Design of High Performance Pattern Matching Engine Through Compact Deterministic Finite Automata

Piti Piyachon and Yan Luo
Dept. of Electrical and Computer Engineering, University of Massachusetts Lowell
Lowell, MA, USA
piti_piyachon@student.uml.edu, yan_luo@uml.edu

ABSTRACT

Pattern matching relies on deterministic finite automata (DFA) to search for predefined patterns. While a bit-DFA method is recently proposed to exploit the parallelism in pattern matching, we identify its limitations and present two schemes, *Label Translation Table* (LTT) and *CAM-based Lookup Table* (CLT), to reduce the DFA memory size by 85%, and simplify the design by requiring only four processing elements of bit-DFA instead of thousands.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: microprocessor/microcomputer applications; C.4 [**Performance of Systems**]: design studies; C.1.4 [**Parallel Architectures**]: Distributed architectures

General Terms

Algorithms, Design, Performance, Security

Keywords

Pattern Matching, Deterministic Finite Automata, Content Addressable Memory

1. INTRODUCTION

Pattern matching is a critical task for numerous applications such as network monitoring, network intrusion detection (NID), virus detection, etc. The matching procedure searches for predefined patterns in data stream. Growing of pattern sets (e.g. Snort NID) makes matching become one of the most challenging tasks.

Pattern matching usually relies on a deterministic finite automaton (DFA) stored in memory to provide deterministic performance and flexibility to update pattern sets. The size of a DFA is normally in the order of tens of megabytes for a realistic pattern set (e.g. Snort [8]), thus a DFA cannot be easily fitted in on-chip memory to achieve high speed.

DAC 2008, June 8–13, 2008, Anaheim, California, USA

Recently, two bit-DFA models for pattern matching are proposed to reduce the size of a DFA, and exploit the parallelism with parallel processing resources in ASIC and FPGA [7, 11].

In this paper, we investigate the key issues related to bit-DFA based pattern matching, and give solutions. The bit-DFA method has not been realized under current technology due to its requirement of thousands of processing elements and demanding memory usage. We attack the root cause of the issues by eliminating bit-vectors [10] used for identifying patterns in bit-DFA. Specifically, we propose two schemes, Label Translation Table (LTT) and CAM-based Lookup Table (CLT), to identify the matched patterns. Our experiment results show that the memory required to implemented bit-DFA are reduced by 85%, making them possible to fit in precious on-chip memory space. The required processing elements are reduced from thousands to four.

The rest of this paper is organized as follows. A classical pattern matching algorithm is introduced in Section 2 along with the bit-DFA method. Section 3 discusses the problems of the bit-DFA pattern matching. We propose our solutions in Section 4, and an architecture supporting the solutions in Section 6. Our experimental results are presented in Section 7. Related works are discussed in Section 8. Finally, the paper is concluded in Section 9.

2. BACKGROUND

Aho-Corasick (AC) algorithm [1] is a classical pattern searching algorithm that builds DFA. Fig. 1 shows a DFA diagram for an example set of patterns {his, her, man, lady}. Each double-circle node is an output state announcing the match of a pattern. A directed edge shows the state transition when matching a byte. The pattern matching procedure scans an input data stream byte by byte, and migrates from the start state (state 0) until all bytes are taken. During the migration, reaching an output state indicates a pattern match in the input data stream.

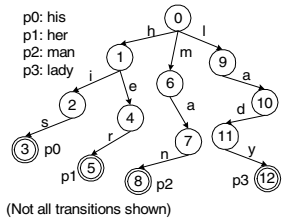

Figure 1: An example of a DFA diagram

Different granularity of parallelism in pattern matching was explored in [6] and [10]. Matching a byte from the input stream at a time can be turned to matching four double-bits in parallel. The DFA example in Fig. 1 becomes four bit-DFA, which are shown in Fig. 2. The construction of bit-DFA also uses the classical AC algorithm although the granularity is now at bit-level. For example, the bit 1 and 0 of ASCII characters 'h', 'i' and 's' are '00', '01' and '11', respectively, thus the bit-1_0 sequence of "his" is "00, 01, 11". Similarly the bit-1_0 sequences of "her", "man" and "lady" are "00, 01, 10", "01, 01, 10", "00, 01, 00, 01", respectively. These four bit-1_0 sequences are used to construct the bit-DFA 0 shown in Fig. 2. We can construct other bit-DFA in the same way. When an input byte comes in, these four double-bits will be extracted from the byte, and used to search on the corresponding bit-DFA in parallel. *A match is claimed iff all bit-DFA find the same match.*

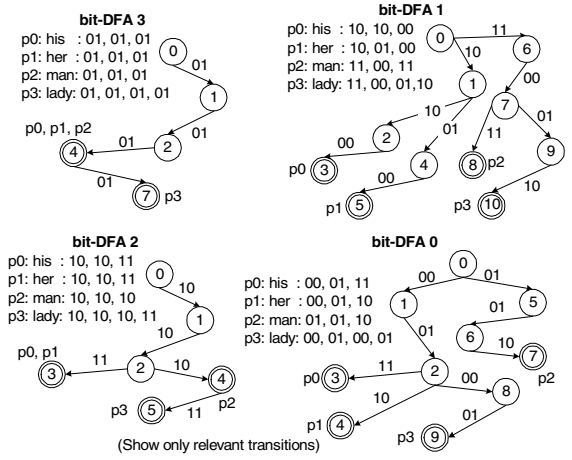

Figure 2: 4 bit-DFA are equivalent to the DFA in Fig. 1

The advantage of the bit-DFA method is the reduction of the alphabet size from 256 characters to 16, 4, or 2, depending on the amount of bits examined at a time. Hence, the amount of possible next states is less, resulting less memory needed to store next state pointers. It has been observed that the alphabet size of 4 is optimum [6, 11]. Thus for the rest of this paper, we use the alphabet size of **4** in our analysis and design.

3. MOTIVATION

Despite of tremendously decreasing the memory requirement, pattern matching with the bit-DFA method has not been deployed due to several reasons.

Implementation Issues: The bit-DFA method requires thousands of processing elements to implement thousands [7, 11] of bit-DFA, which is not practical under today's silicon technology. FPGA based implementation is limited by routing interconnections for this magnitude of bit-DFA. Jung et al [4] and Tan et al [11] show that the bit-DFA method can achieve only around 1 GB/s on FPGAs. Today's multi-core technology has not reached hundreds of cores yet. ASIC seems a good candidate for realizing bit-DFA, however, fine grain of thousand cores consumes both area and power due to duplication of processing and control circuitry [11].

Memory Wastage: Since introduced in 2005 [10], the bit-DFA method has not been put into its limitation to give *optimal* memory efficiency. This is because the method aims to compress DFA memory requirement by reducing alphabet size rather than exploiting the nature of shared states.

Shared State Phenomenon happens when a DFA is constructed from two patterns (or more) that have the same prefix. These patterns share some states that are not shared if each DFA is separately constructed. For example, both "his" and "her" have 'h' as their prefix. In Fig. 1, the pattern "his" individually has 4 states, including a start state, and "her" also has 4 states. When they are built together in the same DFA, they need only 6 states. The state 0 and state 1 are the shared states. A bit-DFA also has sharing opportunities. For the bit-DFA 3 (Fig. 2), shows that the bit-7_6 sequences of patterns "his", "her" and "man" are the same. Thus, these three patterns share every single state. In fact, the chances of shared prefixes are much more in bit-DFA than normal DFA, as shown in Table 1. The first column in the table depicts the case where we construct a separate DFA for each pattern while the second column where constructing one DFA for a whole set of patterns. Existing approaches [7, 10] that partition a set of patterns into many subsets lower the amount of shared states, thus wasting memory space.

	one pattern per DFA	one normal DFA	bit DFA 3	bit DFA 2	bit DFA 1	bit DFA 0
states	97427	67342	27979	42366	50341	53807
$\frac{states}{char.}$	106%	74%	31%	46%	55%	59%

Table 1: Shared State Phenomena for the Snort's patterns (Sept'07) which has 5892 patterns, and 91575 characters.

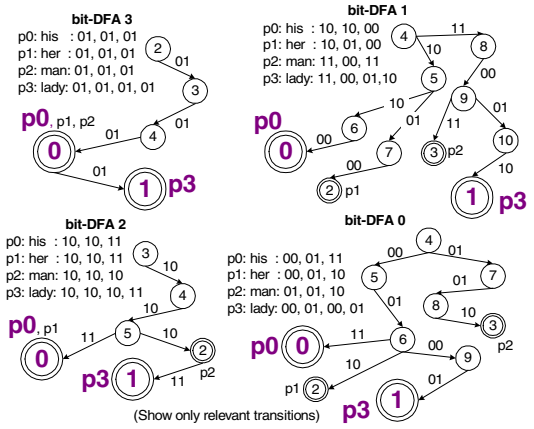

Figure 3: The result of *State Relabeling* from Fig. 2

These above two problems grow from the same root: output states are shared by patterns in a bit-DFA. Our following explanation will make this argument clear.

1. The bit-DFA method introduces opportunities of sharing output states among patterns, which is not the case for any normal DFA method. The bit-DFA 3 (Fig. 2) shows an example where the (output) state 4 is shared by three patterns: p_0, p_1 and p_2. This creates a problem; how to identify an output state matches which pattern. In [7] and [10], *bit-vectors* are used to identify

the matched patterns, where each bit in a bit-vector represents a corresponding pattern. In this case, the bit-vector of the state 4 is "1110". For the match of p_1, the bit-DFA 3, 2, 1 and 0 must reach their own states 4, 3, 5 and 4, respectively. The bit-vectors of these states are "1110", "1100", "0100", and "0100", respectively. The result of ANDing these four bit-vectors is "0100", and only the bit identifier of p_1 asserts.

2. The length of a bit-vector grows with the amount of patterns, so does its required memory space. Snort's pattern set (Sept'07) contains 5892 patterns, which dictates the bit-vector length as 5892 bits and the overall memory needed over 150 MB (Table 2). Results from [7] shows the memory spent on storing bit-vectors is 78% of the total memory.

3. The bit-vector size can be reduced by dividing the pattern set into subsets. Both Piyachon et al [7] and Tan et al [10] adopted this approach to perform pattern matching with thousands of bit-DFA. This finally leads to the aforementioned two issues.

We are motivated to address these issues while still taking advantage of the reduced alphabet size in bit-DFA. Although the sharing of output states is the nature of bit-DFA, we attempt to eliminate the storage needed for bit-vectors. In particular, we propose to use two schemes, Label Translation Table (LTT) and CAM-based Lookup Table (CLT), to determine the final matching pattern, instead of using bit-vectors. In this way we can reduce the size of memory required to store bit-DFA so that the bit-DFA and related tables can reside in limited on-chip memory space to improve pattern matching speed. We next describe LTT and CLT.

patterns/subset	8	16	32	64	128	256	5852
State Trans Tb	7.4	6.3	6.6	6.9	7.0	7.0	1.6
bit-vectors	1.5	2.5	4.8	9.2	17.3	32.1	150.1
Total	8.9	8.8	11.4	16.1	24.3	39.1	151.7

Table 2: Memory usages (MB) of the bit-DFA model of Tan et al with Snort (Sept'07)

4. THE DESIGN

4.1 Common Output States

We develop our schemes based on observations on bit-DFA. Bit-DFA states are identified with unique numbers, and without loss of generality, output states are numbered lower than non-output states. This can be done with a relabeling algorithm that we will explain in Section 4.2.

A pattern set Π contains of patterns $p_0, p_1, ..., p_K$, i.e., $\Pi = \{\forall p_i \mid i = 0, 1, 2, ..., K\}$ when K is the amount of patterns. We denote a state as $\xi_{d,i}$ where d is the bit-DFA number (0 through 3 in Fig. 3), and i is the state label. A state $\xi_{d,i}$ is an output state iff it matches a pattern set, $\pi_{d,i} \subset \Pi$. (A state is *not* an output state iff $\pi_{d,i} = \emptyset$.) For example in Fig. 3, the state 0 in the bit-DFA 3 matches patterns p_0, p_1 and p_2. That is, $\pi_{3,0} = \{p_0, p_1, p_2\}$, and $\pi_{3,0} \subset \Pi$. The state 4 does not match any pattern, i.e., $\pi_{3,4} = \emptyset$. Thus, the state 4 is not an output state.

There are two types of output states: *common* one and *unique* one. When we have 4 bit-DFA ($0 \leqslant d \leqslant 3$), a state $\pi_{d,i}$ is a *common* one iff $\pi_{d,i} \neq \emptyset$ and

$$\pi_{3,i} \cap \pi_{2,i} \cap \pi_{1,i} \cap \pi_{0,i} \neq \emptyset$$

A state is a *unique* one iff $\pi_{d,i} \neq \emptyset$, and it is not *common*.

Theorem 1: The cardinality of $\pi_{3,i} \cap \pi_{2,i} \cap \pi_{1,i} \cap \pi_{0,i}$ is one for a *common* output state $\xi_{d,i}$.

Proof: In the bit-DFA method, the set intersection operation $\pi_{0,i} \cap \pi_{1,i} \cap \pi_{2,i} \cap \pi_{3,i}$ is equivalent to the final AND operation on bit-vectors, which determines if any patterns are matched when all bit-DFA reach output states whose label is i. The definition of a *common* output state means the bit-DFA match some patterns of length L, which is the amount of transitions from the start state to state $\xi_{d,i}$. Considering all patterns of the same length are unique, the amount of patterns matched is one. Therefore the cardinality of the intersection is one.

In Fig.3, the state $\xi_{3,0}$ is a *common* output state and, $\pi_{0,0} \cap \pi_{1,0} \cap \pi_{2,0} \cap \pi_{3,0} = \{p_0\}$. From the definition of *common* output states, it is obvious that all $\xi_{d,0}$, where $0 \leqslant d \leqslant 3$, are *common* output states, so only the state label is significant for *common* states. Similarly, all $\xi_{d,1}$ are *common* output states, and the intersection set is $\{p_3\}$. All other output states are *unique* ones. We design two schemes to determine the matched patterns: LTT for *common* output states, and CLT for *unique* ones. Before explaining LTT and CLT, we next describe algorithms to relabel states.

4.2 State Relabeling Algorithm

The states in bit-DFA are labeled with unique numbers that are determined by Aho-Corasick algorithm. We relabel them with new labels so that: (1) every the output state is numbered lower than non-output states, and (2) every output state that matches the same pattern has the same label. The first goal is to cluster output states into lower label ranges, and the second goal is to bring out *common* output states.

Algorithm 1 Relabeling *Common* Output States

1: Let ν be the new label. And it begins with $\nu = 0$.
2: Go over each bit-DFA to search for states $\xi_{3,a}$, $\xi_{2,b}$, $\xi_{1,c}$, $\xi_{0,d}$ that $\pi_{3,a} \cap \pi_{2,b} \cap \pi_{1,c} \cap \pi_{0,d} \neq \emptyset$
3: Relabel these states with the same new label ν, and save them in a different data structure. That is, $\xi_{3,a,\nu}$, $\xi_{2,b,\nu}$, $\xi_{1,c,\nu}$, $\xi_{0,d,\nu}$. The old labels (a, b, c, d) are still preserved as references to the pre-relabeled (old) bit-DFA.
4: Store the result of $\pi_{3,a} \cap \pi_{2,b} \cap \pi_{1,c} \cap \pi_{0,d}$ in $LTT[\nu]$, i.e, the address ν in LTT. (For example, $\pi_{3,4,0} \cap \pi_{2,3,0} \cap \pi_{1,3,0} \cap \pi_{0,3,0} = \{p_0\}$. We refer to the state 4, 3, 3 and 3 in the bit-DFA 3, 2, 1 and 0 in Fig. 2. These states are relabeled as the state 0 in Fig. 3.)
5: $\nu \leftarrow \nu + 1$
6: Continue repeating the step 1 to 5 until all states are exhausted.

The relabeling are outlined in the algorithms. We begin with finding and relabeling common output states in Algorithm 1. So, new labels of common ones will be numbered lower than non-output states. Note that only common output states are relabeled by checking their intersection operation $\pi_{3,i} \cap \pi_{2,i} \cap \pi_{1,i} \cap \pi_{0,i} \neq \emptyset$. In this algorithm, we keep also adding data footprints to LTT. The algorithm will continue

until it cannot find any common output states. Algorithm 2 begins with relabeling unique output states in each bit-DFA, followed by relabeling non-output states until all states are labeled. In Algorithm 2, N_{bitDFA} is the amount of bit-DFA (4 in our case).

Algorithm 2 Relabeling *Unique* Output States, and Non-Output States

Require: The new label ν from Algorithm 1

1: After finishing Algorithm 1, the new label ν is the next value for the rest states. (For example, ν is '2' in Fig 3.)
2: $tmp \leftarrow \nu$
3: **for** $d \leftarrow 0$ to N_{bitDFA} **do**
4: Relabel output states with *tmp*. Then the new state will be $\xi_{d,i,tmp}$ when i is the old label. And keep increasing *tmp*, i.e, $tmp \leftarrow tmp + 1$, until all output states are exhausted.
5: Relabel non-output states with *tmp*. Then the new state will be $\xi_{d,i,tmp}$ when i is the old label. And keep increasing *tmp*, i.e, $tmp \leftarrow tmp + 1$ until all non-output states are exhausted.
6: **end for**

4.3 Label Translation Table (LTT)

We construct a LTT to translate state labels to pattern labels (instead of using a bit-vector). A state label is used as an address number that points to a row in the table, as shown in Fig. 4. The LTT stores the pattern label of the intersection set at a location pointed by the state label of a *common* output state. An example table shows in Fig. 4, the address '0' and '1' store the pattern label of p_0 and p_3, respectively. This scheme is based on Theorem 1. The match of p_0 occurs when all bit-DFA in Fig. 3 reach their own state 0 at the same time. The state 0 label which is '0' is the address pointing to the LTT.

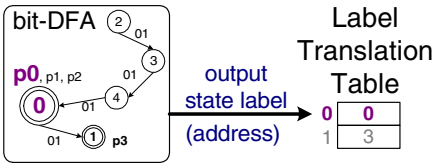

Figure 4: Label Translation Table mechanism for Fig. 3

4.4 CAM-based Lookup Table (CLT)

The identifiers of *unique* output states can form unique sequences that can be used to identify matched patterns. In Fig. 3, when the bit-DFA 3 reaches its output state 0, and the bit-DFA 2, 1 and 0 reach the states 0, 2 and 2, respectively, a match on the pattern p_1 occurs. The state labels '0', '0', '2' and '2' are different among bit-DFA, however, the concatenation of these numbers, "0,0,2,2", is *unique*, thus can be used to identify a matched pattern.

The concatenated state label is a unique number that is stored in a Content Addressable Memory (CAM). The CAM will give the address of a stored datum iff the input equals to the datum. Thus, we can use a CAM to issue the address that points to a lookup table, which contains the corresponding pattern label. Fig. 5 shows an example of a CLT.

The data in the CAM are "0,0,2,2" and "0,2,3,3" at the address '0' and '1', respectively. The first one, "0,0,2,2", is the concatenated state label from the bit-DFA 3, 2, 1 and 0, respectively, for the match of pattern p_1. And the second one, "0,2,3,3", is the concatenated state label for the match of p_2.

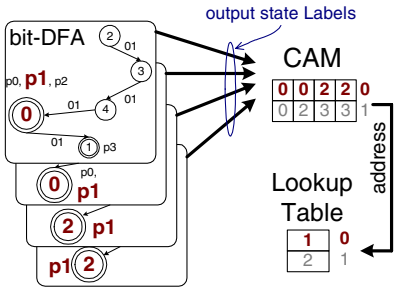

Figure 5: CAM-based Lookup Table mechanism

4.5 Arbiter Mechanism

Arbiter (Fig. 6)

The arbiter operates under the following principles.

- A match occurs only when all bit-DFA reach their own output states at the same time. Thus, we access either LTT or CLT only when all bit-DFA reach their own output states at the same time.
- Since LTT is for *common* output states, and CLT for *unique* ones, we use either one, but not both.

Based upon these two principles, the arbiter can be engineered as Fig.6.

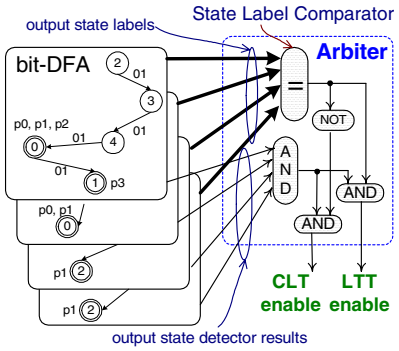

Figure 6: Arbiter mechanism

Output State Detector (Fig. 10)

Each bit-DFA uses an output state detector to distinguish output states from non-output ones. After relabeling states, labels of output states are less than non-output ones. Therefore, the detector is simply a "less-than" comparator. Note that the results of the detectors (from all bit-DFA) must be ANDed together to check whether all bit-DFA reach their own output states at the same time by the 4-input AND gate in Fig. 6.

State Label Comparator (Fig. 6)

We use LTT only when the current states (from all bit-DFA) are equal at the same time, and when these current states

are output states. The first checking can be done with a 4-input "equal" comparator. The second one can be done by the output state detectors.

5. MEMORY REQUIREMENTS

In this section, we form formula to calculate the memory usage by LTT and CLT.

Label Translation Table

In Fig. 4, The memory size of the table is proportional to its rows and width. The amount of rows equals to the minimal amount of output states, $N_{OutSt,Min}$. Fig. 3 shows that the amount of output states in the bit-DFA 3, 2, 1 and 0 are 2, 3, 4 and 4, respectively. Hence, $N_{OutSt,Min}$ is 2. The width is $\lceil log_2(K) \rceil$ where K is the amount of patterns. Hence, the memory required by LTT is:

$$M_{LTT} = N_{OutSt,Min} \cdot \lceil log_2(K) \rceil \quad (1)$$

CAM-based Lookup Table

The memory size of the CAM is proportional to its rows and width. The amount of rows equals to the amount of patterns that cannot be performed by using LTT scheme. And the width is $\lceil log_2(N_{OutSt,Max}) \rceil$. When N_{bitDFA} is the amount of bit-DFA (in our case, $N_{bitDFA} = 4$), the CAM memory size is:

$$M_{CAM} = (K - N_{OutSt,Min}) \cdot N_{bitDFA} \cdot \lceil log_2(N_{OutSt,Max}) \rceil \quad (2)$$

For the symmetrical design, we use the maximal output-states, $N_{OutSt,Max}$, in Equation 2. For example (Fig. 3), $N_{OutSt,Max} = 4$ states.

The memory required by the CAM's Lookup Table is proportional to its rows and width. The width equals to the width of the LTT. Hence, the lookup table memory size is:

$$M_{Lookup} = (K - N_{OutSt,Min}) \cdot \lceil log_2(K) \rceil \quad (3)$$

For Snort (Sept'07), $K = 5852$ patterns, $N_{OutSt,Max} = 4609$ states. $N_{OutSt,Min} = 2094$ states as in Table 3.

bit-DFA	3	2	1	0
output states	**2094**	3932	4520	**4609**
total states	27979	42366	50341	53807
% output st	7.48%	9.28%	8.98%	8.57%

Table 3: The amount of output states and total states in each bit-DFA from Snort's pattern set (Sept'07). $N_{OutSt,Max} = 4609$ states. $N_{OutSt,Min} = 2094$.

6. ARCHITECTURE

We present an architecture supporting the proposed schemes. Fig. 7 depicts the architecture of our **Pattern Matching Machine**. It consists of four *bit-Automaton Engines* (bAEs), an *Arbiter*, an *LTT*, and a *CLT*.

A *bit-Automaton Engine* (Fig. 8) consists of a *State Transition Table* (STT), a *Next State Selector*, and an *Output State Selector*. Every week (or day), our in-house pattern compiler generates four footprints (of four bit-DFA) from a new pattern set. Then, each footprint is uploaded to the STT (Fig. 8), the *Start State Register* (Fig. 9), and the *Lowest Non-Output State Register* (Fig. 10) in each bAE. The compiler also generates footprints for the LTT and CLT.

After the *uploading process*, the machine begins *running processes*. In the *Next State Selector* (Fig. 9), the value in

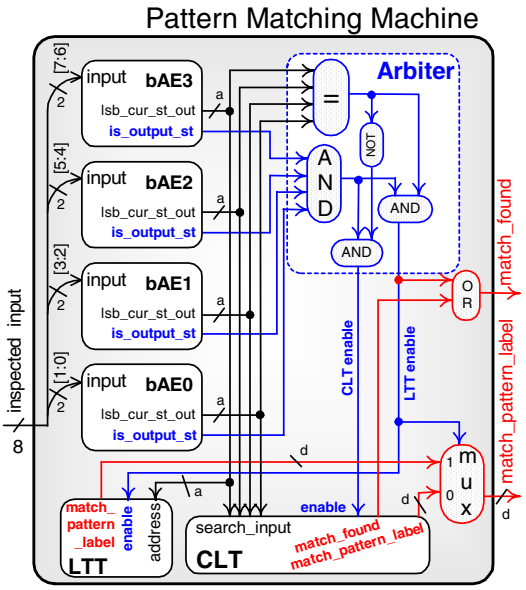

Figure 7: Pattern Matching Machine. The blue lines show the paths of the Arbiter mechanism. And the red lines show the paths of declaring a match, and a match pattern label. ($a = \lceil log_2(N_{OutSt,Max}) \rceil$, and $d = \lceil log_2(K) \rceil$)

the start state register is the start state label. Fig. 3 shows examples of the start states of the bit-DFA 3, 2, 1 and 0 are 2, 3, 4 and 4, respectively. Note that a start state label may not be '0' after relabeling. The start state label is loaded into the current state register in order to point the current address of the STT. The data at the address is the four possible next state labels that will be loaded into the next state register. Every cycle, each inspected input byte is split to four 2-bits. The mux selects a next state label (from the next state register) based on this 2-bit value. The selected next state label is inputted into the current state register. Then, the next round continues until the machine receives the last byte of the inspected input stream. A new running process starts again by loading the start state label from the start state register to the current state register.

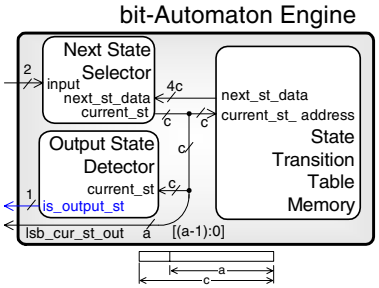

Figure 8: bit-Automaton Engine (While N_{St} is the amount of states, $a = \lceil log_2(N_{OutSt,Max}) \rceil$, and $c = \lceil log_2(N_{St}) \rceil$.)

The *Output State Detector* (Fig. 10) distinguishes output states from non-output states by using the comparator. The detector result will be used by the arbiter (Fig. 7). The state relabeling assigns all output states with lower number than

non-output states. For example, in the bit-DFA 1 (Fig. 3), output state labels are 0, 1 and 2; non-output ones are 3, 4 and 5. That is, any state that is less than 3 is an output state. So, the value in this Lowest Non-Output State Register is 3.

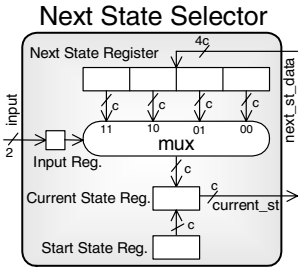

Figure 9: Next State Selector

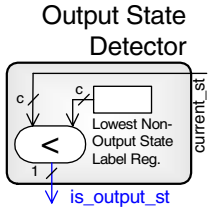

Figure 10: Output State Detector

7. EXPERIMENTAL EVALUATION

We compare memory usages of our method against previous works in Table 4. The memory sizes are acquired from our pattern compiler. The compiler gives the sizes based on models of Tan-Sherwood (T-L) [10], Piyachon-Luo (P-L) [7], and our method by using Snort's pattern set of Apr'04 and Sept'07. While the previous works use *bit-vectors*, we use LTT and CLT instead. Our method shows significant improvement over both works. Especially on Snort'07, both P-L and T-S methods are too expensive to implement in on-chip memory while our method requires only 1382.4 KB. Our method uses the memory of the STT less than others because their methods have less shared states. That is because our method has only one set while the others have hundreds of subsets. And, this is the practical proofs about *shared state phenomenon* (Section 3). The total memory usage of our LTT, CAM and the lookup table is also less than the memory usage of bit-vectors.

Snort		STT	bit-Vect	LTT	Lkup Tb	CAM	Total Mem
Apr 2004	T-S	750.8	333.7				1084.5
	P-L	750.8	14.1				764.9
	ours	313.9		0.97	1.40	5.7	321.9
Sept 2007	T-S	6455.0	2582.0				9037.1
	P-L	6455.0	45.7				6500.8
	ours	1349.6		3.20	5.74	23.8	1382.4

Table 4: Memory (KB) Comparison of Tan-Sherwood's (T-S), Piyachon-Luo's (P-L), and our method. (STT = State Transition Table)

8. RELATED WORK

There are recently research works on pattern matching. Artan et al proposed a multi-signature detection using prefix Bloom Filters [2]. While we do not partition a pattern set into many subsets to exploit shared state phenomena, Becchi et al [3] proposed complex algorithms to merge states. Sourdis et al [9] proposed building non-deterministic finite automata (NFA) using logic structures in FPGA rather than memory based DFA. There are some implementation using network processor [6, 5], and some papers using the bit-DFA method [4, 6, 7, 10, 11]

Piyachon et al [7] proposed a state relabeling scheme to eliminate *zero* bit-vectors, and cluster output state labels in one group. Our state relabeling algorithms (Section 4) not only cluster output state labels in one group, but also make sure that every *common* output state (which matches the same pattern) has the same state label.

9. CONCLUSION

In this paper, we have addressed the problems of the bit-DFA method that it has faced since it is introduced in 2005. We proposed using *Label Translation Table* and *CAM-based Lookup Table* methods to tackle the problems. The proposed schemes reduces the usage by up to 85%, compared against the previous works by [7, 11]. At the same time, the amount of processing elements required is reduced from thousands to four. We present the architecture that realizes our proposed methods. The architecture suits for both ASIC and FPGA implementation as well as multi-core system. In the near future, we plan to apply our proposed schemes for regular expression based pattern matching.

10. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] N. S. Artan and H. J. Chao. Multi-packet Signature Detection using Prefix Bloom Filters. In *IEEE GLOBECOM*, November 2005.

[3] M. Becchi and S. Cadambi. Memory-Efficient Regular Expression Search Using State Merging. In *IEEE INFOCOM*, 2007.

[4] H-J Jung, Z. K. Baker, and V. K. Prasanna. Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems. In *RAW at IEEE IPDPS*, April 2006.

[5] J. Ni, C. Lin, Z. Chen, and P. Ungsunan1. A Fast Multi-pattern Matching Algorithm for Deep Packet Inspection on a Network Processor. In *International Conference on Parallel Processing*, 2007.

[6] P. Piyachon and Y. Luo. Efficient memory utilization on network processors for deep packet inspection. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, December 2006.

[7] Piti Piyachon and Yan Luo. Compact State Machine for High Performance Pattern Matching. In *ACM/IEEE Design Automation Conference*, June 2007.

[8] Snort. http://www.snort.org/, 2003.

[9] I. Sourdis, J. Bispo, J. M.P. Cardoso, and S. Vassiliadis. Regular Expression Matching in Reconfigurable Hardware. In *Journal on VLSI and Signal Processing*, October 2007.

[10] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *IEEE/ACM International Symposium on Computer Architecture*, 2005.

[11] L. Tan and T. Sherwood. Bit-Split String-Matching Engines for Intrusion Detection and Prevention. In *ACM Transactions on Architecture and Code Optimization (TACO)*, March 2006.