International Journal of Computer and Communication Technology

Volume 7 | Issue 2 Article 3

April 2016

Fast and Scalable Pattern Matching for Memory Architecture

J. Santhi
Department of Electronics and Communication Engineering, Nalanda Institute of Engineering and Technology, Sattenapalli, jagarlamudi_santhi@yahoo.com

L. Srinivas
Department of Electronics and Communication Engineering, Nalanda Institute of Engineering and Technology, Sattenapalli, srinivas_lanki@yahoo.com

Fast and Scalable Pattern Matching for Memory Architecture

J. Santhi & L. Srinivas

Department of Electronics and Communication Engineering,
Nalanda Institute of Engineering and Technology, Sattenapalli
E-mail : jagarlamudi_santhi@yahoo.com, srinivas_lanki@yahoo.com

Abstract – Multi-pattern matching is known to require intensive memory accesses and is often a performance bottleneck. Hence specialized hardware-accelerated algorithms are being developed for line-speed packet processing. While several pattern matching algorithms have already been developed for such applications, we find that most of them suffer from scalability issues. We present a hardware-implementable pattern matching algorithm for content filtering applications, which is scalable in terms of speed, the number of patterns and the pattern length. We modify the classic Aho-Corasick algorithm to consider multiple characters at a time for higher throughput. Furthermore, we suppress a large fraction of memory accesses by using Bloom filters implemented with a small amount of on-chip memory. The resulting algorithm can support matching of several thousands of patterns at more than 10 Gbps with the help of a less than 50 KBytes of embedded memory and a few megabytes of external SRAM.

Keywords *- Aho–Corasick (AC) algorithm, finite automata, pattern matching, Content Filtering, Pattern Matching, Network Intrusion Detection, Bloom Filters.*

I. INTRODUCTION

Signature-based NID(P)S looks for the presence of the predefined signature strings deemed harmful to the network such as an Internet worm or a computer virus in the payload. In some cases the starting location of such predefined strings can be deterministic. For instance, the URI in a HTTP request can be spotted by parsing the HTTP header and this precise string can be compared against the predefined strings to switch the packet. In certain cases one doesn't know where the string of our interest can start in the data stream making it imperative for the system to scan every byte of the payload. This is typically true of the signature based intrusion detection systems such as Snort [1]. Snort is a light-weight NIDS which can filter packets based on predefined rules.

Each Snort rule first operates on the packet header to check if the packet is from a source or to a destination network address and/or port of interest. If the packet matches a certain header rule then its payload is scanned against a set of predefined patterns associated with the header rule. Matching of one or multiple patterns implies a complete match of a rule and further action can be taken on either the packet or the TCP flow. The number of patterns can be in the order of a few thousands. Snort version 2.2 contains over 2000 strings.

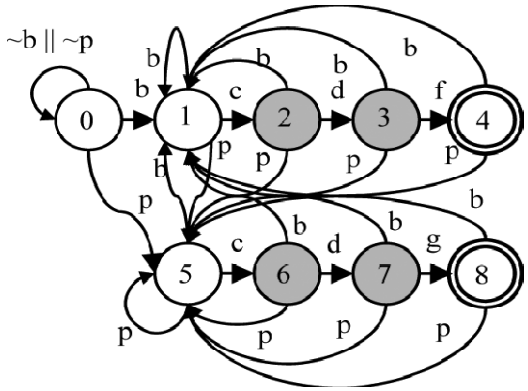


Fig 1. DFA for matching "bcdf" and "pcdg".

Several interesting pattern matching techniques for network intrusion detection have been developed, a majority of them produced by the FPGA community. The Main Purpose of a signature-based network intrusion detection system is to prevent malicious network attacks by identifying known attack patterns. Due to the increasing complexity of network traffic and the growing number of attacks, an intrusion detection system must be efficient, flexible and scalable. The primary function of an intrusion detection system is to

perform matching of attack string patterns. Because string matching is the most computative task in network intrusion detection (NIDS) systems, many hardware approaches are proposed to accelerate string matching. The hardware approaches may be classified into two main categories, the logic and the memory architectures.

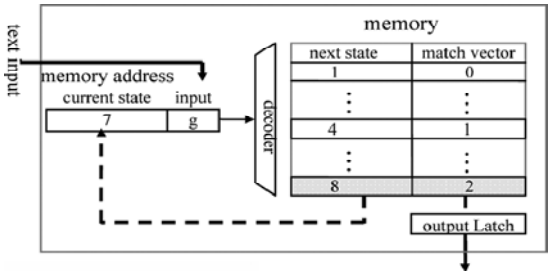


Fig 2. Basic memory architecture

Fig. 2 presents a simple memory architecture to implement the FSM. In the architecture, the memory address register consists of the current state and input character; the decoder converts the memory address to the corresponding memory location, which stores the next state and the *match vector* information.

A "0" in the match vector indicates that no "suspicious" pattern is matched; otherwise the value in the matched vector indicates which pattern is matched.

For example in Fig. 2, suppose the current state is 7 and the input character is . The decoder will point to the memory location which stores the next state 8 and the match vector 2. Here, the match vector 2 indicates the pattern "pcdg" is matched.

II. RELATED WORK

In the past few years, several interesting algorithms and techniques have been proposed for multi-pattern matching in the context of network intrusion detection. The hardware-based techniques make use of commodity search technologies such as TCAM [1] or reconfigurable logic/FPGAs [6][1][3][7] Some of the FPGA based techniques make use of the on-chip logic resources to compile patterns into parallel state-machines or combinatorial logic.

An approach presented in [5] uses FPGA logic with embedded memories to implement parallel Pattern Detection Modules (PDMs). PDMs can match arbitrarily long strings by segmenting them in smaller substrings and matching them sequentially.

A Bloom-filter based algorithm proposed in [8] makes use of a small amount of embedded-memory along with commodity offchip memory to scan a large number of strings at high speed. Using on-chip Bloom filters, a quick check is done on the payload strings to see if it is likely to match a string in the set. Upon a Bloom filter match, the presence of the string is verified by using a hash table in the off-chip memory. The authors argue that since the strings of interest are rarely found in the packets, the quick check in Bloom filter reduces more expensive memory accesses and improves the overall throughput greatly.

However, since the algorithm involves hashing over a maximum length pattern size text window, it does not scale for arbitrarily long strings (100s of bytes). It is reported that up to 16 bytes is a feasible pattern length for a high-speed implementation. As we will see, our algorithm combines the techniques in [8] with Aho-Corasick algorithm to get rid of the string length limitation.

III. AHO-CORASICK ALGORITHM

In multi-string matching problem, we have a set of strings S and we would like to detect all the occurrences of any of the strings in S in a text stream T. We will denote by T[i...j] the character sequence from ith character to jth character of stream T. For a given set of strings, the Aho-Corasick algorithm constructs a finite automaton.

This finite automaton can be a Deterministic Finite Automaton (DFA) or a Non-deterministic Finite Automaton (NFA). For our purpose, we will focus on NFA version of the algorithm since that is the one we will improve upon. Otherwise, it makes a failure transition. In case of a failure transition the machine must reconsider the character causing the failure for the next transition and the same process is repeated recursively until the given character leads to a non-failure transition. The first fundamental problem Aho-Corasick algorithm suffers from is a high memory access requirement.

At least one memory access is needed to read the state node on each input character. Furthermore, the sequential failure transitions can cause more memory accesses. In the worst case, the average number of memory accesses required per input character is two. Therefore, given the high latency and slow speed of commodity memory chips, using them to implement Aho-Corasick algorithm can severely degrade the throughput of the system.

The second problem we observe in the regular Aho-Corasick algorithm is that it can not be readily parallelized. Hence, we are forced to consider only one character at a time from the text stream no matter how much logic and memory resources are available. The processing of one character per clock cycle of the system clock can create a bottleneck for high speed networks

IV. HARDWARE ARCHITECTURE

We now describe the hardware architecture we assume for implementing the state machine. The architecture consists of a TCAM, static RAM (SRAM) and a logic. Each TCAM entry represents a certain transition in the state machine, and has a corresponding memory block (structure) in the SRAM whose address can be computed from the TCAM index. We logically partition the TCAM entries into two fields: *current state* and *input*. If the state machine transitions from state s1 to state s2 on input a, then the TCAM contains an entry (s1, a) and the corresponding entry in SRAM contains s2. If the state s2 corresponds to one or more keywords, then the SRAM entry also contains pointers to those keywords.

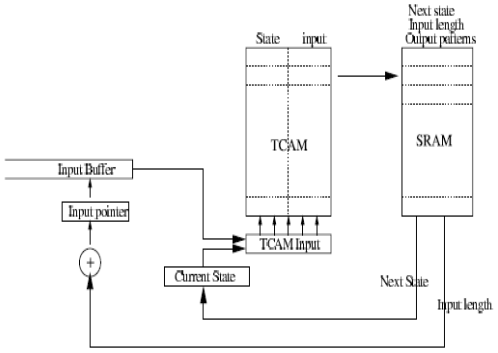


Fig 3. Hardware Architecture

Figure 3 presents the details of the hardware architecture. In this paper, we assume the existence of a standard flow classification hardware module that takes care of identifying the packets of a flow, sequencing etc. The pattern matching hardware module runs a unique state machine instance for each flow. This is essential to detect patterns spread across multiple packets in a flow. The incoming packets for the current flow are stored in the input buffer. For each flow, we store two pieces of information, namely the current state in the state machine and a pointer to the next input (character) to be fed to the state machine.

Search Speed Enhancement

The techniques described thus far implement the Aho- Corasick state machine, while processing only one input character per TCAM lookup. As this does not scale to high speeds required today, we now propose techniques to achieve greater speed-up using the same architecture.

Consider the state machine in Figure 4. This is functionally similar to the state machine in Figure 1, except that the transitions are now on four characters each. We call this state machine, a *multi-character (compressed) state machine.*

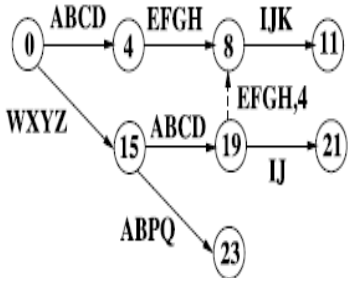


Fig. 4.Compressed AC DFA (Not all *next* transitions shown)

This state machine can be implemented using our proposed architecture with the following changes: the TCAM entries now contain four characters in the input field requiring 32 bits for their representation, and the input pointer is now incremented by 4 for every state transition (i.e every TCAM lookup). Hence we get a speedup of upto four, provided the input bus to the TCAM is wide enough (which is achievable when implemented in custom hardware).

However, it might not always be possible to make state transitions on the same number of characters (eg. 4 in the above example), and hence we have another field called *length*, in the SRAM corresponding to each transition. Henceforth, we refer to the maximum number of input characters that are placed in a single TCAM entry as *transition width* and denote it by k. Additionally, we need to "synchronize" the input with the state machine to account for the offset at which a pattern might occur in the input stream, and use additional shallow states and transitions to ensure correctness.

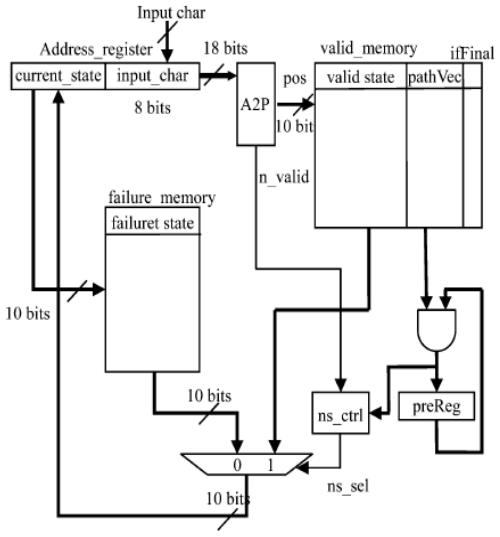


Fig 5. Hardware Module for new Algorithm

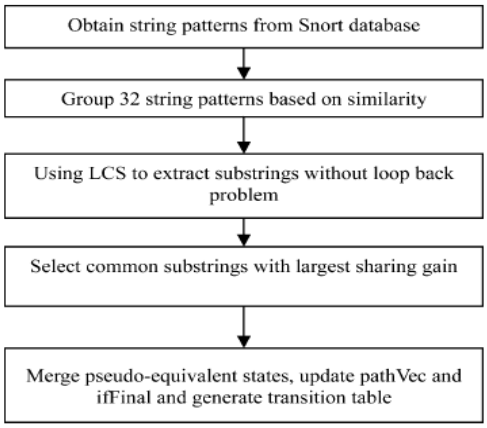


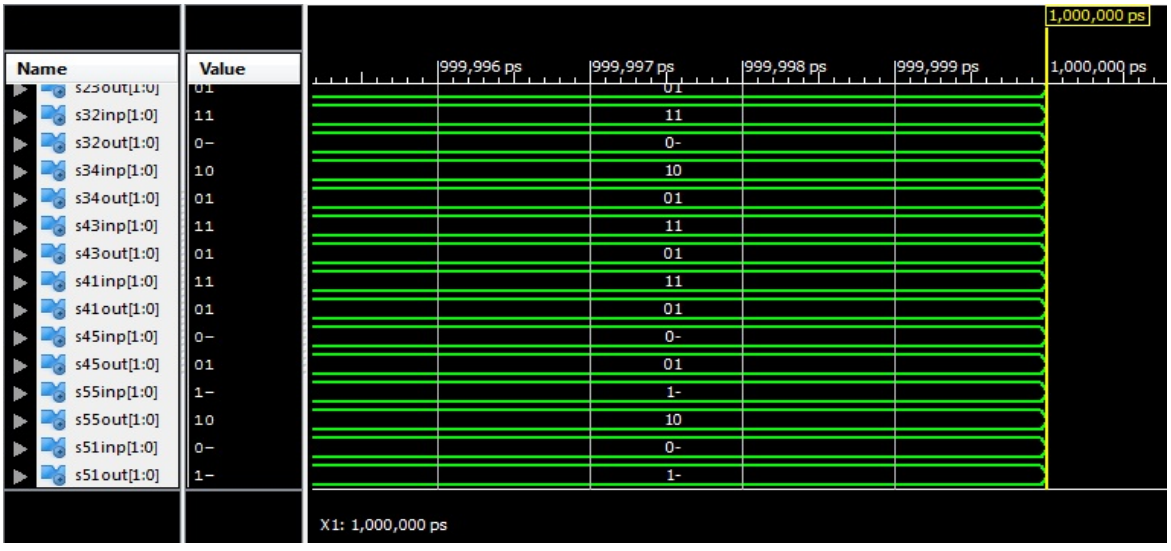
Fig 6. Flow Chart Of Experiments

V. EXPERIMENTAL RESULTS

The results are compared with the methods of the AC algorithm and the bit-split algorithm. The flow of our experiment is shown in Fig. 6. In the first stage, we obtain string patterns from Snort rule database. In the second stage, we group 32 string patterns as a module based on the similarity of string patterns. Further, in the third stage, we use LCS to extract substrings without loop back problem. Because the solution of LCS may not be unique, we select the common substrings which have the largest *sharing gain*.

TABLE I
EXPERIMENTAL RESULTS AFTER APPLYING OUR ALGORITHM TO THE AC ALGORITHM.

Rule Sets	# of patterns	# of char.	Aho-Corasick			Aho-Corasick + Our algorithm			
			# of transitions	# of states	Memory (bytes)	# of transitions	# of states	Memory (bytes)	Memory reduction
Oracle	337	11,128	6,793	6,804	49,267	4,432	3,846	30,699	38%
Deleted	310	4,636	3,241	3,251	22,702	2,541	2,322	17,776	22%
Exploit	160	2,052	1,561	1,567	10,545	1,234	1,135	8,348	21%
Web-misc	156	1,644	1,420	1,425	9,592	1,305	1,247	8,892	7%
Smtp	104	989	715	719	4,653	625	600	4,109	12%
Misc	97	1,403	1,133	1,137	7,653	897	820	5,846	24%
Ftp	96	466	402	406	2,517	385	374	2,442	3%
Other sets	957	14,041	10,912	10,871	81,768	9,345	8,777	70,262	14%
Total rules	**2,217**	**36,359**	**26,177**	**26,180**	**188,697**	**20,764**	**19,121**	**148,374**	**21%**
Reduction (%)					1			**21%**	



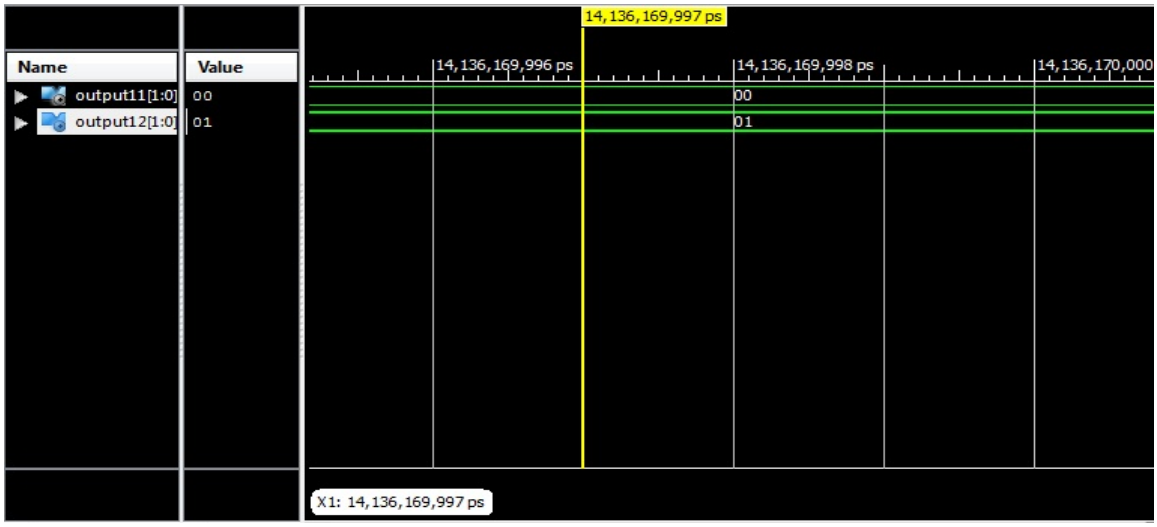


Fig.7 Simulation Results

The sharing gain of common substrings is defined as the length of common substrings multiplied by the number of patterns sharing the common substrings. For example, three patterns, "1common1", "2common2", and "3common3" have the common substrings "common". The sharing gain of the common substrings is 6*3=18 because the substring "common" has six characters which are shared by three patterns. In the final stage, we merge the extracted common substrings and generate the transition table.

Table I shows the results before and after integrating our algorithm to the AC algorithm. Columns one, two and three show the name of the rule set, the number of patterns, and the number of characters of the rule set. Column ten shows the memory reduction compared to the AC algorithm. As shown in Fig. 5, the memory requirement includes the size of the valid memory and the failure memory.

VI. CONCLUSIONS

In this paper, we propose a novel multiple string matching algorithm that uses multi-character transitions on a finite state automata to increase the throughput, and also leverages a clever transition optimization technique to reduce the memory requirements. We describe a TCAM-based hardware architecture to realistically achieve these higher data rates for virus/worm detection employing signature matching. Additionally, packet inspection in the network is essential for various applications including QoS monitoring, bandwidth metering, stateful packet filtering etc.Our simulation results demonstrate that the proposed algorithm indeed scales well in practice to meet the current day requirements, as tested on real virus signature databases.

VII. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their comments which were very helpful in improving the quality and presentation of this paper.

REFERENCES:

[1] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to ddos attack detection and response," in DISCEX, 2003.

[2] L. Spitzner, Honeypots: Tracking Attackers. Addison-Wesley, 2002. C.Morrow http://www.secsup.org/Tracking.

[3] BlackHole Route Server and Tracking Traffic on an IP Network.

[4] http://www.snort.org. SNORT: Open-Source Network IDS/IPS.

[5] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," Commun. ACM, vol. 18, no. 6, pp. 333–340, 1975. D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," SIAM J. Comput., vol. 6, no. 2, pp. 323–350, June 1977.

[6] R. S. Boyer and J. S. Moore, "A fast string matching algorithm," Commun. ACM, vol. 20, no. 10, pp. 762–772, October 1977.

[7] D. M. Sunday, “A very fast substring search algorithm,” Commun. ACM, vol. 33, no. 8, pp. 132–142, August 1990.

[8] M. Crochemore and D. Perrin, “Two-way string matching,” J. ACM,vol. 38, no. 3, pp. 650–674, 1991

[9] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” IBM J. Res. Dev., vol. 31, no. 2, pp. 249–260, 1987. Z. Galil and J. Seiferas, “Time-space optimal string matching (preliminary report),” in STOC, 1981.

[10] N. T. et.al., “Deterministic memory-efficient string matching algorithmsfor intrusion detection,” INFOCOM, 2004

Authors Profile:

Ms. J.Santhi is pursuing her M.Tech in Nalanda institute of engineering and technology with specialization embedded systems.

L.Srinivas is an assistant professor and Head of the ECE department in nalanda institute of engineering and technology. He got his M.Tech from Narasaraopet Engineering College in 2010.His Area of interest is communication field

◈◈◈