)
TCAM (3 char/cycle)	100K	24 bits TCAM + 10 bits SRAM	not applicable	Up to 3 (max 266MHz clock)
CDFA (4 groups, 2-way associative)	29K	49.6 bits	data not available	1 (interleave 2 data streams)
CDFA (4 groups, 8-way associative)	29K	26.4 bits	data not available	1 (interleave 2 data streams)
P-AC	100K	18.1 bits	0.04	2 (2 concurrent data streams)

Note: Up to 4 char/cycle is possible, but the hardware cost will also be increased substantially.

of memory [Sourdis and Pnevmatikatos 2008]. Hence, the actual hardware cost for the two sets of implementation parameters shown in Table XI are more or less the same.

From Table XI, we can see that P-AC has a clear performance advantage over the other methods. The memory cost of P-AC is lower than BFPM and CDFA (four groups, 8-way associative) by 75% and 30%, respectively. Moreover, the throughput of P-AC is two times that of CDFA.

The Altera Stratix II EP2S60 FPGA contains 2×512Kbit RAM blocks, 255×4Kbit RAM blocks, and 329×512 bit RAM blocks. The small memory blocks can be cascaded to form larger memory arrays. The total memory capacity of the EP2S60 is about 2.5Mbits. In our proposed P-AC method, the signature set is only divided into two groups, namely the case-sensitive signatures and case-insensitive signatures. Hence, there are two sets of LUTs, where TA_1 has a larger size compared to the other tables. The total memory required by P-AC is about 1.8Mbits. The two 512Kbit RAM blocks are used to implement TA_1 of the two pipelines, and the remaining tables can be implemented using the smaller size RAM blocks. The control logic (including the duplicated copy

to enable concurrent processing of two data streams) consumes about 7% of the logic elements available in the EP2S60. The 512bit, 4Kbit and 512Kbit memory blocks can operate at 500MHz, 550MHz, and 420MHz, respectively. We have simulated the critical path of the pipelines. Pipeline stages with four parallel LUTs constructed using 500/550MHz memory blocks can operate at 270MHz. Pipeline stages with LUTs constructed using the 420MHz 512Kbit memory block can operate at 253MHz. Hence, the overall system clock frequency is equal to 253MHz.

5.1 Dimensioning of LUTs

In a practical deployment of the string-matching engine, the engineer needs to decide on the data field sizes and memory allocations based on some design targets. Let n be the target number of strings to be supported by the system, and L is the expected average string length. The total number of characters in the signature set $m = n \times L$. The expected number of distinct k-character segments is estimated using the equation $S = f_d \times m/k$, where the factor f_d depends on the statistical property of the signature set. Based on the experiences in our evaluation, f_d is about 0.4 for the case-insensitive signature set, which is dominated by text-based strings, and f_d is about 0.56 for the case-sensitive signature set, where 40% of the strings contain binary data. For pure binary data signature sets, the expected value of f_d can be in the range of 0.8 to 0.9.

For the pipeline unit, the length of the *bs*-mask (hence the bit selection circuit) and comparator circuits are no more than 8 bits as long as we are doing character-based string matching. The size of LT_0 is fixed at 256. The sizes of LT_1 to LT_3 will increase progressively. The number of entries in LT_3 is equal to S, and the physical table size can be set to $1.5 \times S$ to account for possible overhead due to the DIBS method.

For the aggregation unit, the number of states is given by m/k, that is, the number of 4-character segments. The size of the *ns* field is upper-bounded by $\log(m/k)$. The size of Table TA_0 is equal to n. The size of the *bs*-mask is upper-bounded by $\log S$. About 50% of the segment IDs are distinct, so the allocation for Table TA_1 can be set at $1.5 \times m/k$. Assume the lengths of the signatures are uniformly distributed, the size of CM_1 to CM_3 can be set at $n/4$.

The previous recommended table sizes are only for the nominal cases. If the system is implemented on FPGA, the engineer may try to utilize all the memory blocks available in the device to make provision for spare capacity to accommodate future expansion of the signature set.

6. CONCLUSION AND FUTURE WORK

Hardware implementation of the AC algorithm using FPGA or ASIC has received much attention in the past few years. A major issue in the design of string-matching engine based on the AC algorithm is the prohibitive memory requirements. In this article, we show that by incorporating pipelined processing, all cross edges in the AC state graph can be removed. This offers very substantial reduction of the memory cost in the hardware implementation. We have derived an intelligent strategy to assign physical IDs to states and

string segments such that the LUTs can be implemented with good efficiency, and the look-up operation is guaranteed to complete in one cycle. Our design is memory-based. Incremental changes to the signature set can be accommodated by updating the contents of the LUTs without the needs to reconfigure the FPGA.

The control structure of the pipeline system is simple and elegant. If dual-port memories are available, we can duplicate the control logic to allow the pipeline system to process two data streams concurrently. By doing so, the system throughput is increased to two characters per cycle. We have detailed the design of our method using a signature set extracted from the Snort database. The signature set contains 6,166 strings with close to 100K characters. The memory cost of our method is only 18 bits/char, while the cost of the control logic is only 0.04 LUT/char. The pipelines can operate at 253MHz when implemented on the Stratix II FPGA device.

A fundamental property of AC-based string-matching method is that the whole pattern set needs to be stored in embedded memory. Hence, the memory cost cannot be lower than 8 bits/char. For antivirus systems, such as ClamAV, the number of strings in the virus database is over 82 thousands, and the average pattern length is about 102 bytes. The overall size of the ClamAV pattern set exceeds 8Mbyte, which is more than 80 times the size of the Snort pattern set. If a string-matching engine is to be built for the ClamAV pattern set using the AC-based approach, over 20Mbyte of embedded memory will be required. The required memory resource is well beyond the capacity of today's FPGA devices. In a related study by the first author [Pao et al. 2010], a hybrid architecture for matching the ClamAV pattern set was developed, where P-AC was one of the major building blocks. The hybrid architecture has two major processing pipelines, namely the P-AC and the QSV. QSV is a checksum-based approach that uses quick sampling of fixed-length prefix plus on-demand verification of variable-length suffix. The overall memory cost of the hybrid architecture for the ClamAV pattern set is about 1.4Mbyte (i.e., only 1.4 bits/char).

Specification of intrusion patterns and virus using regular expression is gaining popularity. In a future study, we will investigate hardware architectures to accelerate the matching of regular expressions.

ACKNOWLEDGMENTS

The authors would like to thank Mr. Xing Wang for his assistance in carrying out the hardware performance evaluation. The authors are grateful to the anonymous referees for their constructive comments and advice that helped to improve the presentation of this article.

REFERENCES

AHO, A. V. AND CORASICK, M. J. 1975. Efficient string matching: An aid to bibliographic search. *Comm. ACM. 18*, 6, 333–340.

ALDWAIRI, M., CONTE, T., AND FRANZON, P. 2005. Configurable string matching hardware for speeding up intrusion detection. *ACM SIGARCH Comput. Archit News 33*, 1, 99–107.

ALICHERRY, M., MUTHUPRASANNA, M., AND KUMAR, V. 2006. High-speed matching for network IDS/IPS. In *Proceedings of the IEEE International Conference on Network Protocols*. IEEE, Los Alamitos, CA, 187–196.

BAKER, Z. K. AND PRASANNA, V. K. 2005. A computationally efficient engine for flexible intrusion detection. *IEEE Trans. VLSI Syst. 13*, 10, 1179–1189.

BAKER, Z. K. AND PRASANNA, V. K. 2006. Automatic synthesis of efficient intrusion detection systems on FPGAs. *IEEE Trans. Depend. Secure Comput. 3*, 4, 289–300.

CLARK, C. R. AND SCHIMMEL, D. E. 2003. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*. Springer, Berlin, 956–959.

CHO, Y. H. AND MANGIONE-SMITH, W. H. 2005. A pattern matching co-processor for network security. In *Proceedings of the IEEE Design Automation Conference*. IEEE, Los Alamitos, CA, 234–239.

DHARMAPURIKAR, S., KRISHNAMURTHY, P., SPROULL, T. S., AND LOCKWOOD, J. W. 2004. Deep packet inspection using parallel Bloom filters. *IEEE Micro 24*, 1, 52–61.

DHARMAPURIKAR, S. AND LOCKWOOD, J. 2006. Fast and scalable pattern matching for network intrusion detection systems. *IEEE J. Sel. Areas Comm. 24*, 10, 1781–1792.

DIMOPOULOS, V., PAPAEFSTATHIOU, I., AND PNEVMATIKATOS, D. 2007. A memory-efficient reconfigurable Aho-Corasick FSM implementation for intrusion detection systems. In *Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulations*. IEEE, Los Alamitos, CA, 186–193.

JUNG, H. J., BAKER, Z. K., AND PRASANNA, V. K. 2006. Performance of FPGA implementation of bit-split architecture for intrusion detection systems. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. IEEE, Los Alamitos, CA.

KNUTH, D., MORRIS, J., AND PRATT, V. 1977. Fast pattern matching in strings. *SIAM J. Comput. 6*, 323–350.

LU, H., ZHENG, K., LIU, B., AND SUN, C. 2007. A robust approach for matching mixed case-sensitive and case-insensitive patterns. In *Proceedings of the International Conference on Networking and Services*. Springer, Berlin.

LU, H. B., ZHENG, K., LIU, B., ZHANG, X., AND LIU, Y. H. 2006. A memory-efficient parallel string matching architecture for high-speed intrusion detection. *IEEE J. Sel. Areas Comm. 24*, 10, 1793–1804.

LUNTEREN, J. 2006. High-performance pattern-matching for intrusion detection. In *Proceedings of the 25th Conference on Computer Communication (INFOCOM'06)*. IEEE, Los Alamitos, CA, 1–13.

PAO, D., LI, Y. K., AND ZHOU, P. 2006. Efficient packet classification using TCAMs. *Comput. Networks 50*, 18, 3523–3535.

PAO, D., LIN, W., AND LIU, B. 2008. Pipelined architecture for multistring matching. *IEEE Comput. Archit. Lett. 7*, 2.

PAO, D., WANG, X., WANG X., CAO, C., AND ZHU, Y. 2010. String searching engine for virus scanning. *IEEE Trans. Comput.* To appear.

PAPADOPOULOS, G. AND PNEVMATIKATOS, D. 2005. Hashing + memory = low cost, exact pattern matching. In *Proceedings of the IEEE International Conference on Field—Programmable Logic and Applications*. IEEE, Los Alamitos, CA, 39–44.

RAMAKRISHNA, M. V., FU, E., AND BAHCEKAPILI, E. 1997. Efficient hardware hashing functions for high—performance computers. *IEEE Trans. Comput. 46*, 12, 1378–1381.

SONG, T., ZHANG, W., WANG, D., AND XUE, Y. 2008. A memory efficient multiple pattern matching architecture for network security. In *Proceedings of the IEEE INFOCOM*, 673–681.

SOURDIS, I. AND PNEVMATIKATOS, V. 2004. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA.

SOURDIS, I., PNEVMATIKATOS, V., AND VASSILIADIS, S. 2008. Scalable multigigabit pattern matching for packet inspection. *IEEE Trans. VLSI Syst. 16*, 2, 156–166.

TAN, L., BROTHERTON, B., AND SHERWOOD, T. 2006. Bit-split string-matching engines for intrusion detection and prevention. *ACM Trans. Archit. Code Optim. 3*, 1, 3–34.

TUCK, N., SHERWOOD, T., CALDER, B., AND VARGHESE, G. 2004. Deterministic memory-efficient string matching algorithms for intrusion detection In *Proceedings of the 23rd Conference on Computer Communications (INFOCOM'04).* IEEE, Los Alamitos, CA, 2628–2639.

WU, S. AND MANBER, U. 1992. Fast text searching: Allowing errors. *Commun. ACM 35*, 10, 81–93.

YU, F., KATZ, R. H., AND LAKSHMAN, T. V. 2004. Gigabit rate packet pattern-matching using TCAM. In *Proceedings of the IEEE International Conference on Network Protocols*. IEEE, Los Alamitos, CA, 174–183.

ZHENG, K., HU, C., LU, H., AND LIU, B. 2006. A TCAM-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM Trans. Network. 14*, 4, 863–875.

Received September 2008; revised September 2009; accepted March 2010