This paper was presented as part of the main technical program at IEEE INFOCOM 2011

FEACAN: Front-End Acceleration for Content-Aware Network Processing

Yaxuan Qi, Kai Wang, Jeffrey Fong, Yibo Xue, Jun Li
Tsinghua National Lab for Information Science and Technology (TNList), Beijing, China
{yaxuan, yiboxue, junl}@tsinghua.edu.cn

Weirong Jiang, Viktor Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, CA, USA
{weirongj, prasanna}@usc.edu

Abstract—**Modern networks are increasingly becoming content aware to improve data delivery and security via content-based network processing. Content-aware processing at the front end of distributed network systems, such as application identification for datacenter load-balancers and deep packet inspection for security gateways, is more challenging due to the wire-speed and low-latency requirement. Existing work focuses on algorithm-level solutions while lacking system-level design to meet the critical requirement for front-end content processing. In this paper, we propose a system-level solution named FEACAN for front-end acceleration of content-aware network processing. FEACAN employs a software-hardware co-design supporting both signature matching and regular expression matching for content-aware network processing. A two-dimensional DFA compression algorithm is designed to reduce the memory usage and a hardware lookup engine is proposed for high-performance lookup. Experimental results show that FEACAN achieves better performance than existing work in terms of processing speed, resource utilization, and update time.**

***Keywords*—regular expressions; DFA; hardware acceleration**

I. INTRODUCTION

Modern networks are increasingly becoming content aware, to improve either data delivery or network security via content-based processing. Content-aware network processing requires new levels of support in network appliance in form of the ability to comprehend complex network behaviors. With the rapid growth of cloud computing and system virtualization, content-aware network processing at the front end of large-scale network systems is in great demand to provide global optimization of data delivery and holistic protection for network security.

Figure 1 shows a typical architecture of modern distributed network systems [1]. Network traffic comes from external network interfaces on the front-end processing blades. Multi-core network processors, such as Netronome NFE-i8000 [2], NetLogic XLP832 [3] and Cavium OCTEON5860 [4], are the main processors responsible for network-level (L2~L4) processing. Content-aware processing (L4~L7), however, are often done by acceleration co-processors (accelerator), such as LSI Tarari T2000/2500 [5], NetL7 NLS2008 [6] and Cavium DFA engine [4]. These content-aware accelerators have high-bandwidth interconnection (e.g. PCI-Express, XAUI) to interchange data with the main processors, dedicated memory (e.g. on-chip SRAM or low-latency RLDRAM) to store local data structures and parallel lookup engines to process multiple flows at the same time. According to the processing results by the content-aware coprocessors, network traffics will be forwarded to certain back-end servers through internal network interfaces for further processing. Table I show some important application of front-end content-aware processing.

TABLE I. FRONT-END CONTENT-AWARE PROCESSING

Network	Purpose	Front-end Processing
Service Provider	Traffic billing	Application identification
Datacenter	Optimized storage	Content-aware load balancing
Enterprise	Intrusion prevention	Stateful and deep inspection

Content-aware network processing at the front end is more challenging than host-based back-end content processing due to the following critical requirements:

- *High speed*: Front-end content-aware processing must support wire-speed (e.g. 100 Gbps) data rate, because slow or unstable processing speed at the front end will affect the overall performance of a distributed network system.
- *Low latency*: Due to high bandwidth at the front end of a network system, content-aware processing accelerators must have extremely low latency to avoid bottlenecks in packet buffering and system synchronization.
- *Efficient update*: For security issues, new policies for front-end content-aware processing should take effect as soon as possible to react to sudden attacks. The accelerator should also support online update to apply new policies without affecting established connections.

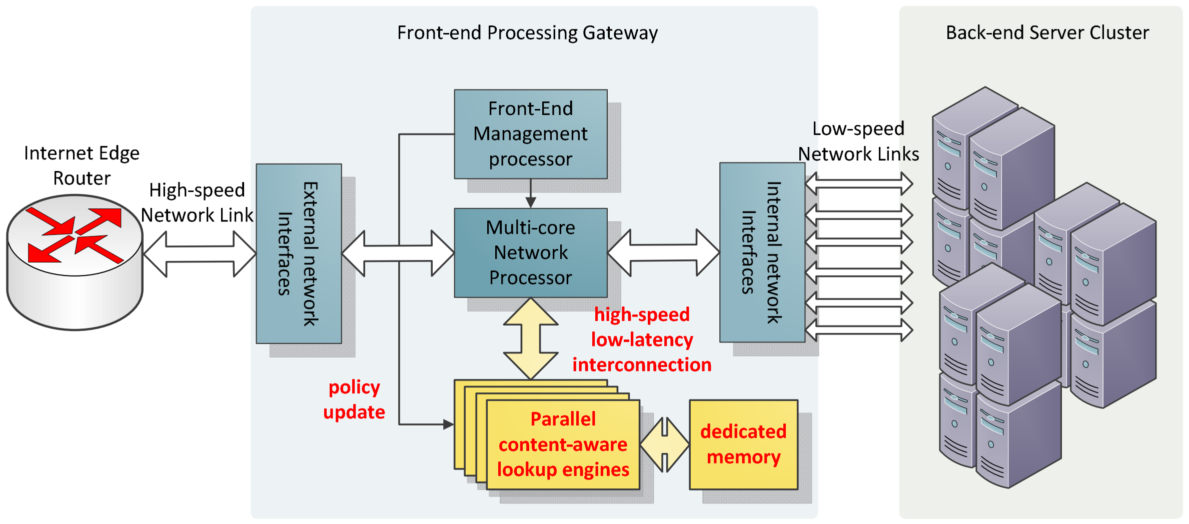

Front-end Processing Gateway
Back-end Server Cluster
Internet Edge Router
High-speed Network Link
External network Interfaces
Front-End Management processor
Multi-core Network Processor
Internal network Interfaces
Low-speed Network Links
high-speed low-latency interconnection
policy update
Parallel content-aware lookup engines
dedicated memory

Figure 1. Content-aware processing accelerator for a distributed network system

To meet these requirements, we propose a system-level solution named FEACAN for front-end acceleration of content-aware network processing. FEACAN has a software-hardware co-designed architecture supporting both signature matching and regular expression matching. Main contributions include:

- *An efficient algorithm*: The FEACAN algorithm uses a two-dimensional compression technique that achieves both small memory usage and fast lookup speed.
- *Low-latency lookup engine*: Based on the FEACAN algorithm, a low-latency lookup engine is designed to provide wire-speed content-processing performance.
- *Performance evaluation*: Experimental results on real-life data sets and network traffic show that FEACAN can achieve more than 100 Gbps processing speed based on state-of-the-art hardware technologies.

The rest of the paper is organized as follows. Section II gives the related work of our research. Section III presents the basic FEACAN algorithm, which is further optimized in Section IV. Section IV describes the hardware design of the FEACAN lookup engine. Section V evaluates the performance of FEACAN and compares to some existing best-known solutions. Section VI states our conclusion.

II. Related Work

Regular expression matching has been proved to be fundamentally efficient and flexible for content-aware network processing [7]. Theoretically, the regular expression matching problem can be solved by using either nondeterministic or deterministic finite automata (NFAs and DFAs) [8] [9].

Floyd et al first showed that NFA based regular expression matching can be efficiently implemented on programmable logic array [10]. Sindhu et al. [11] and Clark et al. [12] have implemented NFA-based regular expression matching on FPGA devices and achieved good performance in terms of processing speed and space efficiency. However, because these approaches encode the NFA automata in hardware logics, the online update requirement is difficult to meet due to the need for re-synthesizing the regular expression logics [13]. Therefore, regular expression engines which use memory rather than logic, are often more desirable as they provide better degree of flexibility [14].

Memory-based solutions often use DFA to achieve predictable and acceptable memory bandwidth requirements [15]. However, the memory usage of DFAs on complex regular expressions rule sets can be prohibitively large. Recent researchers solve this problem by proposing various DFA compression algorithms. In this context, Tuck et al. use bitmap technique to compress consecutive transitions with the same next state transition [16]. Because their approach only considers the redundancy inside a state, the performance might be unstable when the number of unique transitions increases. According to experiments, their approach achieves up to 90% compression ratio on regular expression rule sets and less than 30% compression ratio on large number of strings. Another problem of their approach is that the bitmaps are stored together with their corresponding state, i.e. an ***N***-state DFA will have ***N*** bitmaps. So the storage and computation of large number of bitmaps become another problem for efficient hardware implementation.

Kumar et al. observed that a lot of the states in a DFA have similar next state transitions [14]. Based on this observation they proposed the D^2FA algorithm to compress the DFA table by employing default paths and default transitions. States along a default path only need to store its unique transitions, and its default transitions can be found from their ancestors along the path. Without controlling the depth of default path, D^2FA achieves 95% compression ratio on real-life regular expression rules. Becchi et al. proposed an improved D^2FA with directional default-path selection, making each default transition to a default-state closer to the root node [15]. This algorithm achieves superior worst-case search speed and significantly reduces the preprocessing time. Although D^2FA based algorithms achieve good compression ratio, they are inherently difficult for efficient hardware implementation because: i) each state transition requires comparing all the non-default transitions in the current state before taking the default-

$$\begin{bmatrix}
\ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \iddots \\
\cdots & 1 & 1 & 1 & 1 & 1 & 9 & 35 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
\cdots & 1 & 1 & 36 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
\cdots & 1 & 1 & 1 & 1 & 1 & 9 & 121 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
\cdots & 1 & 1 & 1 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
\cdots & 1 & 1 & 123 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
\cdots & 1 & 1 & 1 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\
\iddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{bmatrix}$$

Figure 2. Segment of a real DFA transition table

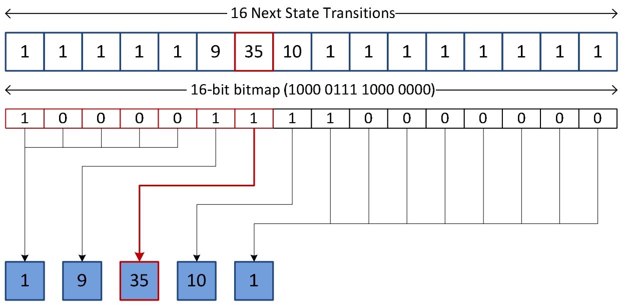


Figure 3. Bitmap compression technique

transition; ii) such linear comparisons are recursively done among all states in the default-path until a non-default transition is found. In our experiments, we observed that the average number of transitions traversed per input character can be 100 times larger than the number of traversed states, meaning that the memory bandwidth required by D^{2}FA might be 100 times larger than that of the original DFA algorithm to achieve similar performance. Kumar et al. refined this algorithm by content-addressed hashing [18], but the proposed CD^{2}FA algorithm only holds when most states have less than 5 non-default transitions [15].

Therefore, existing best known work on regular expression matching cannot meet all the three requirements for front-end content-aware processing. The NFA-based solutions suffer from online update while the DFA-based algorithms cannot be efficiently implemented in hardware. In our research, we focus on the DFA-based solution.

III. FEACAN Algorithm Design

A. Motivation

Consider a DFA with $\boldsymbol{N}$ states representing regular expressions over an alphabet $\boldsymbol{\Sigma}$ with size $\boldsymbol{M} = |\boldsymbol{\Sigma}|$ contains $\boldsymbol{N} * \boldsymbol{M}$ next state transitions. We can use a two dimensional table $\boldsymbol{T_{N*M}}$ to represent the DFA. Each element $\boldsymbol{t(n,m)}$ in $\boldsymbol{T_{N*M}}$ is a next state transition and each row $\boldsymbol{S_n} = \{\boldsymbol{t(n,1)}, \boldsymbol{t(n,2)}, \ldots, \boldsymbol{t(n,M)}\}$ represents a DFA state. Given an input character $\boldsymbol{c} \in \boldsymbol{\Sigma}$ and a current state $\boldsymbol{S_{n0}}$, we can find the next state $\boldsymbol{S_{n1}}$ by loading the c^{th} next state transition $\boldsymbol{t(n0,c)}$ in the DFA table $\boldsymbol{T_{N*M}}$, i.e. $\boldsymbol{S_{n1}} = \boldsymbol{t(n0,c)}$.

According to experiments on real-life regular expression rule sets, we found that the DFA table $\boldsymbol{T_{N*M}}$ is very sparse and has a lot of redundancies. Figure 2 shows a real-life DFA table generated by real-life regular expression rules. From this figure we can see that the redundancy lies in two dimensions:

- *Intra-state redundancy*: Within a certain state $\boldsymbol{S_n}$, the number of unique transitions is likely to be small, i.e. the number of different $\boldsymbol{t(n,m)}$ in $\boldsymbol{S_n}$ is far less than $\boldsymbol{M}$.
- *Inter-state redundancy*: A certain group of states $\boldsymbol{S_{n1}}, \boldsymbol{S_{n2}}, \ldots, \boldsymbol{S_{nk}}$ are likely to have similar transitions, i.e. for most $\boldsymbol{m}$, $\boldsymbol{1 \le m \le M}$, $\boldsymbol{t(n1,m)} = \cdots = \boldsymbol{t(nk,m)}$.

Motivated by these two observations, the FEACAN algorithm uses both intra-state and inter-state compression techniques to compress the original DFA table. For intra-state compression, the FEACAN algorithm uses an advanced bitmap technique which not only reduces the number of redundant transitions, but also supports fast bitmap computation. For inter-state compression, the algorithm uses a 2-stage grouping algorithm which efficiently reduces inter-state redundancy while retaining intra-state position information for fast lookup.

B. The FEACAN Algorithm

The FEACAN algorithm takes the following steps to compress the original DFA table in Figure 4:

1) Bitmap compression

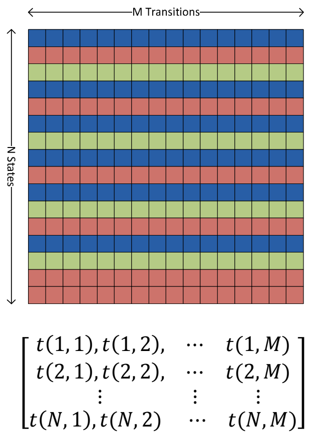
First, we define the *unique transition* as: in the $\boldsymbol{n}^{\text{th}}$ state, $\boldsymbol{1 \le n \le N}$, i) the first transition $\boldsymbol{t(n,1)}$ is a unique transition, and ii) The $\boldsymbol{m}^{\text{th}}$, $\boldsymbol{1} < m \le M$ transition $\boldsymbol{t(n,m)}$ is a unique transition if and only if $\boldsymbol{t(n,m)} \neq \boldsymbol{t(n,m-1)}$. Then we can use the bitmap technique shown in Figure 3 to represent each $\boldsymbol{M}$-transition state by an $\boldsymbol{M}$-bit bitmap and $\boldsymbol{U}$ unique transitions. To find the $\boldsymbol{m}^{\text{th}}$ transition of original state among the $\boldsymbol{U}$ unique transitions, we can count the number of 1's (defined as $\boldsymbol{u}$) in the first $\boldsymbol{m}$ bits of the bitmap, and then load the $\boldsymbol{u}^{\text{th}}$ transition in of the unique transition list, which has the same next state transition as the $\boldsymbol{m}^{\text{th}}$ transition in the original state. Figure 5 shows the data structure after the intra-state bitmap compression.

2) First-stage grouping

The first-stage grouping is to group states associated with identical bitmap together, so that we can reduce the number bitmaps. As shown in Figure 6, after the first-stage grouping (totally $\boldsymbol{K}$ groups), only unique bitmaps are retained and each one of them corresponding to a group of corresponding states. First-stage grouping not only reduces the number of bitmaps from $\boldsymbol{N}$ to $\boldsymbol{K}$, but also facilitates inter-state compression because states sharing the same bitmaps are more likely to have similar transitions (see Section VI).

3) Second-stage grouping

The second-stage grouping is to group similar states with in each of the $\boldsymbol{K}$ groups into $\boldsymbol{L}$ sub-groups. Two states are similar if they satisfy: i) They share the same bitmap (so that they have the same number of transitions); ii) More than ***thresh***% transitions are identical to each other. According to our experiments, the value of ***thresh***% can be within the range



M Transitions
N States
$$\begin{bmatrix} t(1,1), t(1,2), & \cdots & t(1,M) \\ t(2,1), t(2,2), & \cdots & t(2,M) \\ \vdots & \vdots & \vdots \\ t(N,1), t(N,2) & \cdots & t(N,M) \end{bmatrix}$$

Figure 4. Original DFA table

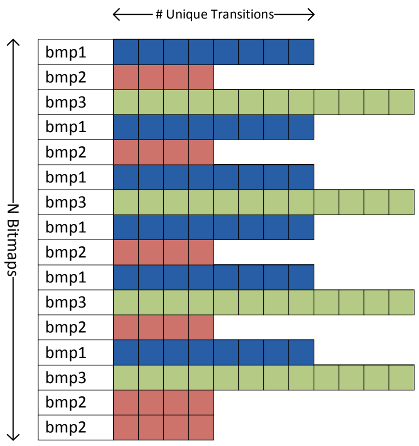
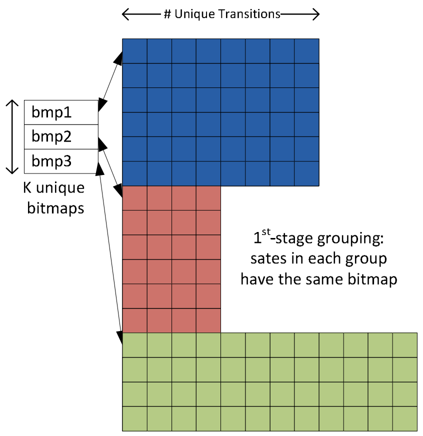
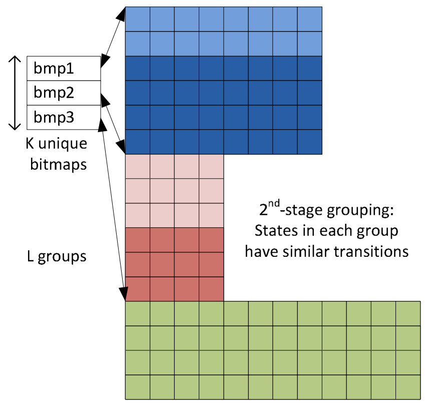


Figure 5. Bitmap compression



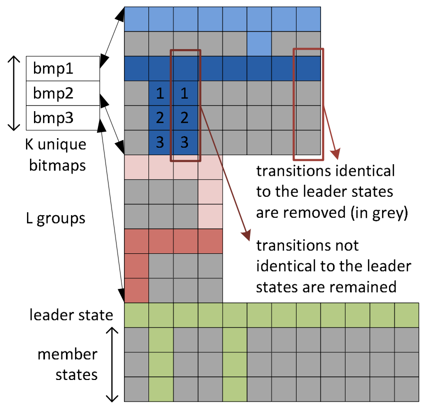
# Unique Transitions
bmp1
bmp2
bmp3
K unique bitmaps
1st-stage grouping: sates in each group have the same bitmap

Figure 6. 1st-stage grouping



bmp1
bmp2
bmp3
K unique bitmaps
L groups
2nd-stage grouping: States in each group have similar transitions

Figure 7. 2nd-stage grouping.



bmp1
bmp2
bmp3
K unique bitmaps
L groups
leader state
member states
transitions identical to the leader states are removed (in grey)
transitions not identical to the leader states are remained

Figure 8. State merging

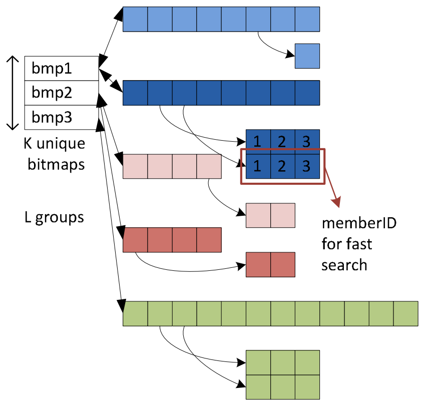


Figure 9. Transition mapping

80%~95%, and the performance is not sensitive to the choice (see Figure 16 for experimental results). Figure 7 shows the results after second-stage grouping, states within each of the ***L*** groups are drawn in the same color.

4) State merging

After the two-stage grouping, the algorithm selects the first state of each of the ***L*** groups as a *leader state*, and define other states as *member states*. Within each group, states are merged by removing redundant transitions in member states. As shown in Figure 7, redundant transitions are shown in gray and each of them is identical to their corresponding transition in the leader state. State merging is inter-state compression because it reduces the redundancy among multiple states.

5) Transition mapping

After both intra-state (by bitmap) and inter-state (by grouping) compression, the original DFA table in Figure 4 is compressed to the data structure in Figure 9 via transition mapping. Transition mapping is to replace each original transition, i.e. the next state ID with its corresponding <***groupID***, ***memberID***> pair. The ***groupID*** is the index to find the group to which the next state belongs and the ***memberID*** represents the index to locate member states within the group. For leader states, all ***memberID*** is zero. Each transition in leader states is associated with a ***memberOffset*** to record the base address to find the list of its member transitions. So each transition in leader states is replaced with a 3-tuple entry: <***groupID***, ***memberID***, ***memberOffset***>. In addition, we also maintain an ***L***-entry address mapping table, in which each entry in the table records <***bitmapID***, ***leaderBase***, ***memberBase***> for the corresponding group. The ***bitmapID*** is used to find the bitmap of the group. The ***leaderBase*** is the address of the first transition in the leader state. The ***membeBase*** is the address of the first transition in member states.

To lookup the data structure shown in Figure 9, we take the following steps: i) Use the ***groupID*** to read <***bitmapID***, ***leaderBase***, ***memberBase***> from the address mapping table. ii) Use the ***bitmapID*** to read the corresponding bitmap of the group and count the number of 1's (***u***) in the first ***c*** (***c*** is value of the input char) bits of the bitmap. iii) Use (***leaderBase***+***u***) as index to read the ***leaderTransition***. iv) If ***memberID*** or ***memberOffset*** is 0, the next state is <***groupID***, ***memberID***> stored in this transition. v) Otherwise, use (***memberBase*** + ***memberOffset*** + ***memberID***) as the index to read ***memberTransition***, and the next state is the <***groupID***, ***memberID***> stored in this transition. Hardware implementation of the search is shown in Section V.

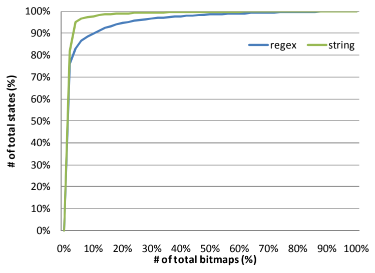


Figure 10. State distribution over bitmaps

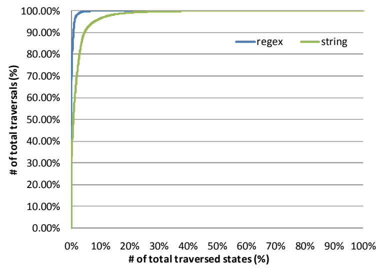
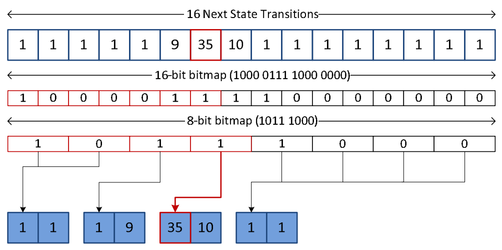


Figure 11. Traversal distribution over states



16 Next State Transitions
16-bit bitmap (1000 0111 1000 0000)
8-bit bitmap (1011 1000)

Figure 12. Hierarchical bitmap compression

TABLE II. NUMBER OF BITMAPS IN REAL-LFE RULES

Rule set	# DFA States (N)	# Unique Bitmaps (K)
regular expression	6533	73
string signatures	56280	112

TABLE III. EFFECT OF BITMAP COMBINATION

Rule set	# of bitmaps before comb.	# of bitmaps after comb.	# of trans. change (%)
regular expression	73	22	↓ 6%
string signatures	112	18	↓ 3%

C. Discussion

According to our design choice, the FEACAN algorithm has the following advantages for system-level implementation:

- *Effective memory compression*: Compared to the original bitmap compression algorithm proposed by Tuck et al. [16], FEACAN DFA uses two-dimensional table compression techniques that fully exploit the redundancy of the DFA table. Experimental results show that the 2-dimensional compression has up to 80% additional memory reduction than the original bitmap compression algorithm.
- *Deterministic memory access*: Different from D^{2}FA [14], no linear searches over all transitions within a state for FEACAN DFA. The worst-case number of transitions visited for each input character is 2, i.e. one in leader state and the other in the member state.
- *Fast preprocessing*: The original D^{2}FA algorithm has $\boldsymbol{O(N^2 logN)}$ time complexity in preprocessing, while FEACAN DFA has only $\boldsymbol{O(NlogN)}$ preprocessing time. Because $\boldsymbol{N}$ is the number of states, the improvement in practice is in multiple orders of magnitudes.

Note that for some complex regular expression rules, the size of the original DFA table can be exponentially large and even with a 90% compression ratio the overall data structure might be still unable to fit in to on-chip memories [27]. In this case, we can integrate the FEACAN with H-cFA [29], HybridFA [28] [32], and XFA [30] [31] to further reduce the memory usage.

IV. FEACAN ALGORITHM OPTIMIZATION

In this section, the basic FEACAN algorithm is optimized at system level for efficient implementation. One optimization is to control the number of bitmaps as well as the size of each bitmap, so that bitmap operations can be efficiently accelerated by hardware. The other optimization is to reduce the number of memory accesses per input character by exploiting network traffic statistics.

A. Advanced Bitmap Compression

To efficiently implement bitmap operations such as the *pop_count* instruction, we must limit the overall size of the bitmap table. There are two ways to achieve this goal: i) reduce the number of bitmaps and ii) reduce the size of each bitmap.

In the worst case, the number of bitmaps $\boldsymbol{K}$ is equal to the number of states $\boldsymbol{N}$. Such a large number of bitmaps are not feasible for efficient hardware implementation. However, with extensive experiments on real-life regular expression data sets, we find that in most of the case $\boldsymbol{K \ll N}$. Table II shows the statistics from two typical real-life regular expression rule sets. From this table we can see that the number of bitmaps is very small. Even for the large DFA with 56,280 states (generated by all Snort [19] string signatures), the number of bitmaps is only 112. Similar experimental results are obtained from all data sets publicly available at [20]. Further study on state/bitmap mapping shows most states share a very small number of bitmaps. In Figure 10, we can see that more than 90% states share only 10% unique bitmaps. This means most of state transitions only need the operation on about 10~20 bitmaps.

The number of bitmaps can be made even smaller by *bitmap combination*. Two bitmaps can be combined into one bitmap by "OR" each bit of them. Assuming only the $\boldsymbol{m}$th bit of two bitmaps, $\boldsymbol{bmp_1}$ and $\boldsymbol{bmp_2}$, is different, and the value of the bit value is 1 for $\boldsymbol{bmp_1}$ and 0 for $\boldsymbol{bmp_2}$. If we use the combined bitmap $\boldsymbol{bmp_0} = \boldsymbol{bmp_1}$ OR $\boldsymbol{bmp_2}$ to compress the states corresponding to $\boldsymbol{bmp_2}$, the $\boldsymbol{m}$th transitions of the states will become a unique transition and thus cannot be removed from leader states. Although bitmap combination increases the number of unique transitions in the intra-state compression step, the overall number of transitions tends to be even smaller because with fewer bitmaps more states can be grouped together to take the inter-state compression (see Table III for experiments on real-life rule sets).

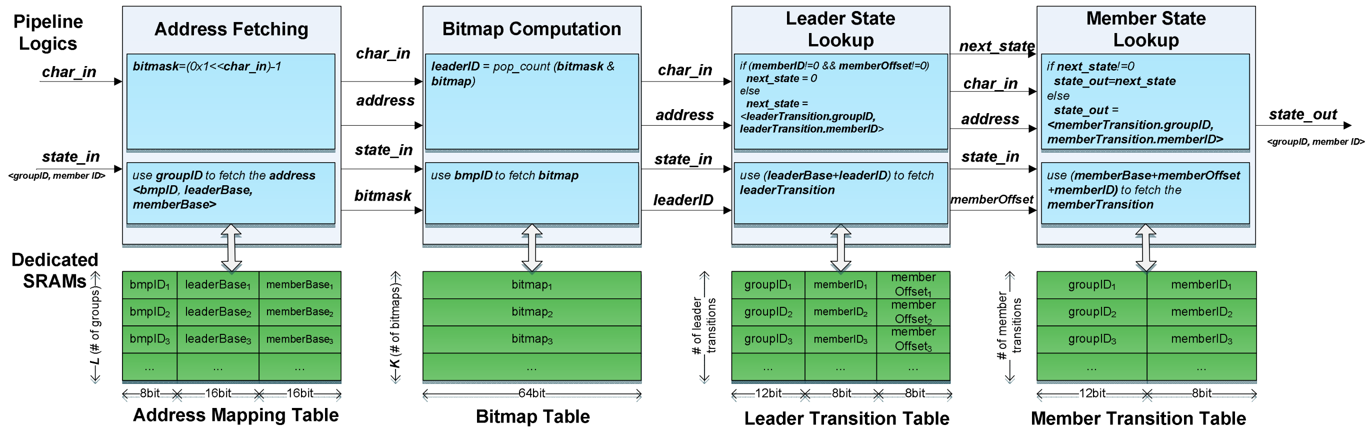

Pipeline Logics
char_in
state_in
<groupID, member ID>
Address Fetching
bitmask=(0x1<<char_in)-1
use groupID to fetch the address <bmpID, leaderBase, memberBase>
char_in
address
state_in
bitmask
Bitmap Computation
leaderID = pop_count (bitmask & bitmap)
use bmpID to fetch bitmap
char_in
address
state_in
leaderID
Leader State Lookup
if (memberID!=0 && memberOffset!=0) next_state = 0 else next_state = <leaderTransition.groupID, leaderTransition.memberID>
use (leaderBase+leaderID) to fetch leaderTransition
next_state
char_in
address
state_in
memberOffset
Member State Lookup
if next_state!=0 state_out=next_state else state_out = <memberTransition.groupID, memberTransition.memberID>
use (memberBase+memberOffset +memberID) to fetch the memberTransition
state_out
<groupID, member ID>
Dedicated SRAMs
L (# of groups)
bmpID1 leaderBase1 memberBase1
bmpID2 leaderBase2 memberBase2
bmpID3 leaderBase3 memberBase3
8bit 16bit 16bit
Address Mapping Table
K (# of bitmaps)
bitmap1
bitmap2
bitmap3
64bit
Bitmap Table
# of leader transitions
groupID1 memberID1 member Offset1
groupID2 memberID2 member Offset2
groupID3 memberID3 member Offset3
12bit 8bit 8bit
Leader Transition Table
# of member transitions
groupID1 memberID1
groupID2 memberID2
groupID3 memberID3
12bit 8bit
Member Transition Table

Figure 13. Algorithm mapping on hardware

In addition, we can use the hierarchical bitmap scheme proposed in [21] to reduce the size of each bitmap. As shown in Figure 12, the original 16-bit bitmap can be replaced by a 8-bit hierarchical bitmap in expense of duplicating certain transitions. Experimental results show that using 64-bit bitmap can achieve nearly the same compression ratio as the 256-bit scheme.

B. *Optimizing Inter-state Compression*

According to the FEACAN algorithm, the leader state can be any one of the states in the group. Optimized leader selection can greatly improve the search speed because transitions between leader states take only one memory access for the next state. From Figure 11 we can see that, more than 90% of input characters traverse less than 10% states. Therefore, we can select those most frequently traversed states as the leader states. Pseudo code for optimized grouping algorithm is shown below:

```
PROCEDURE Grouping(bitmaps, states, thresh);
while (TRUE)
    for (state s • states && s.done == NO)
            allover = NO;
            if (s.hotval > thresh)
                    s.done = YES; break;
            else if (hotover == YES) s.done = YES; break;
            else allover = YES; continue;
            end if
    end for
    if (s ∉ states)
            if (allover == NO)
            hotover = YES; continue;
            else break;
            end if
    else
            add s in a new group G as the leader state;
    end if
    for (state t ∈ states && group.size < max_size)
            if (t.bitmapID == s.bitmapID && t.done == NO
               && t.hotval <= thresh)
                    add state t in G as a member state;
                    if (diversity(G) < max_div) t.done = YES;
                    else delete t from G;
                    end if
            end if
    end for
end while
```

V. FEACAN HARDWARE DESIGN

The goal of FEACAN hardware design is to meet the wires-speed and low-latency front-end processing requirement. In this section, we first propose a low-latency algorithm mapping for a single lookup engine. Then we propose the architecture for parallel processing to achieve scalable performance. We also present dynamic solution for online update.

A. *Algorithm Mapping*

The hardware design for the FEACAN lookup engine uses a 4-stage pipeline mapping shown in Figure 13. The input of this engine is the current state ***state_in*** and the input character ***char_in***. The output of the engine is the next state ***state_next***.

In the first pipeline stage, the hardware fetches the addresses according to the ***groupID*** in ***state_in***, including the bitmap (***bitmapID***), leader state base address (***leaderBase***) and the member states base address (***memberBase***). The ***L***-entry address table is stored in block RAMs. In the second pipeline stage, the hardware uses the ***bitmapID*** to fetch the bitmap, and then compute the ***leaderID*** with bit operations. In the third pipeline stage, the hardware reads the ***leaderTransition*** at the address of (***leaderBase***+***leaderID***) and then decides whether the next state ***state_out*** has been found by checking the values of ***memberID*** and ***memberOffset***. If found, set the ***state_out*** to be the <***groupID***, ***memberID***> pair in ***leaderTransition***. Otherwise, set ***state_out*** as 0. In the last pipeline stage, if ***state_out*** is 0, the hardware reads the ***memberTransition*** at (***memberBase***+ ***memberOffset***+ ***memberID***) and set ***state_out*** to be the <***groupID***, ***memberID***> pair in ***memberTransition***.

Because each pipeline can be done by hardware logics within a single clock cycle, the state transition for each input character can be completed within only 4 clock cycles.

To support online update, the SRAMs in the lookup engine can be rewrite by inserting write bubbles [22]. New data structures of each pipeline stage are computed offline. When an update is initiated, a write bubble is inserted into the pipeline. According to Figure 14, each write bubble is assigned with a

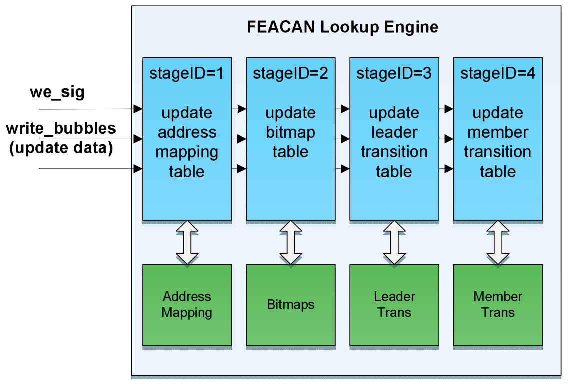


Figure 14. Online update

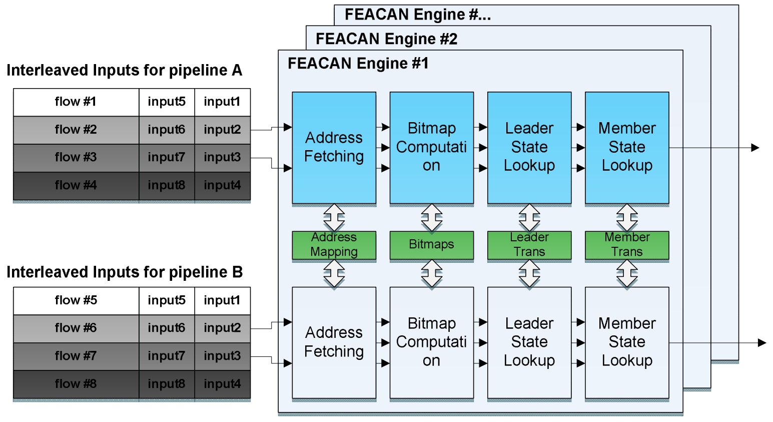


Figure 15. Parallel processing in FEACAN lookup engines

stageID. If the write enable signal ***we_sig*** is set, the write bubble will use the new content from ***state_in*** to update the memory at the stage specified by the ***char_in***. This update mechanism supports on-the-fly memory update.

B. Architecture Improvement

We can use the input-interleaving technique proposed in [23] to hide the 4-cycle latency. Instead of feeding the engine with the byte stream from a single packet payload, we can use 4 bytes from different flows as consecutive inputs characters for the lookup engine, making each pipeline has new input at every clock cycle. Therefore, we can hide the latency and achieved one cycle per character performance.

Based on the advanced feature of modern SRAM technology, we can further improve the performance by parallel search. Because modern SRAMs support dual-port reads, i.e. two memory reads can be done in a single clock cycle, a dual-lookup pipeline can be implemented to achieve a 2x speedup. Figure 15 shows the overall architecture for parallel processing. Each lookup engine has two parallel pipelines with dual-port SRAM support, and multiple lookup engines can work in parallel to achieve scalable performance.

VI. PERFORMANCE EVALUATION

A. Test Bed and Data Sets

The regular expression rules used in our test are all publicly available real-life rule sets, including both regular expression rules snort24.re (24 rules from Snort [19]), bro217.re (217 rules from Bro [24]) and string matching rules snortPart.str (1,982 short signatures from Snort), snortAll.str (all 5,766 signatures from Snort). We also use real-life network traffic traces for statistical performance evaluations. The ap020.tr and ap060.tr traces are from university campus network. The c20010.tr trace is from a large enterprise network. The inside.tr and outside.tr traces are publicly available at MIT Lincoln Lab [25]. The trace0seg.tr and trace15seg.tr traces are from the gateway of a research center with about 1,000 users.

Algorithm evaluation is done on a 3.0 GHz dual-core XEON server with CentOS 5.0 operating system. Hardware simulation is based on a Xilinx Virtex-6 FPGA device (XC6VSX475T) with 7,640 Kb Distributed RAM and 38,304 Kb block RAMs [26].

TABLE IV. AVERAGE TRANSITIONS PER INPUT CHARACTER (BRO217.RE)

Data sets	*DFA*	*D^2FA*	*D^2FA imprvd*	*FEACAN*
ap020	1	121.86	121.81	1.01
ap060	1	120.05	120.00	1.01
C20010	1	85.89	85.59	1.02
inside	1	82.97	82.70	1.02
outside	1	86.11	84.81	1.03
trace0seg	1	123.33	123.27	1.01
trace15seg	1	122.87	122.82	1.01

B. Algorithm Evaluation

The algorithms used in our test for performance comparison include the original DFA, D^2FA by Kumar et al. and the improved D^2FA by Becchi et al. For convenience, these algorithms are denoted as **DFA**, **D^2FA**, and **D^2FA improved** in our test. The performance metrics in our experiments include: i) **memory access** for speed, ii) **memory usage** for storage, and iii) **preprocessing time** for update.

1) Memory Access

Because all these algorithms are memory-based solutions, the efficiency of memory access determines the lookup performance. For fair performance comparison, we set the depth of default path in D^2FA as 2, making it has the same worst case in search as FEACAN. Figure 14 and Figure 15 show the average number of states traversals per input character on two different regular expression rule sets bro217.re and snortPart.str. From these figures we can see that FEACAN achieves near 1 traversal/state on both data sets, i.e. most of the state transitions take place only among leader states. In comparison, both D^2FA and D^2FA improved require more state traversals (15%~75%) and the D^2FA improved algorithm is not stable when tested with different input trace files.

Although counting state traversals can show the number of average memory accesses, the size of each memory access is unknown because each state may have different size (number of transitions). So instead, we use the average number of transitions traversed per each input character to show the memory bandwidth requirement of each algorithm. From Table IV we can see that, the average number of transitions traversed

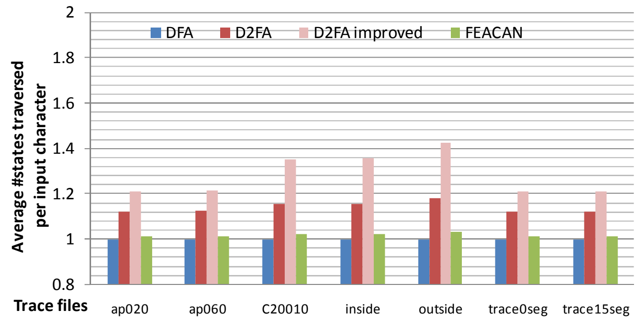


Figure 14. Memory access comparison on *bro217.re*

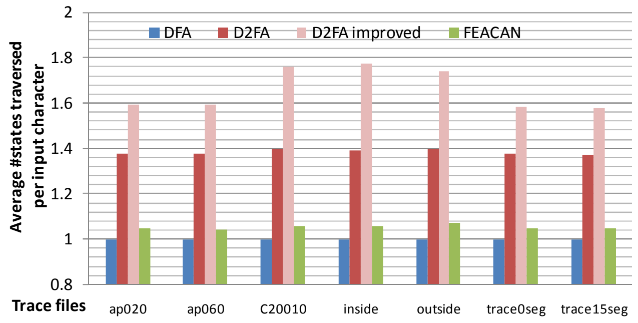


Figure 15. Memory access on *snortPart.str*

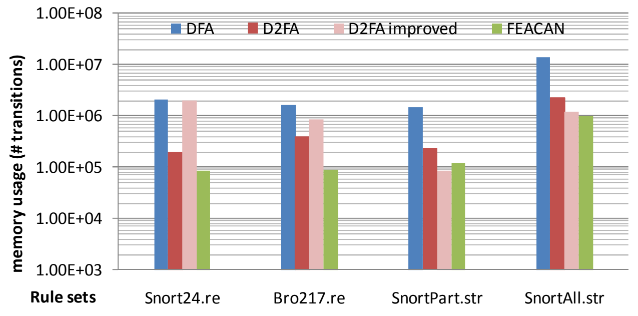


Figure 16. Memory usage comparison

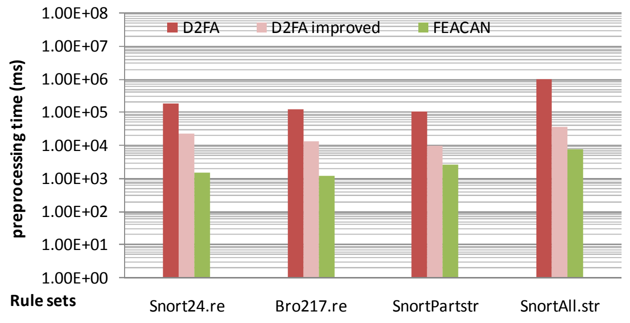


Figure 17. Preprocessing time comparison

per character of D^2FA and D^2FA improved is about 100 (due to the linear search). If the size of a single transition is 3 bytes (24 bits), these two algorithms then need 100*24*10Gbps = 24Tbps memory bandwidth to support the wire-speed processing at a 10Gbps link. In comparison, assuming we use 64-bit bitmap, the FEACAN algorithm only requires about 1*(24+64)*10Gbps = 880Gbps bandwidth, which is achievable with modern on-chip block RAMs [26].

2) *Memory Usage*

In this experiment, we measure the total number of transitions of each algorithm for memory usage comparison. From Figure 16 we can see that FEACAN achieves more than 90% compression ratios compared to the original DFA algorithm. Even with the largest snortAll.str rule set, the memory usage of FEACAN is less than 2MB. Compared to D^2FA and D^2FA improved, FEACAN also has superior memory compression in most of the cases. Although D^2FA-based algorithms can achieve better compression ratio with long default path (e.g. achieving 95% compression ratio with a worst-case depth 12), efficient implement of long default path in hardware is difficult because deep pipeline in hardware will result in high processing latency and complicated memory management, which is not feasible for front-end processing.

3) *Preprocessing Time*

Preprocessing time is important for system management. It determines the response time of a network system to react to new policies. Experimental results in Figure 17 show the preprocessing time for DFA compression (not including the regular expression to DFA conversion time) by D^2FA, D^2FA improved and FEACAN. The preprocessing time of FEACAN is only about 1/100 and 1/10 of that of D^2FA and D^2FA improved on all data sets. For the largest rule set snortAll.str, FEACAN preprocessing time is less than 10 seconds. In comparison, D^2FA improved needs about 2 minutes and D^2FA requires more than 20 minutes. The performance evaluation is in accordance with the complexity analysis. D^2FA has

C. Hardware Evaluation

The FEACAN lookup engine is implemented on a Xilinx Virtex-6 chip for hardware performance evaluation. Because a single pipeline in the FEACAN engine processes one byte per clock cycle, its performance can be derived from the maximum clock rate. With the timing analysis on ISE simulator, we notice that the maximum clock rate depends on the size of the bitmap and the implementation of the *pop_count* instruction. We can achieve a maximum clock rate of 156.006 MHz for the 256-bit bitmap implementation and 227.583 MHz for 32-bit bitmaps. Based on the 150 MHz FPGA implementation, each FEACAN lookup engine can achieve $150MHz * 8bits * 2 = 2.4Gb$ throughput. With modern ASIC (application specific integrated circuit) technologies, a modest clock rate of 500 MHz can be achieved [17]. Therefore, the throughput achieved by each FEACAN engine with ASIC implementation is $500MHz * 8bit * 2 = \ 8\,Gbps$.

We can achieve scalable performance by using parallel FEACAN lookup engines. Because FEACAN is a memory-based design, the number of parallel engines is limited by the overall size of on-chip SRAMs available on the FPGA platform. Table V shows the maximum number of engines can be implemented on a single Virtex-6 chip with about 5MB block RAMs. From this table we can see that FEACAN can achieve

10~40 Gbps throughput on FPGA device and over 100 Gbps on ASIC chips.

TABLE V. HARDWARE PERFORMANCE OF FEACAN LOOKUP ENGINE

Rule set	# of engines supported	FPGA (150MHz)	ASIC (500MHz)
Snort24.re	15	36.0 Gbps	120 Gbps
Bro217.re	17	40.8 Gbps	136 Gbps
SnortAll.str	4	9.6 Gbps	32 Gbps

VII. CONCLUSION

Content-aware processing at the front end of distributed network systems is challenging due to the wire-speed, low-latency and stateful processing requirement. Existing work for fast content-ware network processing mainly focuses on algorithmic solutions while lacking system-level design to meet the critical requirement for front-end processing. In this paper, we propose a system-level solution named FEACAN to accelerate front-end content-aware network processing by employing a software-hardware co-design method. A two-dimensional DFA compression algorithm is designed to reduce the memory usage and a low-latency lookup engine is presented for wire-speed performance. Experimental results on real-life data sets and network traffic show that FEACAN can achieve more than 100 Gbps processing speed based on state-of-the-art hardware technologies.

REFERENCES

[1] PICMG, AdvancedTCA, http://www.picmg.org/

[2] Netronome, Product Brief - NFE-i8000 Network Acceleration Card, 2006, http://www.netronome.com/

[3] NetLogic, XLP832 http://www.netlogicmicro.com/Products/MultiCore/XLP.htm

[4] Cavium, OCTEON5860, http://www.cavium.com/OCTEON_MIPS64.html

[5] LSI Tarari T2000/T2500, http://www.lsi.com/networking_home/networking_products/tarari_content_processors/

[6] NetLogic, NETL7, http://www.netlogicmicro.com/Products/Layer7/Layer7.htm

[7] R. Sommer, V. Paxson, Enhancing Byte-Level Network Intrusion Detection Signatures with Context, in Proc. of ACM CCS, 2003.

[8] J. E. Hopcroft and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 1979.

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

[10] R. W. Floyd, and J. D. Ullman, The Compilation of RegularExpressions into Integrated Circuits, Journal of ACM, vol. 29, no. 3, pp 603-622, July 1982.

[11] R. Sidhu and V. K. Prasanna, Fast Regular Expression Matching using FPGAs, in Proc. of FCCM, 2001.

[12] C. R. Clark and D. E. Schimmel, Efficient Reconfigurable Logic Circuit for Matching Complex Network Intrusion Detection Patterns, in Proc. of FPL, 2003.

[13] J. Levandoski, E. Sommer, and M. Strait, Application Layer Packet Classifier for Linux, http://l7-filter.sourceforge.net/.

[14] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection, in Proc. of ACM SIGCOMM, 2006.

[15] M. Becchi and P. Crowley, An Improved Algorithm to Accelerate Regular Expression Evaluation, in Proc. of ACM/IEEE ANCS, 2007.

[16] N. Tuck, T. Sherwood, B. Calder and G. Varghese, Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection, in Proc. of IEEE INFOCOM, 2004.

[17] B. C. Brodie, R. K. Cytron. nd D. E. Taylor, A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching, in Proc. of ISCA, 2006.

[18] S. Kumar, J. Turner and J. Williams, Advanced Algorithms for Fast and Scalable Deep Packet Inspection, in Proc. of ACM/IEEE ANCS, 2006.

[19] Snort Open Source IDS/IPS, http://www.snort.org

[20] M. Becchi, Regular Expression Processor, http://regex.wustl.edu/

[21] Y. Qi, B. Xu, F. He, B. Yang. J. Yu, J. Li, Towards High-performance Flow-level Packet Processing on Multi-core Network Processors, in Proc. of ACM/IEEE ANCS, 2007.

[22] Q. Wang and V. K. Prasanna, Multi-Core Architecture on FPGA for Large Dictionary String Matching, in Proc. of FCCM, 2009.

[23] A. Basu and G. Narlikar, Fast Incremental Updates for Pipelined Forwarding Engines, in Proc. of IEEE INFOCOM, 2003.

[24] http://www.bro-ids.org/

[25] DARPA intrusion detection data sets, http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html

[26] http://www.xilinx.com/products/virtex6/

[27] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman and R. H. Katz, Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection, in Proc. of ACM/IEEE ANCS, 2006.

[28] M. Becchi and P. Crowley, A Hybrid Finite Automaton for Practical Deep Packet Inspection, in Proc. of CoNEXT, 2007.

[29] S. Kumar, B. Chandrasekaran, J. Turner and G. Varghese, Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia, in Proc. of ACM/IEEE ANCS, 2007.

[30] R. Smith, C. Estan and S. Jha, XFA: Faster Signature Matching with Extended Automata, in Proc. of IEEE Symposium on Security and Privacy, 2008.

[31] R. Smith, C. Estan and S. Jha and S. Kong, Deflating the big bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata, in Proc. of ACM SIGCOMM, 2008.

[32] M. Becchi, C. Wiseman and P. Crowley, Evaluating Regular Expression Matching Engines on Network and General Purpose Processors, in Proc. of ACM/IEEE ANCS, 2009.