2008 14th IEEE International Conference on Parallel and Distributed Systems

Pipelined Parallel AC-based Approach for Multi-String Matching

Wei Lin[1, 2], Bin Liu[1]
[1] *Department of Computer Science and Technology, Tsinghua University, Beijing, China*
[2] *Department of Electronic Engineering, City University of Hong Kong, Hong Kong*
lin-w04@mails.tsinghua.edu.cn

Abstract

New applications such as real-time packet processing require high-speed string matcher, and the number of strings in pattern store is increasing to tens of thousands, which requires a memory efficient solution. In this paper, a pipelined parallel approach for hardware implementation of Aho-Corasick (AC) algorithm for multiple strings matching called P2-AC is presented. P2-AC organizes the transition rules in multiple stages and processes in pipeline manner, which significantly simplifies the DFA state transition graph into a character tree that only contains forwarding edges. In each stage, parallel SRAMs are used to store and access transition rules of DFA in memory. Transition rules can be efficiently stored and accessed in one cycle. The memory cost is less than 47% of the best known AC-based methods. P2-AC supports incremental update and scales well with the increasing number of strings. By employing two-port SRAMs, the throughput of P2-AC is doubled with little control overhead.

1. Introduction

String matching has been extensively studied in the past 30 years. A string C of length n is a sequence of characters C1C2…Cn. Let Σ={Y1, Y2, … YN} be a finite set of strings called patterns, and let I be an arbitrary string. The string matching problem is to identify and locate all substrings of I which are patterns in Σ.

Nowadays new applications such as real time intrusion detection, anti-virus scanner and spam filter etc. require high-speed hardware assisted string matcher, and the number of strings in pattern store is increasing to tens of thousands, which requires a memory efficient solution.

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 linear time proportional to the length of the input stream. However, the memory requirement is too expensive in a naive hardware implementation. In this paper, a multiple string matching architecture for tens of thousands of patterns called P2-AC is proposed, which uses a pipelined processing approach with parallel SRAMs to the implementation of AC algorithm.

Our contributions are three folds. Firstly, transition rules of are classified into multiple groups and stored in corresponding stages. Input character is sent to all the pipeline stages simultaneously, which simplifies the state graph into a tree and guarantees the edge reduction algorithmically. Previously published AC-based methods are heuristic-based, such as bit-map encoding and path compression [2], bit-slice implementation [3], categorizing alphabets based on frequency count [4], and pattern set partitioning [5]. Performances of these methods are sensitive to the size and statistical properties of the pattern set.

Secondly, in real implementation, the number of pipeline stages is fixed to a constant value K. Patterns longer than K are divided into multiple segments with the length equal to or less than K. A DFA aggregation approach is used to combine partial match results to match the whole pattern. By carefully organizing the lookup tables and allocating the state and segment IDs, only a small quantity of history information needs to be maintained.

Thirdly, in order to efficiently store and access transition rules in the DFA state graph, the field of current state ID in transition rule is not stored in SRAM but is used as address to access candidate next state. In order to get the next state ID in one cycle, transition rules with the same current state ID in one stage are stored at the same address of different SRAMs. During

This work is supported by NSFC (60573121 and 60625201), the Cultivation Fund of the Key Scientific and Technical Innovation Project, MoE, China (705003), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20060003058), and 863 high-tech project (2007AA01Z216,2007AA01Z468)

1521-9097/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPADS.2008.47

lookup process, entries at the same address of multiple SRAMs are accessed in parallel. The outputs of SRAMs are stored into corresponding registers, which are functioned like a small BCAM compared with input symbol to decide the next state.

The discussion in this paper is restricted to the processing of simple strings. Extensions to handle strings defined by regular expressions will be discussed in the future work. A brief review of related works is given in section 2. The proposed P2-AC architecture will be presented in section 3. Section 4 is the performance evaluation and section 5 is the conclusion.

2. Related works

In the basic AC algorithm 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.

To facilitate discussion, some terminologies are introduced. 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. Nodes in the state graph can be assigned 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.

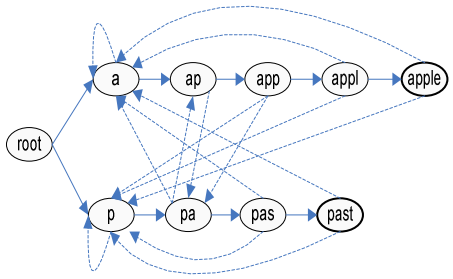

Figure 1 State graph for Σ ={apple, past}.

Some pattern matching architectures are logic based. The key issue is how to efficiently map patterns into programmable circuit logics of FPGA[6-9]. Those logic based methods must reprogram FPGA when pattern set needs to update, and the size of pattern set is restricted because the logic resource in FPGA is expensive and limited. I. Sourdis [9] and C.R.Clark [10] present their architectures to give several methods to efficiently map patterns on FPGAs.

Some pattern matching architectures are RAM based using AC like algorithms. Patterns are stored in RAMs other than logics. The key issue is how to use less memory to support bigger pattern set with deterministic high throughput. Tuck et al [2] proposed a bit-map encoding and path compression technique to reduce the memory cost of the transition rule table. Tan and Sherwood [3] proposed to use bit-split finite state machines (FSMs) where each bit-split FSM processed one bit of an input byte, if there are N signatures, there will be N/2 FSMs. Lunteren [5] proposed a novel approach to substantially reduce the size of the transition rule table. All edges pointing to the same node v on L1 are 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. Pao proposed a multiple stages architecture [11] to reduce the transition rules, which used hashing scheme to get the next state when accessing each lookup table, hashing confliction cannot be avoided and the performance is not guaranteed. Song proposed a n-step cache architecture [12] to eliminate a part of cross edges from the graph and use a complicated data structure to store and access transition rules with the same current state, it assumes that most of the cross edges belong to 1-step category and the number of states shared by multiple transition rules as current state is rare, if the statistical properties of the pattern set change, the memory cost for Song's method may not have an ideal result.

Some pattern matching architectures are TCAM based. Patterns are fully or partly stored in Ternary Content addressable Memory (TCAM). The speed and memory efficiency are partially offset by the slower clock rate (maximum 266 MHz) and higher cost of TCAM. Alicherry et al [13] used TCAM to implement the transition rule table. In their method, state transitions are based on multiple (typically 2 to 4) input characters. Dimopoulos et al [4] 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 transitions rule table for infrequent characters is implemented using CAM, the

performance depends on statistical properties of the signature set.

There are also proposals based on hashing [14-16] and Bloom filters [17, 18]. 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 and/or hardwired aggregation logic are required to combine the partial match results.

In this paper, P2-AC utilizes the parallelizability of hardware implementation to store and process the transition table in pipeline manner, which eliminates all the cross edges and greatly reduces the memory requirement. P2-AC also utilizes the resource of multiple SRAM blocks in FPGA or ASIC chipset, which guarantees the deterministic line speed throughput and supports incremental updating the pattern set. Moreover, P2-AC gives a simple solution to handle long signatures without affecting the number of stages in pipeline.

3. P2-AC algorithm and architecture

In order to reduce the number of states in DFA state graph, all patterns and input characters are converted to lower case. For case sensitive pattern, a bitmap is used to specify each character of the pattern is lower or upper case, which is used to check with the input stream when the lower case pattern has been matched.

In the DFA, the longest matching substring for the current input is represented by the state value of the current state. Suppose the input stream is "appastxyz". The DFA of Fig. 1 will visit the 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 transits to state <pa>.

The reason for the existence of cross edges is that the starting position of a pattern can be anywhere in the input stream, and the substring of a pattern may be the prefix of another pattern. In the basic AC algorithm, there is only one current state active at one time. When the first character of a pattern arrives and the current state is not <root>, the transition rule corresponding to the cross edge may be accessed later.

It's interesting if multiple threads are used to access the state graph, each time an input character is sent to all the threads, and there is always a new thread beginning from <root>. As shown in Fig. 2, all cross edges can be eliminated from the state graph, which is simplified to a character tree. A thread is terminated when there is no forwarding edge with the input symbol equal to the input character, otherwise, this thread will follow the matched forwarding edge to the next active state on the next level. When an output state is reached, the system will report a pattern is matched by the thread. There is at most one thread on each level.

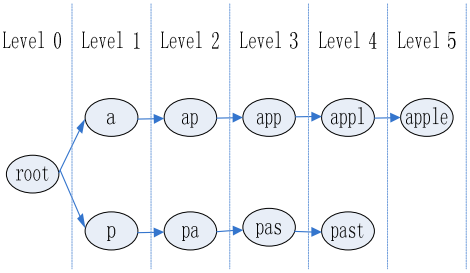

Figure 2 Simplified state tree for Σ = {apple, past}.

In the worst case, on each level of the simplified state tree there is an active state, the number of threads is equal to the height of the tree, which is equal to the length of the longest pattern plus one (plus the <root> node represents empty string). In Snort pattern set, the maximum length is more than 200, it's uneconomical and impractical to track so many threads simultaneously in hardware implementation.

In order to handle long patterns and restrict the maximum number of threads to a constant value K, patterns longer than K are divided into multiple segments of length K except the last segment whose length is equal to or less than K. The pattern matching system is composed of two parts, the first part named pipeline unit (PU) uses at most K threads to match each segment, the second part named aggregation unit (AU) uses the matching result from the first part to aggregate the partial matched segments together to match the whole pattern.

In PU, a character tree is built using the divided segments. The height of the tree is K+1 (plus the <root> node which represents empty segment). Edges representing transition rules are stored in K pipeline stages. The logical format of transition rule in PU is <current state, input character, next state>. A transition rule is stored in stage X if level number of the node in the <next state> field is equal to X. During each cycle, copies of a character from input stream are sent to K stages, at most one active thread exists on one stage, which will use the active state and the input character to search the transition rule table to get the next state and send it to the next stage, if it's an output state, the matched segment ID will be sent to AU.

In AU, a DFA is used to aggregate partial matched segments together to match the whole pattern. The logical format of transition rule in AU is <current state, input segment, next state>. The number of states in DFA of AU is much fewer than the original DFA constructed directly. Transition rules in AU are classified into K groups based on the length of input segment, which is from 1 to K. Each group is stored in a table (T_1 to T_k). In the worst case, each cycle K input

segments with different length are sent to AU, and look up K tables separately. The system need to maintain K current states for each table.

Because a segment with length less than K must be the last segment of a pattern, the search result is the index of a pattern in tables from T_1 to T_{k-1}. Next state (NS) is generated in T_k. In the following i^{th} cycle ($1 \le i < K$), NS will become the current state of T_i.

When looking up the transition rule tables in PU or AU, it's desirable to get the expected transition rule in one memory access cycle. In order to achieve this requirement, transition rules with the same current state are stored in the same address of different SRAMs. The current state ID is used as address to get all the candidate transition rules, which are then compared with the input character or segment to get the expected matching result. The number of SRAMs for each stage is equal to maximum transition rules with the same current state. The special case is that transition rules in PU or AU with <root> as current state are stored in one SRAM, which is indexed by input character or carefully assigned segment ID.

The architecture of P2-AC is shown in Fig. 3. Assuming K=4, in each cycle, one character from the input stream is sent to PU. PU is implemented by 4 pipeline stages. Transition rules are stored separately in tables from LT_1 to LT_4. The active current state is maintained by each stage. Related transition rules in one stage are extracted and compared with the input character. The output to the next stage is the next state ID, the output to AU is the matched segment ID, the output to Pattern store is matched short pattern ID. By carefully assignment, the three IDs can be the same value.

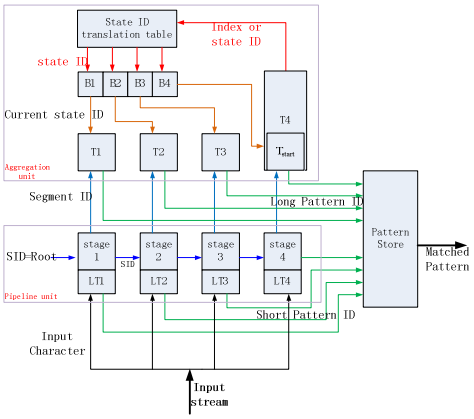

Figure 3 Architecture of P2-AC

In AU, transition rules are stored separately in tables from T_1 to T_4. Input segment ID is sent to T_i if the length of the segment is equal to i ($1 \le i \le 4$). T_{start} is a suitable of T_4, which stores the first segment for each long pattern. Active current state for each table is maintained in current state registers, which are updated by state ID translation table (STT) each cycle. STT is used to record the transition rules' addresses for an active state in tables from T_1 to T_4. The index of the active state is provided by T_4. In order to reduce the size of STT, if an active state only exists in one table, it will not be stored in STT but directly provided by T_4. Related transition rules for the current active state is extracted from the table and compared with the input segment ID. The output of T_i ($1 \le i \le 4$) to pattern store is the long pattern ID. The output of T_4 to STT is the index of next active state existing in multiple tables, or the transition rules' address of the next active state in one table.

Let's consider the structure of transition rule table in each stage of PU. According to the statistics of patterns in SNORT rules, the number of child nodes for one state is less than 50 and usually 1 or 2, except for <root>; the number of child nodes of <root> is a large one. Transition rules in stage1 are stored in one SRAM which is indexed by the ASCII code of the input character. Transition rules on other stages are stored in parallel SRAMs, and the State ID is assigned as address to locate all the related transition rules, which are read into registers and compared with the input character.

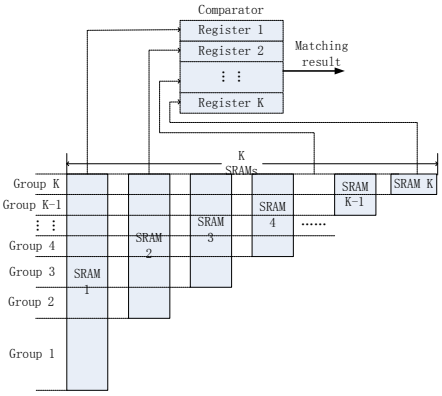

Figure 4 Structure of transition rule table

As shown in Fig. 4, transition rules with the same current state are stored at the same address in multiple SRAMs. The number of SRAMs in one stage is determined by the maximum number of transition rules with the same current state, which is called maximum fan-out number in this paper. According to the statistics of SNORT rules, there are only a few states with a large fan-out number less than 56, while most of the states' fan-out number is equal to 1 or 2. The sizes of the parallel SRAMs are not the same. Transition rules are grouped into multiple groups and arranged in descending order based on the fan-out number of the current state. By choosing the size of the SRAMs, there are unallocated entries between the groups, which can

be used to support incremental update when new patterns are added into the rule set.

Because all the transition rules in LT_1 have the same current state <root>, they are all stored in one SRAM. As shown in Fig. 5, the SRAM has 256 entries. Transition rule use the ASCII code of the input Character as its address in the SRAM. When looking up LT_1, the input character will be used as index to locate the related transition rule in the SRAM.

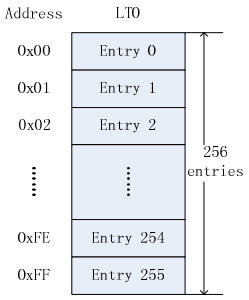

Figure 5 Transition rules in LT1 of PU

Transition rules in AU are classified into K tables (K=4) according to the length of the input segment. Similar to the structure of tables in PU, transition rules in one table in AU are also stored in parallel SRAMs as shown in Fig. 4. The main difference is the next matching position among tables in AU is not shifted one by one, but determined by the length of the next segment sent from PU. State register B_i in Fig. 3 are used to record the active current state for T_i in AU, which will be updated by STT in the next cycle. Transition rules with the same current state may not be stored at the same address among different tables. As shown in Fig. 6, STT is used to record the addresses for one state in different tables in AU. If the state only exists in one table, its address in that table will not be stored in STT but provided directly by T_4. Otherwise, T_4 will provide the index of the state to access STT. This operation can be pipelined, so it will not slow down the processing speed of the system.

For some transition rules in T_4 whose current state is <root>, they are all stored in one SRAM. When assigning segment IDs, segments in the transition rules with <root> as their current state are assigned values from 0 to M-1 (M is the number of the transition rules whose current state is <root>). Then if the current state of T_4 is equal to <root>, segment ID is used as index to locate the related transition rule in T4. The other transition rules in T_4 are stored in parallel SRAMs as shown in Fig. 4.

State Index	Address in T1	Address in T2	Address in T3	Address in T4
0	a1	a2	a3	a4
1	b1	Null	Null	b4
2	Null	c2	Null	c4
3	Null	Null	d3	d4
4	e1	e2	Null	e4
5	Null	f2	f3	f4
6	g1	Null	g3	g4
⋮	⋮			

Figure 6 The format of State ID translation table (STT)

In order to support concurrent accesses, pattern store uses 2K SRAMs to store patterns. When K=4, the number of SRAMs in pattern store is 8.

Let's consider an example where the pattern set is {apple, applause, ampliation, past, pat, parable} and the input stream is "appampliation".

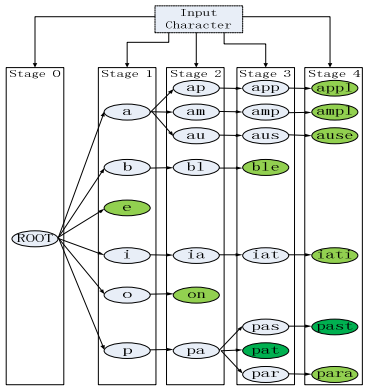

Figure 7 Logical organization in PU

As shown in Fig. 7, the patterns are divided into 10 segments when K=4. {pat, past} are patterns shorter than 4, which can be directly matched. {appl, e, ause, ampl, iati, on, para, ble} are segments, which will be sent to AU for aggregation. As shown in Fig. 8, segments of one pattern are aggregated to match the whole pattern. {apple, applause, ampliation, parable} are patterns longer than 4, which are matched in AU.

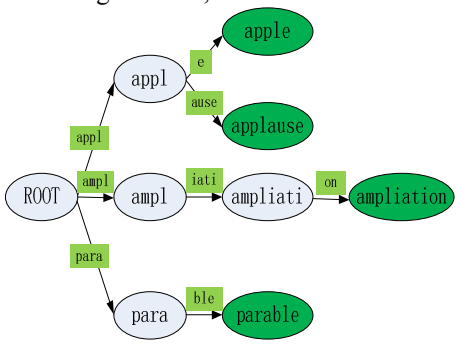

Figure 8 Logical organization in AU

Considering the physical organization in PU, LT_1 uses one SRAM to store transition rules to nodes on

level 1, which is indexed by the ASCII code of the input character. LT_2, LT_3 and LT_4 use parallel SRAMs to store the transition rules to the nodes on the corresponding levels, which are indexed by the current state IDs. As shown in Fig. 9, segment {e} can be matched in LT_1, segment {on} can be matched in the first SRAM of LT_2, segment {ble} can be matched in the first SRAM of LT_3, pattern {pat} can be matched in the second SRAM of LT_3, segments {appl, ampl, ause, iati, para} can be matched in the first SRAM of LT_4 and pattern {past} can be matched in the first SRAM of LT_4.

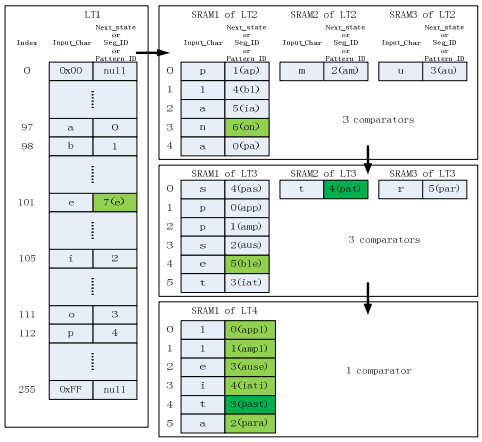

Figure 9 Physical organization in PU

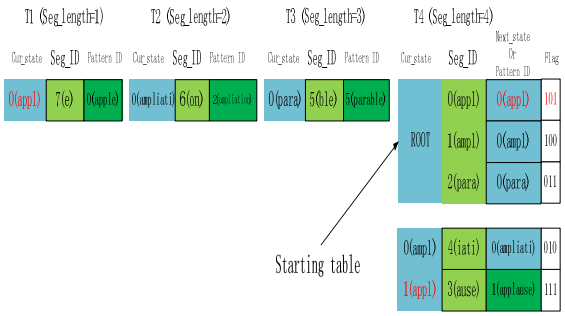

Figure 10 Physical organization in AU

Considering the physical organization in AU, transition rules with <root> as their current state are stored in one SRAM in T_4, which is called starting table and indexed by the pre-assigned input segment IDs. The other transition rules in T_1, T_2, T_3 and T_4 are stored in parallel SRAMs. As shown in Fig. 10, transition rules <root, appl, appl>, <root, ampl, ampl> and <root, para, para> are stored in the starting table in T_4; transition rules <appl, ause, **applause**> and <ampl, iati, ampliati> are stored in the other part of T_4; transition rule <appl, e, **apple**> is stored in T_1; transition rule <ampliati, on, **ampliation**> is stored in T_2; transition rule <para, ble, **parable**> is stored in T_3. The bold states in transition rules are matched patterns.

In Fig. 10, state {appl} both exists in address 0 of T_1 and in address 1 of T_4. We use one entry in STT to record the address mapping relationship for state {appl}.

To support accessing pattern store concurrently, patterns are stored in multiple SRAMs, which are one to one mapping relationship with tables in PU and AU.

Assuming the input stream is {appampliation}, in the 7th cycle, segment {ampl} is matched in PU and state {ampl} is matched in AU. In the 11th cycle, segment {iati} is matched in PU and state {ampliati} is matched in AU. In the 13th cycle, segment {on} is matched in PU and pattern {ampliation} is finally matched in AU.

For one type of transition rules, whose output represents more than one meaning, such as next state ID, segment ID or pattern ID, we assign these IDs with the same value and use three bits to indicate the effective meanings of the stored value. Because the numbering space in different tables is independent, the ID allocating strategy is reasonable and memory efficient.

4. Performance evaluation

We extract 5669 distinct patterns from the Snort V2.8 rule set [19]. The signatures are converted into lower case letters. The total character count is 79211. The maximum pattern length is 109 characters and the average length is about 14 characters. About 96% of the signatures have no more than 36 characters.

Our evaluation indicates that better memory efficiency is obtained when the segment length is in the range of 4 to 6. As shown in Table 1, for k=4, the number of entries in the lookup tables for the pipeline unit and the DFA unit are 18599 and 15876, respectively. The width of the state ID for the pipeline unit and the DFA unit are 15 bits and 14 bits, respectively. Because LT_1 and T_{start} are indexed by input character and input segment ID respectively, one SRAM with width 18 bits and length 256 is allocated to LT_1, another SRAM with width 20 bits and length 2561 is allocated to T_{start}. STT stores the addresses for states existing in multiple tables in AU, one SRAM with width 45 bits and length 376 is allocated to STT. Other tables in PU and AU employ parallel SRAMs to store transition rules with the same current state at the same address, which are indexed by the current state.

Table 1 Allocation of Snort rule segments

K=4	LT_1	LT_2	LT_3	LT_4		
# of Entries in PU	208	2092	6811	9494		
Entry Width	18	26	26	26		
# of SRAMs	1	50	29	18		
# of	0	50	29	18		

Comparators						
Comparator Width		8	8	8		
K=4	T_1	T_2	T_3	T_4	T_{start}	STT
# of Entries in AU	1143	1094	974	9432	2622	379
Entry Width	35	35	35	35	20	45
# of SRAMs	6	4	5	31	1	1
# of Comparators	6	4	5	31	0	0

The resource cost of SRAMs and logic registers in FPGA is shown in Table 2. In all, the Snort pattern set costs 305 M512 SRAM blocks and 228 M4K SRAM blocks, which are about 133KB and 13.68 bits per character for the utilized Snort pattern set. Logic registers are used to implement the parallel comparators for the lookup tables, which are 4132 LUTs in FPGA and about 0.05 LUTs per character. We use Altera's StratixII EP2S60 FPGA to implement the pattern matching system using P2-AC method. The resource cost for the Snort pattern set used is about 42.8% SRAM bits and 8.5% LUTs in EP2S60 FPGA.

Table 2 SRAMs and logic Cost in FPGA

K=4	M512	M4K	M-RAM	LUTs
LT_1	3	1	0	0
LT_2	118	2	0	1300
LT_3	68	37	0	754
LT_4	37	57	0	468
T_{start}	5	12	0	0
T_1	13	9	0	210
T_2	7	8	0	140
T_3	8	8	0	175
T_4	44	90	0	1085
STT	2	4	0	0
SRAM blocks	305	228	0	4132
EP2S60	329	255	2	48352
Total cost	1,090,048 bits			4132
EP2S60	2,544,192 bits			48352
Percentage	42.8%			8.5%
Cost per char	13.68 bits/char			0.05 LUTs/char

Performance comparison with 5 other AC-based methods is given in Table 3. The memory cost of P2-AC, BFPM [5] and the TCAM-based method of [13] are evaluated using the same signature set. When the given signature set is divided into 8 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 [13], we assume state transitions are based on 2 input characters per cycle. The hardware cost of bit-split FSM [3] and split-AC [4] are quoted from [4]. The hardware cost of CDFA are quoted from [12]. 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 [3]. Hence, the actual hardware cost for the 2 sets of implementation parameters shown in Table 3 are more or less the same.

From Table 3, we can see that P2-AC has a clear performance advantage over the other methods. The memory cost of P2-AC is lower than BFPM and CDFA by 85% and 47%, respectively. Moreover, the throughput of P2-AC is two times that of CDFA. In general, methods such as BFPM and CDFA that divide the signature set into a larger number of subsets will be in a disadvantage position when considering FPGA implementation. In these methods, the system will have 8 to 16 large lookup tables of roughly equal size. Even though the total memory requirement may be smaller than the memory capacity of a FPGA, but the required memory modules may not fit the size of the RAM blocks available on the device. As a result, a higher-end more expansive device needs to be used.

Table 3 Comparison with other AC-based methods

Method	Pattern 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)	80K	90 bits	simple	1
TCAM (2char/cycle)	80K	56 bits TCAM + 27 bits SRAM	simple	2 to 4 (max 266MHz clk)
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)
P2-AC	80K	13.68 bits	4132LUTs	2

The control logic for P2-AC is very simple. The pipelining of output results from one stage to the next stage can be realized by simple clocked-register. LT1 is

a 256-entry table indexed by the input character. T_{start} is a table indexed by the input segment ID. Other tables are implemented by parallel SRAMs indexed by the current state ID, related transition rules are extracted and compared together to get the matched result. By carefully assignment, the next state ID, segment ID and pattern ID represented by one transition rule can be the same value.

5. Conclusion

A pipelined approach with parallel SRAMs for hardware implementation of the Aho-Corasick algorithm is presented. The system maintains multiple threads that traverse the automaton concurrently so that only forward edges needed to be stored in the state graph. In contrast to previously published heuristic-based methods, state graph reduction in P2-AC is guaranteed algorithmically. This is a definite advantage that ensures scalability of the method to handle the fast expanding signature set for network intrusion detection. Parallel SRAMs are used to store and access transition rules efficiently. Incremental update is also supported by P2-AC, patterns can be added into or deleted from the pattern set within one cycle. The memory cost of P2-AC is as low as 13.68 bits/char for a signature set with 5.7K strings which is less than 47% 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 two-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 a little overhead.

6. References

[1] Alfred V. Aho and Margaret J. Corasick, Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM, 1975. 18(6): p. 333-340.

[2] Nathan Tuck, et al. Deterministic memory-efficient string matching algorithms for intrusion detection. 2004. Hongkong, China: Institute of Electrical and Electronics Engineers Inc., Piscataway, NJ 08855-1331, United States.

[3] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. 2005. Madison, WI, United States: Institute of Electrical and Electronics Engineers Inc., New York, NY 10016-5997, United States.

[4] I. Papaefstathiou V. Dimopoulos, D. Pnevmatikatos, A memory-efficient reconfigurable Aho-Corasick FSM implementation for intrusion detection systems, in IEEE IC-SAMOS. 2007. p. 186-193.

[5] Jan Van Lunteren. High-performance pattern-matching for intrusion detection. 2006. Barcelona, Spain: Institute of Electrical and Electronics Engineers Inc., Piscataway, NJ 08855-1331, United States.

[6] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on FPGAs. 2004. Monterey, CA., United States: Association for Computing Machinery.

[7] Zachary K. Baker and Viktor K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. 2004. Napa, CA, United States: IEEE Computer Society, Los Alamitos, CA 90720-1314, United States.

[8] Long Bu and John A. Chandy. FPGA based network intrusion detection using content addressable memories. 2004. Napa, CA, United States: IEEE Computer Society, Los Alamitos, CA 90720-1314, United States.

[9] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. 2004. Napa, CA, United States: IEEE Computer Society, Los Alamitos, CA 90720-1314, United States.

[10] Christopher R. Clark and David E. Schimmel. Scalable pattern matching for high speed networks. 2004. Napa, CA, United States: IEEE Computer Society, Los Alamitos, CA 90720-1314, United States.

[11] Derek Pao, Wei Lin, and Bin Liu, Pipelined Architecture for Multi-String Matching. Computer Architecture Letters, 2008.

[12] Tian Song, Wei Zhang, Dongsheng Wang, Yibo Xue, A Memory Efficient Multiple Pattern Matching Architecture for Network Security, in IEEE INFOCOM 2008. 2008.

[13] M. Alicherry, M. Muthuprasanna, and V. Kumar, High Speed Pattern Matching for Network IDS/IPS, in IEEE ICNP. 2006. p. 187-196.

[14] Young H. Cho and William H. Mangione-Smith. A pattern matching co-processor for network security. 2005. Anaheim, CA, United States: Institute of Electrical and Electronics Engineers Inc., Piscataway, NJ 08855-1331, United States.

[15] Giorgos Papadopoulos and Dionisios Pnevmatikatos. Hashing + Memory = low cost, exact pattern matching. 2005. Tampere, Finland: Institute of Electrical and Electronics Engineers Computer Society, Piscataway, NJ 08855-1331, United States.

[16] Ioannis Sourdis, Dionisios N. Pnevmatikatos, and Stamatis Vassiliadis, Scalable multigigabit pattern matching for packet inspection. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2008. 16(2): p. 156-166.

[17] Sarang Dharmapurikar, et al., Deep packet inspection using parallel bloom filters. IEEE Micro, 2004. 24(1): p. 52-61.

[18] Haoyu Song, et al. Snort offloader: A reconfigurable hardware NIDS filter. 2005. Tampere, Finland: Institute of Electrical and Electronics Engineers Computer Society, Piscataway, NJ 08855-1331, United States.

[19] SNORT network intrusion detection system, http://www.snort.org