Scalable Multi-Pipeline Architecture for High Performance Multi-Pattern String Matching

Weirong Jiang, Yi-Hua E. Yang and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
Email: {weirongj, yeyang, prasanna}@usc.edu

***Abstract*—Multi-pattern string matching remains a major performance bottleneck in network intrusion detection and anti-virus systems for high-speed deep packet inspection (DPI). Although Aho-Corasick deterministic finite automaton (AC-DFA) based solutions produce deterministic throughput and are widely used in today's DPI systems such as Snort [1] and ClamAV [2], the high memory requirement of AC-DFA (due to the large number of state transitions in AC-DFA) inhibits efficient hardware implementation to achieve high performance. Some recent work [3], [4] has shown that the AC-DFA can be reduced to a character trie that contains only the forward transitions by incorporating pipelined processing. But they have limitations in either handling long patterns or extensions to support multi-character input per clock cycle to achieve high throughput. This paper generalizes the problem and proves formally that a linear pipeline with H stages can remove all cross transitions to the top H levels of a AC-DFA. A novel and scalable pipeline architecture for memory-efficient multi-pattern string matching is then presented. The architecture can be easily extended to support multi-character input per clock cycle by mapping a compressed AC-DFA [5] onto multiple pipelines. Simulation using Snort and ClamAV pattern sets shows that a 8-stage pipeline can remove more than 99% of the transitions in the original AC-DFA. The implementation on a state-of-the-art field programmable gate array (FPGA) shows that our architecture can store on a single FPGA device the full set of string patterns from the latest Snort rule set. Our FPGA implementation sustains 10+ Gbps throughput, while consuming a small amount of on-chip logic resources. Also desirable scalability is achieved: the increase in resource requirement of our solution is sub-linear with the throughput improvement.**

***Keywords*-Deep packet inspection; DFA; FPGA; pipeline; string matching;**

I. INTRODUCTION

Deep packet inspection (DPI) systems (e.g. Snort [1] and ClamAV [2]) are an effective mechanism for detecting various network threats such as intrusion, virus and spam. The functions of DPI systems rely on *multi-pattern string matching* which scans the input stream to find all occurrences of a predefined set of string-based patterns rather than a single pattern [6], [7]. Due to the explosive growth of network traffic, multi-pattern string matching has been a major performance bottleneck in DPI systems which have to scan the incoming traffic in real time on fast links (e.g. 10 Gbps Ethernet and beyond) [8], [9]. For example, it has been reported that the string matching time accounts for 40% to 70% of the Snort running time [10]. Some existing solutions employ a number of identical instances of the basic engine to process multiple input streams so that the aggregate throughput is increased [9], [11]. But it remains a challenge to improve the per-stream throughput as the worst case when there is only one single stream. Though it is possible to split a stream into several sub-streams with partial overlap among the sub-streams, additional complexity is introduced in scheduling, buffering, and ordering [12]. Simple and efficient hardware-based multi-pattern string matching engines become a necessity for high-speed DPI systems [6]. However, following requirements must be met for the hardware-based solutions to be efficient and practical:

1) Deterministic throughput: To keep the DPI system itself to be robust, the string matching engine should preserve high throughput independent of the characteristics of the input stream or of the pattern set [13], [14]. Aho-Corasick deterministic finite automaton (AC-DFA) based solutions are thus preferred over those heuristic-based schemes [15] in most of today's string matching engines, as the worst-case processing time complexity of AC-DFA is linear with the length of the input stream [6].

2) Low memory requirement: The size of the pattern sets in many DPI systems is increasing rapidly. Those large pattern sets can require enormous amount of memory, which has to be placed off-chip. The low speed to access large (external) memory becomes a major bottleneck of the string matching engine. Hence it has recently been an active research topic to reduce the memory requirement for AC-DFA so that it can be fit in the on-chip memory to minimize the memory access time [14].

3) Dynamic update: Since the pattern set in DPI systems is frequently updated (by adding or removing patterns), the multi-pattern string matching engine should support dynamic updates without major performance degradation. Such a requirement can be very critical for some urgent events such

This work is supported by the United States National Science Foundation under grant No. CCF-0702784. Equipment grant from Xilinx Inc. is gratefully acknowledged.

as a new virus breakout. Hence memory-based architectures are usually preferred over purely logic-based solutions [15].

4) Supporting all types of strings: The string patterns in DPI systems such as Snort and ClamAV are specified not only in ASCII or English alphabet, but also as binary or hexadecimal strings. This results in a larger problem space than the traditional string matching problem, since each character can represent 256 distinct values. As a result, those schemes to reduce the memory requirement by character encoding such as converting all characters into lower cases [3] become infeasible or less effective for DPI systems.

5) Implementation cost: When implemented in hardware, the overall resource usage of the string matching engine should be predictable and minimized. A string matching engine usually operates together with other DPI processing engines such as the packet header classifier [16] and regular expression matching engine [17] on the same chip. High implementation cost can affect the practical performance [6] and limit the scalability of the overall system.

To the best of our knowledge, none of the existing hardware-based solutions has met all the above requirements. Taking advantage of pipelining and parallelism, this paper proposes a memory-based multi-pipeline architecture for high-speed, low-cost multi-pattern string matching. We make the following contributions:

- Motivated by recent work on using pipelining to reduce the transition edges in a AC-DFA [3], [4], we generalize the problems and proves formally that a linear pipeline with H stages can remove all cross transitions to the top H levels of a AC-DFA. Experiments with real-life pattern sets show that using $H = 8$ stages eliminates most of the cross transition edges. Unlike the previous pipelined solutions, our work removes the limitation of the maximum pattern length that is supported.
- Our pipeline architecture is simple and shown to be easily extended to support multi-character input per clock cycle by mapping a compressed AC-DFA [5] onto multiple linear pipelines. The sustained per-stream throughput is thus improved multiplicatively, while the increase in resource requirement is sub-linear with the throughput improvement.
- FPGA implementation results show that, our architecture can store on a single FPGA device over 9K string patterns from the latest Snort rule set. It sustains 10+ Gbps throughput, while consuming only 10% of the on-chip logic resources. As far as we know, this work is the first FPGA design based on memory rather than logic, that supports the full set of Snort string patterns while sustaining 10+ Gbps throughput.

The rest of the paper is organized as follows. Section II gives a brief review of the related work on multi-pattern string matching and revisits the AC-DFA algorithm. Section III analyzes the effectiveness of pipelining from both theoretical and experimental points of view. Section IV discusses the extension to support multi-character input per clock cycle. Section V presents our multi-pipeline architecture. Section VI evaluates the performance of the implementation. Section VII concludes the paper.

II. BACKGROUND

Pattern (string) matching is a classical problem and has been studied in various contexts [6], [7]. For clarity, we define the terminology used throughout this paper:

- *Pattern / String*: pattern matching is a broader problem than string matching where the patterns are specified as fixed strings. However, in this paper a "pattern" refers only to a string-based pattern, and the terms "pattern" and "string" will be used interchangeably within this paper.
- *Multi-pattern string matching* is a specific type of string matching used in DPI systems to search an input stream for a set of patterns rather than a single pattern.
- *Pattern length* is defined as the number of characters in a pattern. As the basic element of a string, each character consists of 1 byte i.e. 8 bits, and can be specified in either ASCII or binary formats.
- The *size* of a pattern set is defined as the number of patterns in it.

A. Related Work

Although string matching is a classic problem for decades, multi-pattern string matching has sparked renewed research interest due to its application in DPI systems [6]. Some excellent surveys can be found in [6], [18]. Based on the platform for implementation, the state-of-the-art solutions can be generally divided into three categories: multi-core processor -based [9], [11], [19], application-specific integrated circuit (ASIC) -based [5], [8], [12], [20] and field programmable gate array (FPGA) -based [4], [15], [21], [22] solutions. Each of them has its own pros and cons. Advanced multi-core processor -based solutions [9], [11], [19] have recently stepped in as a major player for high performance string matching. They can improve the aggregate throughput dramatically by using a large number of threads to process multiple input streams in parallel. On the other hand, it has been observed that the memory access pattern in string matching is irregular [9], [11]. This results in relatively low *per-stream* throughput which is critical for real-time network traffic processing in the worst case. Though it is possible to split an input stream into several sub-streams with partial overlap among the sub-streams, additional complexity is introduced in scheduling, buffering, and ordering [12]. ASIC-based solutions [5], [8], [12], [20] provide impressive high per-stream throughput while their applicability is limited by the high implementation cost and low reprogrammability. Combining the flexibility of software and the near-ASIC performance, FPGA technology has long become an attractive option for implementing various real-time network processing engines [15], [16], [23]. State-of-the-art FPGA devices such as Xilinx Virtex-6 [24] provide high clock rate and large amounts of on-chip dual-port memory with configurable word width. Hence this paper considers FPGA as the target platform for implementation, though the proposed architecture can also be implemented efficiently in ASIC.

The majority of existing FPGA-based string matching engines are based on purely logic [21], [22], [25]. Although they provide desirable high performance, it takes considerable time to resynthesize the design and reprogram the FPGA device. In case of pattern updates, the hardware-wired string matching engine has to be offline; thus it is unable to detect network intrusion or virus during that period. Hence we resort to memory-based architectures which support dynamic updates at run time. Like most of the existing memory-based architectures [4], [8], [12], our work is based on Aho-Corasick (AC) algorithm [26] for its ability to provide deterministic throughput. We revisit the AC algorithm in the next section to unveil the source of its memory inefficiency which is the key limitation to use FPGAs for large-scale string matching [11].

B. Revisiting Aho-Corasick Algorithm

Aho-Corasick (AC) algorithm [26] is one of the earliest algorithms in multi-pattern string matching. Among all variants of AC algorithms, AC-DFA is widely adopted for its deterministic throughput. AC-DFA converts a pattern set which contains n characters into a deterministic finite automaton with $O(n)$ states. Once the DFA which can be stored as a state transition table is built, it reads in the input stream one character per clock cycle. Each input character is processed only once and results in exactly one state transition. Thus it takes $O(m)$ time to process an input stream consisting of m characters.

Figure 1 shows an example of the AC-DFA construction for the three patterns: “SHIP”, “HIS” and “IN”. AC-DFA starts with constructing a trie (we call it *AC-trie*), where the root is the default non-matching state. Each pattern to be matched adds states to the trie, one state per character, starting at the root and going to the end of the pattern. The transition edge between two states is stored as a *goto* function: $g(s_1, c) = s_2$, which means state s_1 receiving the character c will switch to state s_2. In the example shown in Figure 1 (a), $g(3, P) = 4$.

The AC-trie is then traversed and failure transitions are added for each state. The failure transition of a state is used in case the goto function reports *fail* i.e., the string matching on the current traversed path is not found. Initially all nodes have the default failure transitions to the root. But it is possible that the suffix of the previously matched string is the prefix of another string in the trie. Hence the failure transitions are updated to reuse the information associated with the last input characters (suffix) to recognize patterns

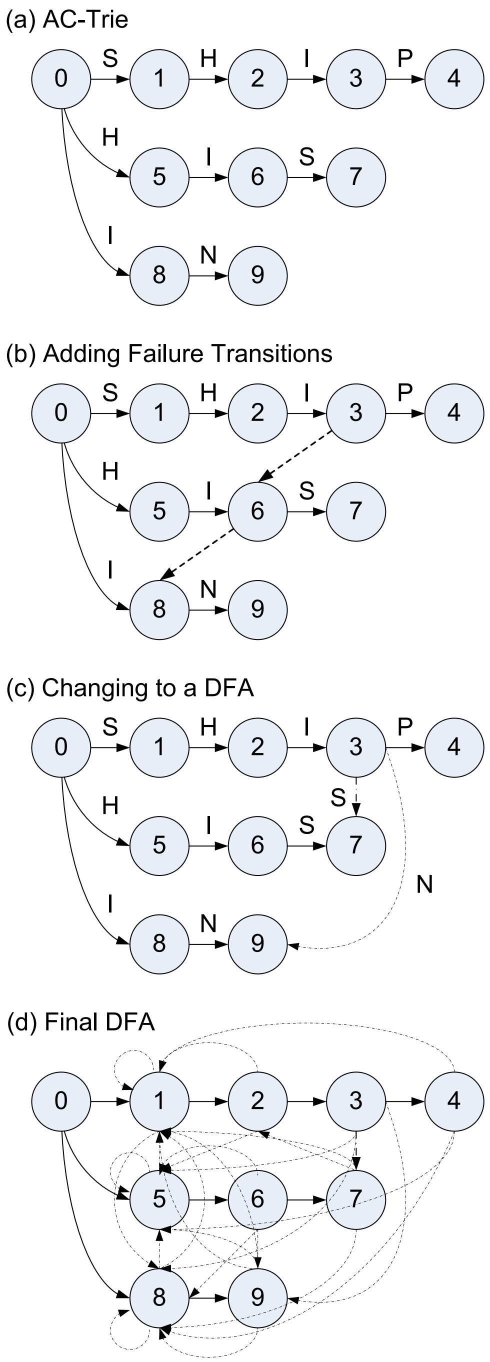

Figure 1. Constructing the AC-DFA for the pattern set: {SHIP, HIS, IN}.

which begin with that suffix, without restarting from the root. To reuse as much history information as possible, the failure transition of a node (denoted $N1$) is from it to such a node (denoted $N2$) that the path from the root to $N2$ is the longest prefix equal to the suffix ending at $N1$. The failure transitions are represented as $failure$ functions: $f(s_1) = s_2$ indicating there is a failure transition from state s_1 to state s_2, and we define s_2 as the *failure state* of s_1. Taking Figure

1 (b) as an example, two prefixes “HI” and “I” are both the suffixes of “SHI”, i.e., both states 6 and 8 are the candidates for the failure state of state 3. But since “HI” is longer, the failure transition of state 3 is to state 6, i.e. $f(3) = 6$.

The AC-trie with failure transitions (we call it *AC-fail*) is not a DFA yet, since an input character may invoke multiple failure transitions. For example in Figure 1 (b), when state 3 receives the character “N”, its own goto function will report *fail*. Then, via its failure transition, state 3 transits to state 6 which will check its goto function to see if the character “N” can be accepted. Since state 6’s goto function still reports *fail*, state 6 has to consult its failure function which directs it to state 8. Finally state 8 accepts “N” and transits to state 9. In other words, it takes 2 failure transitions and 1 goto transition for the input character “N” to be accepted. To convert the AC-fail into a DFA, Aho and Corasick [26] propose to combine the *failure* function with the *goto* function to obtain *next-move* functions: $\delta(s_1, c) = s_2$, which means state s_1 receiving the character c will switch to state s_2. In the above example, $\delta(3, N) = \delta(f(3) = 6, N) = \delta(f(6) = 8, N) = 9$, as shown in Figure 1 (c).

Figure 1 (d) shows the complete DFA after adding all transitions. For the sake of readability, we do not show the default transitions to the root and remove all the labels on the transitions. Unlike in a AC-fail, one character from the input stream results in exactly one transition in a AC-DFA. But the cost is that the memory requirement becomes higher due to the increasing number of transition edges. For example, state 3 in Figure 1 (d) will have 6 transitions, while it has only 2 transitions in Figure 1 (b). Such an increase in the number of transitions is caused by transition duplication due to the combination of the failure transitions with the goto transitions. For example, since $f(3) = 6$, $f(f(3)) = 8$, $f(f(f(3))) = 0$, the goto transitions of states 6, 8, 0 may be all copied to state 3.

Now we have the following definitions which will be used in the rest of this paper:

- The *depth* of a state in a AC-DFA is the directed distance from the root to that state. For example, the depth of state 6 in Figure 1 (d) is 2. The depth of the root is always zero.
- The ith *level* of a AC-DFA includes all the states whose depth is i, $i = 0, 1, \cdots$. For example in Figure 1 (d), states 1, 5, 8 are all in level 1.
- The *depth* of a AC-DFA, denoted L, is defined as the number of distinct levels (excluding level 0) in the AC-DFA. According to the AC-DFA construction procedure, the depth of a AC-DFA is equal to the length of the longest pattern contained in the AC-DFA.
- In a AC-DFA, the *transition* between states is represented as a directed *edge* from one state to another. The two terms “transition” and “edge” are used interchangeably within this paper.
- The transitions generated by the goto functions are called the *forward* transitions. A forward transition is always from a state in level i to another state in level $i + 1$, $i = 0, 1, \cdots, L - 1$. For example, the transitions shown in solid lines in Figure 1 (d) are forward transitions.
- Apart from the forward transitions, the rest of the transitions in a AC-DFA are called the *cross* transitions. A cross transition is always from a state in level i to another state in level j where $j \leq i$, $i, j = 0, 1, \cdots, L$.

C. Reducing AC-DFA Memory Requirement

Currently there are two general approaches to reduce the memory requirement of AC-DFA. One is to minimize the number of states and the other is to minimize the number of transitions. In most cases these two approaches are orthogonal to each other and can be used together to achieve higher performance.

Lin et al. [27] exploit the similarity between different states in a AC-DFA. The number of states is reduced by merging pseudo-equivalent states while maintaining the correctness of string matching. But the memory reduction achieved depends on the characteristics of the pattern set and is quite limited (e.g. 29% memory reduction is achieved for 1,595 Snort string patterns [27]).

Alicherry et al. [5] divide each pattern into W-byte blocks which are then used to construct a AC-DFA. This results in a “compressed” AC-DFA with fewer states and transitions. Meanwhile, the throughput can be improved by W times by running W instances of the AC-DFA in parallel, each of which accepts the same input data stream with an one-character offset (to ensure that no pattern is missed). However, since their architecture is based on ternary content addressable memory (TCAM), the improvements in memory requirement and throughput will be partially offset by the high cost of TCAM [3].

As pointed out in [12], there is very little space to reduce the number of states. Hence a large body of work on memory-efficient string matching is to reduce the number of transitions. Tan et al. [20] propose the bit-split architecture to split a full AC-DFA into several partial state machines (PSM), each accepting a small portion (1 or 2 bits) of the input as transition labels. A partial match vector (PMV), one bit per pattern, is maintained at every state in the PSM to map the state to a set of possible matches. At every clock cycle, the PMVs from all PSMs are bitwise AND-ed to generate a full match vector (FMV) to find the actual matches. However, there are substantial overheads in implementing a large number of PSMs in real hardware, as shown in [28].

Lunteren [14] observes that a large fraction of state transitions are to the root or to the states in level 1. These transitions can be removed from the original AC-DFA by using a separate 256-entry on-chip table to keep track of them [8]. The number of transitions can be further reduced by partitioning the pattern set into multiple subsets [14]. But the performance of the partitioning scheme depends on the characteristics of the pattern set. In addition to [14], Song et al. [8] reduce more transitions by adding one buffer to "cache" the previous state. But it requires a dual-port memory to be accessed by both the cached state and the current state in parallel. The effective throughput is one character per clock cycle, which halves the potential throughput of using dual-port memory.

It is not hard to see, the number of forward transitions in a AC-DFA with n states is always $n + 1$. Hence there is little room to reduce the number of forward transitions. Some recent work [3], [4] proposes to incorporate the pipeline architecture to remove all cross transition edges of a AC-DFA. Instead of being converted into a DFA, the AC-trie is mapped onto a linear pipeline. Each trie level is mapped to a stage containing a separate memory block. At each clock cycle, the input character is carried to all stages to invoke the state transition in each stage. Meanwhile, each stage forwards the output state to its next stage. Such pipelined solutions require a L-stage pipeline for mapping a L-level AC-trie. But the number of levels in a AC-trie corresponds to the length of the longest pattern. As we can see later in Figure 4, the length of the patterns in both Snort and ClamAV can be very large, making those pipelined solutions impractical to handle long patterns. Yang et al. [4] limit the pattern length to be smaller than 64. Pao et al. [3] propose to partition the long patterns into segments. Each segment is matched in the pipeline and the segment IDs are used to build a high-level DFA for matching the entire pattern. Several small tables are needed to take care of the fragmentation when the string is matched in the middle of a segment. Such a solution complicates the overall architecture and is hard to be extended for supporting multi-character input per clock cycle to achieve high throughput. Furthermore, [3] considers only the ASCII characters, which makes their results less interesting for DPI systems.

III. Scalable Pipeline: Few Stages are Sufficient

Existing pipelined solutions aim to eliminate all cross transition edges in a AC-DFA. But this results in a deep and unscalable pipeline. Our motivation is to find the minimal pipeline depth while removing the majority of the cross transition edges. At first, we present a formal proof for the effectiveness of such pipelined solutions. We use the following definitions:

- The *depth* of a pipeline, denoted H, is the number of stages in the pipeline.
- When the *goto* function of a state receiving a character reports *fail*, we call the state is *failing*.
- When a state is failing, it means failure to match the patterns whose prefix corresponds to the path from the root to that state. We call there is a *failed matching*.

A. Correctness

In the original AC-DFA, the underlying system model is using a single memory block which contains only one active state at a time, as shown in Figure 2 (a). This single state has to keep track of all information, and attempts to reuse as much history information as possible for matching another pattern when the current matching becomes failed. In contrast, given a AC-DFA with L levels, our pipelining (shown in Figure 2 (b)) is to map the states in level i of the AC-DFA to the ith stage of a linear pipeline, $i = 0, 1, \cdots, H-1$ where H is the pipeline depth and $H \leq L$. The remaining levels of the AC-DFA are placed in another (single) memory (denoted *AC-remain*). When $H = L$, AC-remain contains no state; we call such a model the *fully* pipelined AC-DFA, which has been exploited in previous pipelined solutions [4]. This paper focuses mainly on the *partially* pipelined AC-DFA where $H < L$. Since each memory block can have one active state, H stages and AC-remain can maintain up to $H+1$ active states simultaneously. At each clock cycle, the input character is delivered to AC-remain and all the stages to invoke possibly multiple state transitions in parallel.

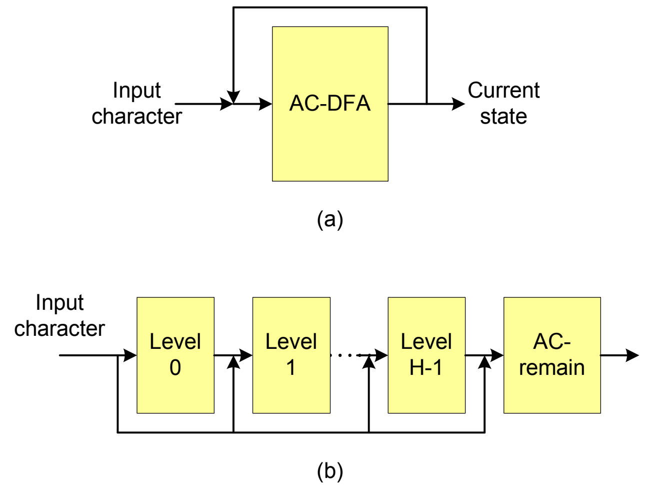

Figure 2. System models for (a) traditional AC-DFA (b) (partially) pipelined AC-DFA

Since the cross transitions in a AC-DFA are generated based on failure transitions, we first discuss why the above pipelining can help remove the failure transitions in a AC-fail (i.e. a AC-trie with failure transitions added). Given a AC-fail with L levels, a pipeline with H ($H \leq L$) stages can be built based on the above model shown in Figure 2 (b). It has following properties:

Lemma 1: K-character ($K <= H$) prefixes of patterns are matched in the first K stages.

Proof: The first K stages store actually a K-level trie. The problem is reduced to prefix matching. Hence the lemma is proved. ■

Theorem 1: All failure transitions in AC-remain to states in levels $0, 1, \cdots, H-1$ can be removed.

Proof: Recall that the failure transitions are added into a AC-trie for a failed matching to reuse the history information without restarting from the root. These history information is the last input characters (suffix) which can be the prefix of other patterns. Since the states in level i ($i = 0, 1, \cdots, L$) represent i-character prefix of the patterns, the failure transitions to those states are used for a failed matching to reuse the last i input characters.

On the other hand, in a H-stage pipeline, a state in stage i ($i = 0, 1, \cdots, H-1$) will be active if and only if the last i characters match the prefix represented by the current state. As a result, there is no need for a failed matching in AC-remain to reuse the last i input characters which have been matched by the first i stages (Lemma 1). ■

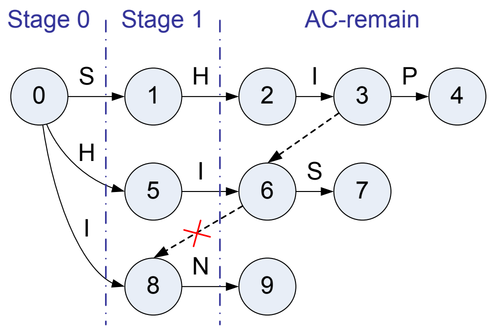

Figure 3. Eliminating failure transitions by pipelining (H=2)

Taking Figure 3 as an example where the AC-fail is mapped onto a 2-stage pipeline, suppose the input stream is "SHINE". When state 3 is the current active state of AC-remain and it reads the input character "N". This becomes a failed matching. There are two suffixes that may be reused: "HI" and "I". The longer suffix "HI" is preferred at first. State 6 which represents the prefix "HI", will be active. Since state 6 is in the same memory block with state 3, it will not be active until state 3 encounters a failed matching. Hence, the failure transition from state 3 to state 6 cannot be eliminated. Since state 6 still gets a failed matching for the input character "N", the suffix of state 6, "I", which is also the shorter suffix of state 3, will be used. At this time, state 8 which represents the prefix "I", has already been activated via the forward transition from state 0. Hence state 6 has no need to record the failure transition to state 8 to reuse the suffix.

Corollary 1: A pipeline with H stages can remove all cross transitions to states in levels $0, 1, \cdots, H$.

Proof: According to the AC-DFA construction procedure, the cross transitions to the root (i.e. the state in level 0) are generated using the failure transitions to the root and the failure transitions of the root. The cross transitions to states in level i ($i = 1, 2, \cdots$) are generated using the failure transitions to states in level $i-1$ and the forward transitions of the states in level $i-1$. In other words, if $s_2 = \delta(f(s_1), c) \neq root$, then $Level(s_2) = Level(s_1) + 1$ where $Level(s)$ denotes the level number of state s. Hence, based on Theorem 1, the corollary is proved. ■

It is possible that the state of AC-remain is not failing and at the same time AC-remain receives another non-failing state from the previous stage. But AC-remain allows only one active state. To solve the conflict, we claim that,

Theorem 2: AC-remain should keep using its own state unless (1) the goto function of its state reports *fail* and (2) a non-failing state cannot be found via the failure transitions within the AC-remain.

Proof: As we did in the proof for Theorem 1, we consider the correspondence between AC-fail and the pipelined AC-DFA. In AC-fail, as long as the *goto* function of the current state does not report *fail*, the state will not consult the failure function. The failure functions are consulted recursively until a non-failing state is found. The failure transitions to states in level i ($i = 0, 1, \cdots$) will not be taken if there is a failure transition to states in level j ($j > i$) that can find a non-failing state.

According to Theorem 1, the AC-remain in the pipelined AC-DFA keeps the failure transitions only to states in levels $H, H+1, \cdots$. When (1) the state of AC-remain becomes failing and (2) a failure transition to some state on the top H levels is needed, the AC-remain will accept the state output from the previous stage. ■

B. Analysis of Real-Life Pattern Sets

To validate the effectiveness of the proposed pipelined solution, we must answer the following two questions:

1) How much of the transitions in a AC-DFA are cross transitions?
2) How many stages are needed for eliminating most of the cross transitions?

We use the latest pattern sets from two widely-used open source DPI systems: Snort [1] and ClamAV [2]. Snort [1] is a network intrusion detection system and ClamAV [2] is an anti-virus system. The statistics of the two pattern sets are shown in Table I. The distribution of the pattern length in terms of the number of characters is shown in Figure 4.

Table I
PATTERN SETS FROM REAL-LIFE DPI SYSTEMS

	Snort	ClamAV
Version	2.80	0.95.2
Date	2009-04-21	2009-06-16
# String patterns	9033	42020
Total # characters	197298	3025497
Aver. pattern length	21.84	72.0
Max. pattern length	232	382
Min. pattern length	1	1

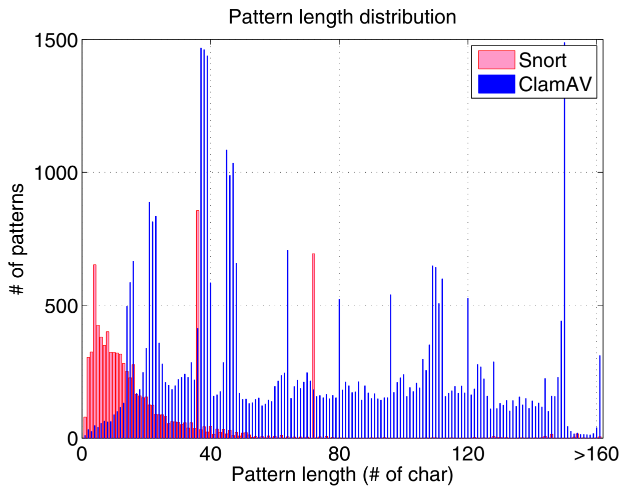

Figure 4. Pattern length distribution

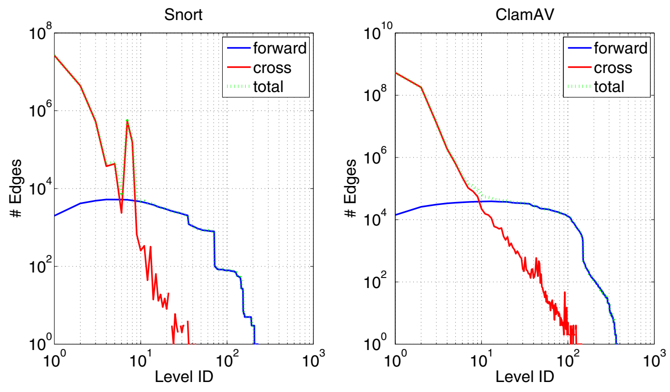

Figure 5. Number of edges on each level

For each pattern set, we built the AC-DFA and counted the number of forward transitions and of cross transitions on each level. The results are shown in Figure 5 where Logarithmic (base 10) scale is used for both the X-axis and the Y-axis. We can make the following observations:

- As a whole the cross transitions constitute the majority of the overall transitions. 99.52% and 99.61% of the transitions are cross transitions for Snort and ClamAV pattern sets, respectively.
- For both Snort and ClamAV pattern sets, the cross transitions dominate the transitions on almost every level until level 10. After level 10, the number of cross transitions decreases dramatically and becomes much smaller than the number of forward transitions.
- For Snort pattern set, there is no cross transition beyond level 40. For ClamAV pattern set, the last level containing cross transitions is level 145. Note that the maximum pattern length for Snort and ClamAV pattern sets is 232 and 382, respectively. This indicates that, even if we want to eliminate all cross transitions, it is unnecessary to map a L-level AC-trie onto a L-stage pipeline. The actual pipeline depth needed can be much smaller.
- For both pattern sets, over 99.9% of the cross transitions are on the top 10 levels of the AC-DFA.

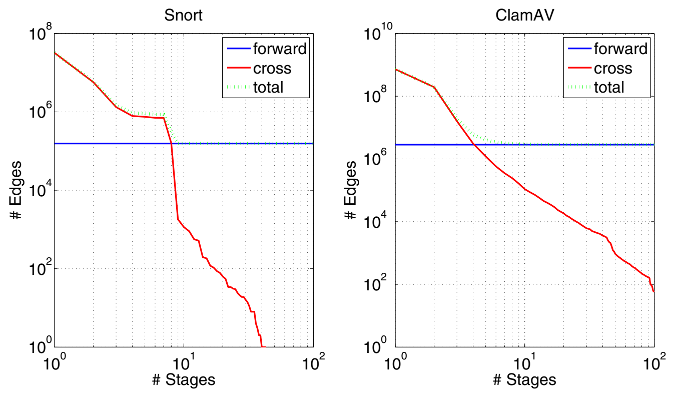

Figure 6. Reducing cross transitions by adding stages

C. Experimental Results

We conducted additional experiments to determine the minimal pipeline depth while removing most of the transitions. We increased the pipeline depth and examined its impact on the reduction of the number of transitions. As expected, Figure 6 shows that, a deeper pipeline results in more reduction in the number of cross transitions. The majority of transitions becomes the forward transitions when using more than 8 and 4 stages, for Snort and ClamAV pattern sets, respectively.

When using a 8-stage pipeline for Snort pattern set, 99.03% of the overall transitions are removed. For ClamAV pattern set, a 4-stage pipelines results in 99.58% reduction in the number of the overall transitions. Hence, a 8-stage and a 4-stage pipelines are sufficient for Snort and ClamAV pattern sets, respectively, to reduce the number of overall transitions by two orders of magnitude.

IV. Supporting Multi-Character Input Per Clock Cycle

Our pipeline architecture is simple and can be used for any AC-DFA. To further boost the throughput, we apply the proposed pipelining scheme to the compressed AC-DFA [5] which can process W ($W \geq 1$) characters per clock cycle. We define W as the input width.

Compressed AC-DFA divides each pattern into W-byte blocks which are used to build the AC-DFA. Figure 7 shows the compressed AC-trie with $W = 2$ for the example shown in Figure 1. To avoid missing matching some patterns, W instances of the compressed AC-DFA should run in parallel. Each of them accepts the same input data stream with a one-character offset. For example, suppose there is an input

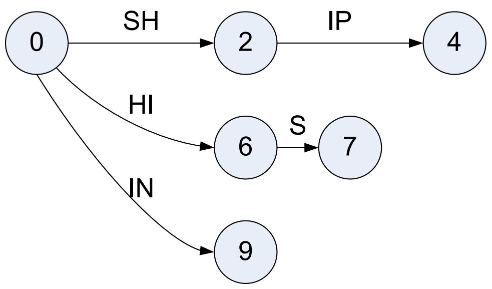

Figure 7. Compressed AC-trie (W = 2) for the pattern set: {SHIP, HIS, IN}

stream "HINT". The first instance of the compressed AC-DFA reads the input stream as "HI" and "NT", while the second instance reads the input stream as "IN" and "T". Although the first instance fails to detect the matching pattern "IN", the second instance will eventually catch the matching pattern.

Combining the compressed AC-DFA with the pipelining scheme can make the pipeline architecture even more scalable. A larger W results in a AC-trie with fewer levels, which can reduce the pipeline depth. We conducted the experiments on Snort and ClamAV pattern sets with various W. Figures 8 and 9 show the number of transitions on each level of the resulting compressed AC-DFA for Snort and ClamAV pattern sets, respectively. As expected, when W is larger, the depth of resulting compressed AC-DFA becomes smaller, and fewer stages are needed to eliminate most of the cross transitions. Our experiments show that, when $W = 8$, a 6-stage pipeline can remove all the cross transitions for the Snort pattern set.

A deep pipeline requires a large amount of I/O pins as well as high bandwidth to access external memory. The above results show that, our pipeline architecture needs only a small number of stages. And there is a tunable trade-off between the pipeline depth and the memory size of the last stage. Thus it is available to use a limited number of external memory in our architecture to support even larger pattern sets.

V. Hardware Architecture

A. Overview

By combining the compressed AC-DFA with the pipelining scheme, we present a multi-pipeline architecture for high throughput multi-pattern string matching. Figure 10 shows

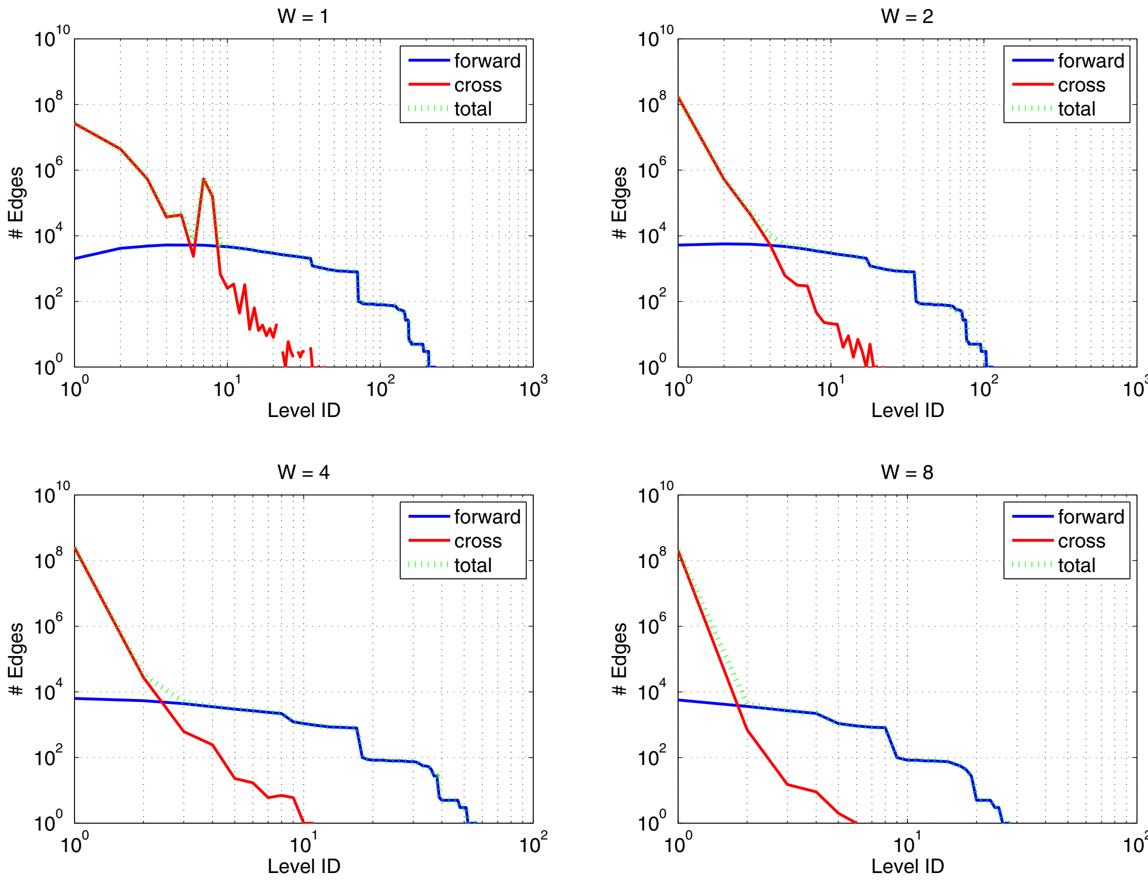

Figure 8. Number of transitions on each level for various input widths ($W = 1, 2, 4, 8$) (Snort)

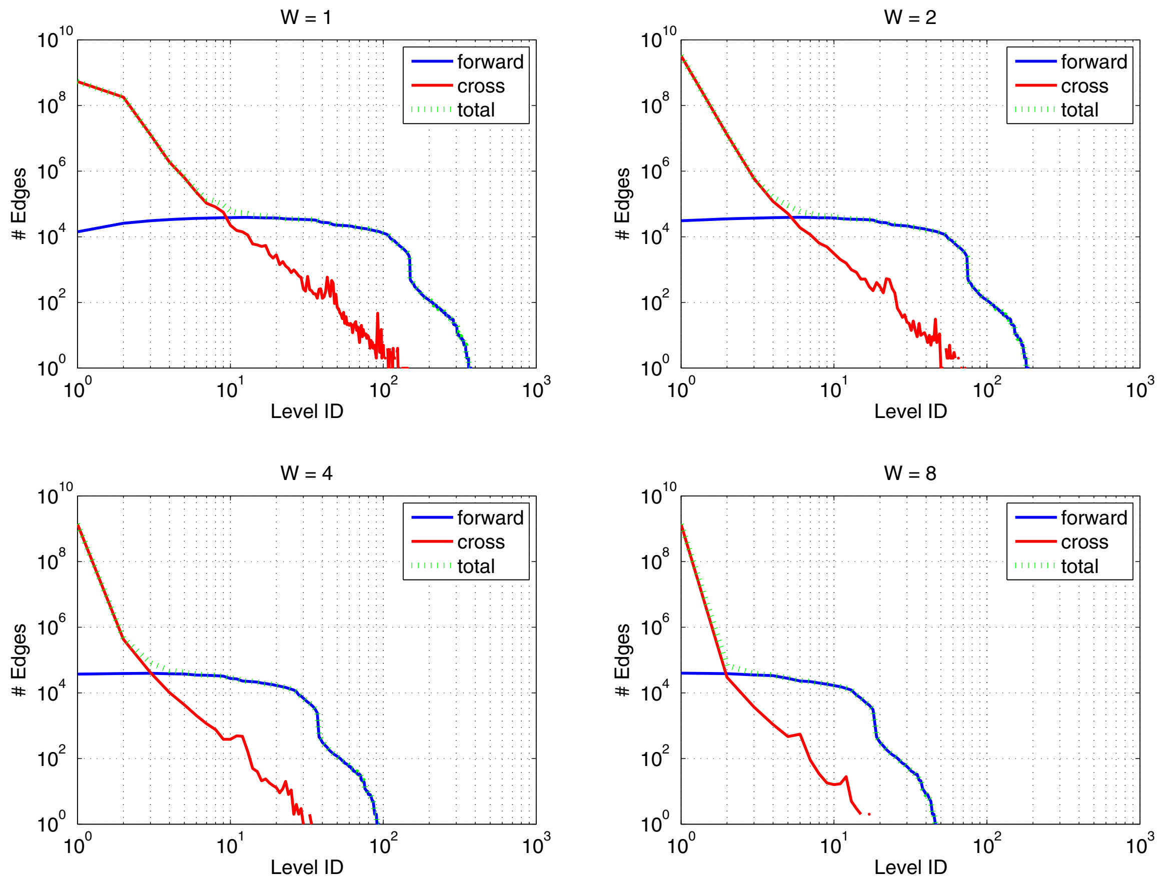

Figure 9. Number of transitions on each level for various input widths ($W = 1, 2, 4, 8$) (ClamAV)

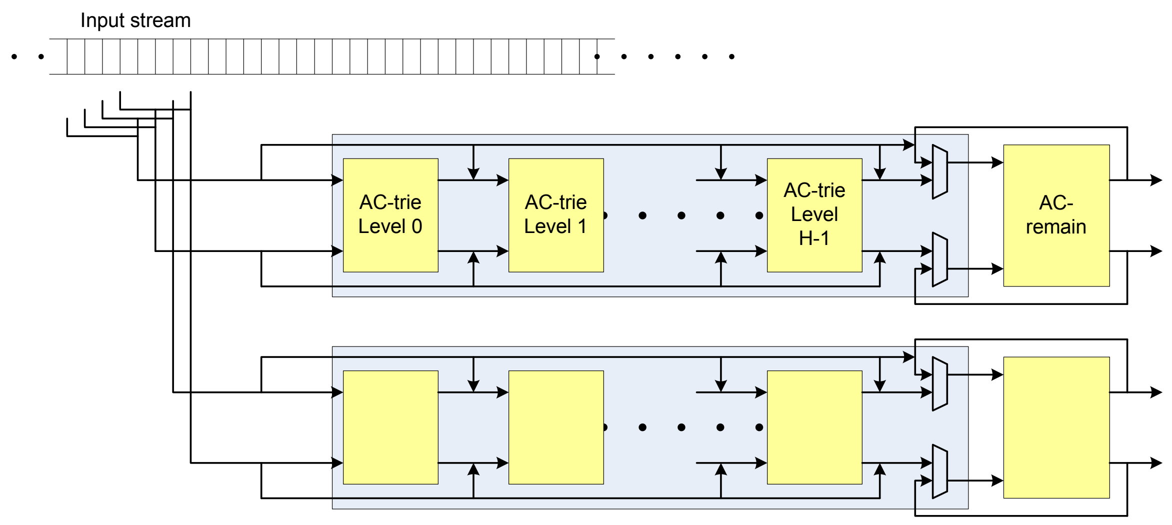

Figure 10. Multi-pipeline architecture for processing multi-character input per clock cycle (P = 2, W = 4)

an example of the architecture. P denotes the number of pipelines and W the input width. Since dual-port memory is used in every stage, two different flows can access the same hardware pipeline in parallel. Hence, the number of hardware pipelines is half that of the pipelined compressed AC-DFA. In other words, $P = W/2$.

As discussed in Section III-A, the AC-DFA is pipelined by mapping its ith level onto the ith stage, $i = 0, 1, \cdots, H-1$. H denotes the pipeline depth i.e. the number of stages. The remaining levels of the AC-DFA, called *AC-remain*, are stored in a single memory, as shown in Figure 10.

During each clock cycle, each pipeline receives two W-character blocks from the input stream. Each W-character block is sent to all the stages including the last stage that stores AC-remain. Each stage receives the W-character block, invokes the state transition by looking up its local state transition table, and forwards the output state to the next neighboring stage. The last stage storing AC-remain will ignore the state output from its previous stage unless its own state becomes *fail*. As discussed in Section III-A, the main function of the previous H stages is to help reuse the last $H - 1$ input characters. As long as the last stage keeps matching the patterns, the information captured in the previous H stages is redundant.

B. State Transition Table

Like many other work (e.g. [3]–[5], [12], [14]), we store the state transition table in each stage as a tuple table to achieve memory efficiency. Figure 11 shows the state transition table corresponding to the compressed AC-trie shown in Figure 7. The state transition table can be implemented in CAM / TCAM [5], as a hashing table [3], [4], [12] or using some specific search structure e.g. BaRT [14]. Our current implementation adopts hashing for its high speed and low cost. The address used to access the state transition table is generated using hardware hashing [3], [4], [12].

Current state	Input label / Label width	Next state	Matching a pattern?
0	SH / 2	2	No
2	IP / 2	4	Yes
0	HI / 2	6	No
6	S / 1	7	Yes
0	IN / 2	9	Yes

Figure 11. State transition table for the example shown in Figure 7

As shown in Figure 11, an input label to be matched can have less than W characters, which occurs only for the last label of a pattern. To match such less-than-W-character labels, we store the label width in terms of the number of characters in the state transition table. When matching the input stream, the label width is used to generate a mask vector. The mask vector is then applied for both the input W-character block and the stored label. The masked values are compared to determine if there is a match.

VI. Performance Evaluation

A. FPGA Implementation Results

We implemented our design for various input widths ($W = 2, 4, 8$) in Verilog, using Xilinx ISE 10.1 development tools. The target device was Xilinx Virtex-5 XC5VFX200T with -2 speed grade. Post place and route results are shown in Table II. The pipeline depth (H) used for each input width ensures that over 99.9% of the cross transitions were removed.

Our architecture exhibited good scalability in terms of throughput and resource usage. When W was increased from 2 to 8, the throughput was improved by $3.5\times$ while the slice usage and the BRAM consumption were increased by just $1.9\times$ and $2.9\times$, respectively. In other words, our solution achieved sub-linear scalability, outperforming the commonly used duplication-based scaling scheme where the increase in resource requirement is linear with the throughput improvement. The overall on-chip logic resource usage was kept low in all of our designs. Even for the largest design that processes $W = 8$ characters per clock cycle, only 10% of the on-chip logic resources were used, which left a large space for incorparating other DPI processing engines such as packet header classifiers [16] and regular expression matching engines [17] on the same chip. We expect larger FPGA devices such as the recent released Xilinx Virtex-6 [24] to support larger input widths (W) to achieve 20+ Gbps throughput.

B. Performance Comparison

Table III compares our design with the state-of-the-art solutions for high performance multi-pattern string matching in DPI systems. To support dynamic updates, only memory-based string matching architectures are considered for ASIC and FPGA based solutions. For a fair comparison, the clock rates of the compared FPGA implementations were scaled to Xilinx Virtex-5 platforms based on the maximum clock frequency. The values in parentheses are the original clock rates reported in those papers. The results of our work is based on the design for $W = 8$. Also note that while most of the results are based on Snort pattern sets, the results of [9], [19] are based on randomly generated binary patterns.

Considering the time-memory and the time-memory-area trade-offs in ASIC / FPGA implementation, we define two new metrics:

$$Efficiency1 = \frac{Throughput}{Bytes/char} \tag{1}$$

$$Efficiency2 = \frac{Throughput}{\#Slices * Bytes/char} \tag{2}$$

Table II
IMPLEMENTATION RESULTS FOR VARIOUS INPUT WIDTHS

Input width	Number of Slices			BRAM usage (Kb)			Clock rate	Throughput
	Used	Available	Utilization	Used	Available	Utilization	(MHz)	(Gbps)
W = 2 (P = 1, H = 16)	1,665	30,720	5%	3,294	16,416	20%	202	3.23
W = 4 (P = 2, H = 8)	2,335		7%	5,472		33%	199	6.38
W = 8 (P = 4, H = 4)	3,168		10%	9,432		57%	178	11.4

Table III
PERFORMANCE COMPARISON

Approaches	Platforms	# of Patterns	Pattern length	# of Slices	Bytes / char	Clock rate (MHz)	Throughput (Gbps)	Efficiency1 (Mbps/Bytes)	Efficiency2 (Kbps/Bytes)
AC-DFA [9]	Cell/B.E.	8,400	≤ 10	NA	NA	3200	2.5	NA	NA
AC-DFA [19]	GeForce 8600GT	4,000	≤ 25	NA	NA	1200	2.3	NA	NA
CDFA [8]	$0.18\mu m$ ASIC	1,785	No limit	NA	3.3	763	6.1	1848	NA
Bit-Split [28]	FPGA	1316	No limit	21,112	23	220 (200)	1.76	77	3.63
B-FSM [14]	FPGA	~8000	No limit	NA	4.05	138 (125)	2.2	543	NA
Field-Merge [4]	FPGA	6944	< 64	12,027	6.33	285	4.56	720	60
Our approach	FPGA	9033	No limit	3,168	6.12	178	**11.4**	1862	588

According to Table III, our architecture outperformed the multi-core processor -based solutions and achieved comparable performance to the ASIC-based solution, with respect to the worst-case throughput. Note that even higher clock rates can be expected when our architecture is implemented in ASIC. As far as we know, our work is the only real implementation of memory-based architectures that sustains over 10 Gbps throughput while supporting more than 9K string patterns from Snort. Our FPGA design achieved more than $2\times$ and $9\times$ improvements over the state-of-the-art FPGA implementations of memory-based string matching engines, with respect to $Efficiency1$ and $Efficiency2$, respectively.

VII. CONCLUSION

This paper gave a detailed discussion on employing pipeline architectures for scalable AC-DFA based multi-pattern string matching in high-speed DPI systems. We revisited the AC algorithm and formally proved that a H-stage linear pipeline can eliminate all the cross transitions to the top H levels of a AC-DFA. Previous pipelined solutions can be viewed as the special case when the pipeline depth is equal to the AC-DFA depth. Simulation using real-life DPI pattern sets showed that a pipeline with no more than 8 stages could remove more than 99% of the overall transitions in a AC-DFA. We further extended the architecture to support multi-character input per clock cycle to achieve multiplicative throughput improvement. The post place and route results of the implementation on a state-of-the-art FPGA showed that our design sustained over 10 Gbps throughput, while consuming a small amount of on-chip resources. Desirable scalability was exhibited: $s\times$ throughput improvement required less than $s\times$ increase in resources. The architecture is also available to support much larger pattern sets by using a limited number of external memory. Our future work includes integrating the proposed string matching architecture with other DPI processing engines such as the packet header classifier [16] and the regular expression matching engine [17].

REFERENCES

[1] Snort: Open source network IDS/IPS, "http://www.snort.org."

[2] Clam AntiVirus, "http://www.clamav.net."

[3] D. Pao, W. Lin, and B. Liu, "Pipelined architecture for multi-string matching," *Computer Architecture Letters*, vol. 7, no. 2, pp. 33–36, Feb. 2008.

[4] Y.-H. E. Yang and V. K. Prasanna, "Memory-efficient pipelined architecture for large-scale string matching," in *FCCM 2009. 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2009.

[5] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network IDS/IPS," in *ICNP '06: Proceedings of the 2006 IEEE International Conference on Network Protocols*. IEEE Computer Society, 2006, pp. 187–196.

[6] P.-C. Lin, Y.-D. Lin, T.-H. Lee, and Y.-C. Lai, "Using string matching for deep packet inspection," *Computer*, vol. 41, no. 4, pp. 23–28, April 2008.

[7] M. Crochemore and T. Lecroq, "Sequential multiple string matching," in *Encyclopedia of Algorithms*, M. Y. Kao, Ed. Berlin: Springer, 2008, pp. 826–829.

[8] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, April 2008, pp. 166–170.

[9] D. P. Scarpazza, O. Villa, and F. Petrini, "High-speed string searching against large dictionaries on the cell/b.e. processor," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–12.

[10] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 207–215, 2004.

[11] O. Villa, D. Chavarria, and K. Maschhoff, "Input-independent, scalable and fast string matching on the Cray XMT," in *Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, April 2009, pp. 1–12.

[12] N. Hua, H. Song, and T. V. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," in *INFOCOM 2009. The 28th Conference on Computer Communications. IEEE*, April 2009.

[13] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, March 2004, pp. 2628–2639 vol.4.

[14] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, April 2006, pp. 1–13.

[15] Z. Baker and V. Prasanna, "A computationally efficient engine for flexible intrusion detection," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 10, pp. 1179–1189, Oct. 2005.

[16] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009, pp. 188–196.

[17] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008, pp. 30–39.

[18] T. AbuHmed, A. Mohaisen, and D. Nyang, "A survey on deep packet inspection for intrusion detection systems," *CoRR*, vol. abs/0803.0037, 2008. [Online]. Available: http://arxiv.org/abs/0803.0037

[19] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Springer-Verlag, 2008, pp. 116–134.

[20] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. IEEE Computer Society, 2005, pp. 112–122.

[21] I. Sourdis, D. N. Pnevmatikatos, and S. Vassiliadis, "Scalable multigigabit pattern matching for packet inspection," *IEEE Trans. VLSI Syst.*, vol. 16, no. 2, pp. 156–166, 2008.

[22] Y. H. Cho and W. H. Mangione-Smith, "Deep network packet filter design for reconfigurable devices," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 1–26, 2008.

[23] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable network packet processing on the field programmable port extender (FPX)," in *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. ACM, 2001, pp. 87–93.

[24] Xilinx Virtex-6 FPGA Family, "www.xilinx.com/products/virtex6/."

[25] P. Moisset, P. Diniz, and J. Park, "Matching and searching analysis for parallel hardware implementation on FPGAs," in *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. ACM, 2001, pp. 125–133.

[26] 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.

[27] C.-H. Lin, Y.-T. Tai, and S.-C. Chang, "Optimization of pattern matching algorithm for memory based architecture," in *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM, 2007, pp. 11–16.

[28] H.-J. Jung, Z. Baker, and V. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006, pp. 8 pp.–.