High Performance Dictionary-Based String Matching for Deep Packet Inspection*

Yi-Hua E. Yang
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: yeyang@usc.edu

Hoang Le
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: hoangle@usc.edu

Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: prasanna@usc.edu

***Abstract*—Dictionary-Based String Matching (DBSM) is used in network Deep Packet Inspection (DPI) applications virus scanning [1] and network intrusion detection [2]. We propose the Pipelined Affix Search with Tail Acceleration (PASTA) architecture for solving DBSM with guaranteed worst-case performance. Our PASTA architecture is composed of a Pipelined Affix Search Relay (PASR) followed by a Tail Acceleration Finite Automaton (TAFA). PASR consists of one or more pipelined Binary Search Tree (*p*BST) modules arranged in a linear array. TAFA is constructed with the Aho-Corasick *goto* and *failure* functions [3] in a compact multi-path and multi-stride tree structure. Both PASR and TAFA achieve good memory efficiency of 1.2 and 2 B/ch (bytes per character) respectively and are pipelined to achieve a high clock rate of 200 MHz on FPGAs. Because PASTA does not depend on the effectiveness of any hash function or the property of the input stream, its performance is guaranteed in the worst case. Our prototype implementation of PASTA on an FPGA with 10 Mb on-chip block RAM achieves 3.2 Gbps matching throughput against a dictionary of over 700K characters. This level of performance surpasses the requirements of next-generation security gateways for deep packet inspection.**

***Index Terms*—Pattern matching; deep packet inspection; network intrusion detection; finite automata; FPGA**

I. Introduction

Popular Deep Packet Inspection (DPI) applications include virus scanning [1] and network intrusion detection [2]. Both applications rely heavily on the operation of dictionary-based string matching (DBSM). In DBSM, input byte-streams from the network interface(s) are matched against a large dictionary of patterns to find all instances of (potentially overlapping) matches. An ideal DBSM solution should have:

1) *High memory efficiency* close to 1 byte of memory usage per dictionary character, allowing the DBSM solution to be implemented with relatively low cost.
2) *High (per-stream) throughput* without the need to aggregate over a large number of slow input streams. DPI with high per-stream throughput is essential for high-speed interfaces (*e.g.*, Gigabit Ethernet).
3) *Guaranteed performance* in the worst case in terms of both memory and sustained throughput performance, making the DBSM robust and less susceptible to performance-based denial-of-service attacks [4].

* This work was supported by U.S. National Science Foundation under grant CCR-0702784.

With a single design methodology, these goals are often very difficult to achieve. We propose a hybrid architecture denoted PASTA. Specifically, our contributions in this paper include the following:

- We analyze the strengths and disadvantages of two popular types of DBSM solutions.
- We propose the hybrid PASTA architecture for solving DBSM. The hybrid design results in efficient realization of the *goto* and *failure* functions of the Aho-Corasick algorithm (AC-alg) over the entire dictionary.
- The PASTA architecture requires on the average only 1.8 bytes per character for practical DPI dictionaries. Our prototype implementation on FPGA achieves 3.2 Gbps throughput over two streams using less than 20K 6-input lookup tables (LUTs).

In Section II, we analyze the existing approaches for DBSM in related work. Section III gives a detailed description of the PASTA architecture its two components: PASR and TAFA. Section IV compares the performance of PASTA with state-of-the-art DBSM solutions. Section V concludes the paper.

II. Related Work

For the purpose of explaining PASTA, we look at two popular types of DBSM solutions: *finite automaton* and *trie-based pipeline*.

A. Finite Automaton

The Aho-Corasick algorithm [3] is commonly used to construct a deterministic finite automaton (AC-DFA) from an arbitrary dictionary. The *goto* and *failure* functions in an AC-DFA define the output transitions of every state for all possible input characters. As a result, a DBSM solution constructed with AC-alg can achieve the lower-bound time complexity of $O(\log|\Sigma|)$ operations per character, Σ being the dictionary alphabet.[1]

On the other hand, the AC-DFA construction increases the number of output transitions per state up to the alphabet size $|\Sigma|$. For a dictionary of N words with λ characters per word, every state transition requires $O(\log(N\lambda))$ bits of memory. The total memory size requirement can thus increase by a

[1] For example, $|\Sigma|$ is 52 for English characters, 128 for ASCII, and 256 for byte (8-bit) values.

factor of $O\left(|\Sigma| \log\left(N\lambda\right)\right)$ over the size of the dictionary. The larger memory size also increases memory access latency.

Several solutions have been proposed to hide this long latency by interleaving a large number of low-bandwidth streams to achieve high aggregated throughput [5], [6], [7]. The memory size can also be reduced by reducing the number of explicitly stored transitions per state to some value much lower than $|\Sigma|$. For example, the state transitions to a few "popular" states can be implicitly resolved at run time by running a small "caching" state machine [8]. The state transitions can also be efficiently packed into a hash table [9]. However, doing so can require several sequential comparisons per transition and/or the use of hash functions. Both can lengthen the processing time per character and make the DBSM non-deterministic.

B. Trie-based Pipeline

It is observed [8] that a large portion (>90%) of state transitions in AC-DFA are "backward" transitions towards the first few levels of states near the root of the dictionary tree. One way to remove all such transitions from consideration is to pipeline the dictionary tree along the tree depth. This results in a DBSM solution with a pipelined trie structure [10], [11].

Essentially, a trie-based pipeline *relays* the matching of the input from the root along the depth of the dictionary trie, until either a trie leaf or an invalid state is reached. Since every input character initiates a new relay process, no dictionary pattern in the input can be missed. The simplicity of this approach comes at the cost of higher memory bandwidth proportional to the number of stages in the pipeline. Conventionally, there are in generals two methods to solve this bandwidth problem:

1) Compare $S > 1$ characters per stage, reducing total number of stages to $\lceil \lambda/S \rceil$ [12].
2) Use an S-stage pipeline in a feedback loop to construct a finite automaton [10].

Method 1 above must consider backward transitions from depth h to depth l if $h-l < S$, while Method 2 must consider those if $h-l$ is a multiple of S. Both can increase the (space and time) complexity of the DBSM solution considerably. It is also much more difficult to efficiently store and access the S-character state transitions in memory, a problem often dealt with by the use of dictionary-specific hash functions.

III. The PASTA Architecture

Figure 1 shows the overall PASTA architecture. Stream(s) of input characters first go through the *Pipelined Affix Search Relay* (PASR), then to the input buffer to be accessed by the *Tail Acceleration Finite Automaton* (TAFA).

Claim 1: For DBSM solutions used in deep packet inspection, the AC-DFA has the following properties:

- Most backward transitions (>98%) have low (<8) "target depth," *i.e.*, their target states are located at the first 7 levels of the dictionary tree.
- Many backward transitions have high (>16) "source depth" up to the end of the words. A state at a high level of the dictionary tree has the same likelihood of backward transition as a state at a low level.

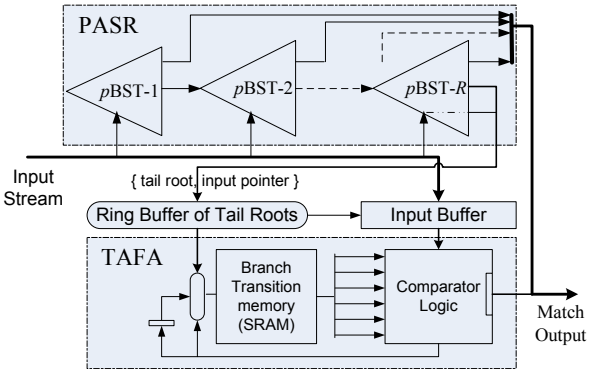


Fig. 1. Overall architecture of PASTA.

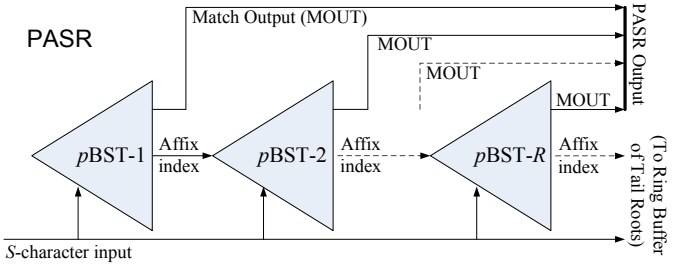


Fig. 2. Relay of affix segments of PASR.

- A large percentage (20%–80%) of dictionary words have long (>30 characters) patterns.

As discussed in Section II-B, the simplest way to eliminate backward transitions lengths less than l is to pipeline the string matching to at least level l of the dictionary tree. This is performed by PASR. On the other hand, TAFA condenses potentially long "tails" of dictionary words into a compact finite automaton.

A. Pipelined Affix Search Relay

The Pipelined Affix Search Relay (PASR, Figure 2) is similar to a trie-based pipeline (see Section II-B) with *R* stages and *S* characters per stage, with the following notable differences:

1) PASR matches the input against dictionary $\mathbb{D}$ up to only depth $R \times S$ in a *relay of affix segments*.
2) Each stage is organized as a *pipelined Binary Search Tree* rather than individual trie nodes.
3) Every state at depth $R \times S$ of $\mathbb{D}$ reached by the input stream is recorded in a *ring buffer of tail roots*.

1) Relay of Affix Segments: To construct the relay structure of PASR, we divide the "initial" part of the dictionary $\mathbb{D}$ (up to depth $R \times S$) into *R* affix segments of length *S*, each segment handled by one stage of the relay.

Definition 1: The *r*-th affix segment of length *S*, $1 \leq r \leq R$, of a dictionary $\mathbb{D}$ consists of the set of affixes $\mathbb{D}_{R,S}[r] \triangleq \{\forall X \in \mathbb{D} : X[(r-1)S, rS]\}$, where $X[a,b]$ is the affix of $X \in \mathbb{D}$ from the *a*-th character up to (not including) the *b*-th character.

TABLE I
THE 2 AFFIX SEGMENTS OF LENGTH 3 OF $\mathbb{D}$ =
{**AN**, **ANGRY**, **CA**, **COMM**, **COMMA**, **COMMAND**, **COMMON**, **DOG**, **DOGMA**}

$x \in \mathbb{D}[1]$			
$v(x)$	$b(x)$	p	d
0 **ANG**	010	1	1
0 **CA**⊙	010	0	2
0 **COM**	000	1	3
0 **DOG**	001	1	4

$x \in \mathbb{D}[2]$			
$v(x)$	$b(x)$	p	d
1 **RY**⊙	010	0	1
3 **MAN**	110	1	2
3 **MON**	101	1	3
4 **MA**⊙	010	0	4

We use p and d to represent $p(x)$ and $d(x)$, respectively.

Four issues must be addressed in order to match $\mathbb{D}$ (up to depth $R \times S$) by the *R* stages of length *S* relay structure:

1) A stage must compare the input efficiently with two neighboring affixes *x* and *y* even if *y* is a prefix of *x*.
2) A stage must handle affixes of length less than *S*.
3) The r-th stage, $r > 1$, must "remember" the matching progress from the earlier ($< r$) stages.
4) The matching must not miss patterns that do not begin at the S-character boundaries in the input stream.

The first issue is solved by maintaining an S-bit *containment bitmap*, $b(x)$, with every affix *x*. The value of $b(x)$ is all 0's by default. If x is a suffix of some $X \in \mathbb{D}$, then the $\lambda(x)$-th bit of $b(x)$ is set to 1, where λ denotes the "length" function. Furthermore, if another affix y in the same affix segment is a prefix of *x*, then *y* is "merged" into *x* by setting the $\lambda(y)$-th bit of $b(x)$ to 1. Suppose $S = 3$, $R = 2$ and $\mathbb{D}$ ={ COMM, COMMA, COMMON }. The affix "**M**" will be merged into both "**MA**" and "**MON**", setting b("MA") and b("MON") to $\langle \mathbf{1}10 \rangle$ and $\langle \mathbf{1}01 \rangle$, respectively.

To solve the second issue, all affixes are padded by 0's, if necessary, to S characters. This allows all affixes to be treated as fixed-length S characters. A *padding bit* is assigned to each affix x to indicate whether the affix was padded. To solve the third issue, we assign to each affix an *affix value* and an *affix index* to keep track of the matching progress. These additional meta data are defined formally below.[2]

Definition 2: Suppose dictionary $\mathbb{D}$ up to depth $R \times S$ is segmented into *R* affix segments of length *S*. Let $\odot$ denote the 0-value character, and x^S the string x padded by $\odot$, if necessary, to *S* characters. $\forall\, x \in \mathbb{D}[r]$, $1 \le r \le R$:

- The *padding bit* $p(x)$ is 1 if $x = x^S$ and 0 otherwise.
- The *affix value* $v(x)$ and *affix index* $d(x)$ are defined inductively from $\mathbb{D}[1]$ up to $\mathbb{D}[R]$ as follows:
  1) $\forall\, x \in \mathbb{D}[1]$, $v(x) \triangleq x^S$.
  2) $\forall\, x \in \mathbb{D}[r]$, $d(x)$ has $\log(|\mathbb{D}[r]|) + 1$ bits, and is the 1-based index of x in $\mathbb{D}[r]$ sorted by $v(x)$.
  3) $\forall\, x \in \mathbb{D}[r]$, $1 < r \le R$ and w precedes x in some $X \in \mathbb{D}$, $v(x) \triangleq d(w) \circ x^S$ where $\circ$ represents bitvector concatenation.

Table I shows an example of 2 affix segments of length 3, addressing issues 1–3 above.

The fourth issue can be solved by initiating an S-character matching at every character offset in the input. For example, suppose the input stream consists of characters $[c_0, c_1, \ldots]$. Each PASR stage will first search $[c_0, \ldots, c_{S-1}]$ within its affix segment, then $[c_1, \ldots, c_S]$, $[c_2, \ldots, c_{S+1}]$, and so on. By doing so, no dictionary word starting at any input character can be missed, while a per-stream throughput of one character per cycle can be achieved.

2) Pipelined Binary Search Tree: Suppose dictionary $\mathbb{D}$ has R affix segments of length S. Each r-th segment will be handled by a pipelined *binary search tree*, *p*BST-*r*, $1 \le r \le R$, as shown in Figure 2. Each *p*BST-*r* node represents an affix in $\mathbb{D}[r]$ with the node value equal to the affix value.

A *p*BST is fully pipelined along its tree depth. The root node at level 0 always has address 0. Any node at a level above 0 is addressed implicitly using the address of the node's parent. Suppose there are N nodes in the *p*BST and a node x is stored at address $a_i(x)$ of level i, $0 \le i \le \lfloor \log N \rfloor$. Then the two children of x, $x_l < x$ and $x_r > x$, will be stored at addresses $a_{i+1}(x_l) = 2a_i(x)$ and $a_{i+1}(x_r) = 2a_i(x) + 1$, respectively, of level $i + 1$. The affix search described above can take $\lfloor \log N \rfloor + 1$ clock cycles. Due to the use of affix index (see Definition 2), all affix searches in the same *p*BST are independent to one another, and the throughput of one character per cycle can be achieved.

Each *p*BST-*r* can output three possible results: *non-match*, *partial match*, and *full match*. A non-match is discarded and a "bubble" is sent to *p*BST-(*r*+1). A partial match relays the affix index *d*(*x*) to *p*BST-(*r*+1) to match with the next input segment. A full match can result in both a dictionary match and/or an affix index relayed to the next *p*BST.

3) Ring Buffer of Tail Roots: In every clock cycle PASR can send (at most) one record consisting of an affix index and an input pointer to the *ring buffer of tail roots*. Each record of { affix index , input pointer } indicates that some state at the end of PASR and the beginning of TAFA is reached. These states are called *tail roots* and are identified by the affix index output from the last PASR stage. The input pointer associated with each tail root specifies the input stream position up to which the tail root is reached.

Any tail root whose value (affix index) is 0 can be discarded. Furthermore, suppose a record $\{s_r, p_r\}$ arrives at the ring buffer with tail root s_r and input pointer p_r. If TAFA has processed the input character(s) at or after p_r, then s_r can also be discarded (see Lemma 1 in Section III-B).

B. Tail Acceleration Finite Automaton

The *Tail Acceleration Finite Automaton* (TAFA) is an amortized deterministic finite automaton constructed with the Aho-Corasick *goto* and *failure* functions [3], where all the (backward) transitions towards the PASR part of the AC-DFA are removed. Figure 3 shows an example where $R = 2$, $S = 2$, and $\mathbb{D}$ = {INTERACT, ACCOUNT, ACT, CIDER, COUNTER, COUNTING, COINCIDE}.

Algorithm 1 describes the operations of TAFA. The finite automaton described in Algorithm 1 is not deterministic, since step 6b can jump back to the beginning of step 6 by taking a *failure* transition without consuming any input character.

[2]To avoid excessive notations, we assume all strings and numbers are represented as bitvectors, which can concatenate with one another.

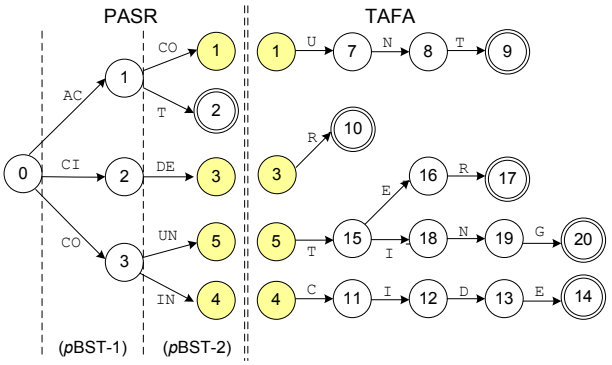


Fig. 3. Example of mapping PASR output states to TAFA tail roots with R=2 and S=3.

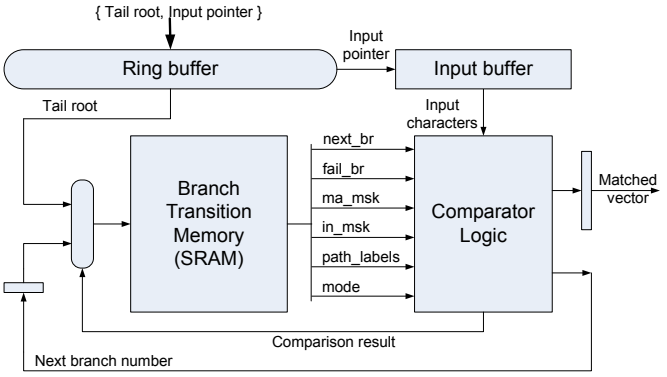


Fig. 4. Architecture of TAFA.

Algorithm 1 Operations of TAFA for a dictionary at depth $L = R \times S$ and above.

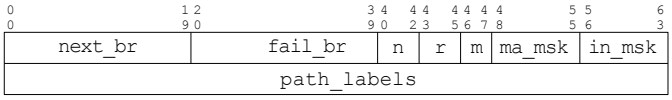
1) Initialize current state $s \leftarrow 0$, input pointer $p \leftarrow 0$.
2) Read and remove a record $\{s_r, p_r\}$ from the ring buffer of tail roots (Section III-A3).
3) If $p_r \leq p$, jump back to step 2.
4) Set $s \leftarrow s_r$, $p \leftarrow p_r$.
5) Read character c at input position p. p++.
6) Check c in state s:
   a) If $goto\,(s, c) = s'$ and $s' \neq 0$, $s \leftarrow s'$. Jump to step 7.
   b) If $\text{depth}\,(failure\,(s)) \geq L$, $s \leftarrow failure\,(s)$. Jump back to the beginning of step 6.
   c) Jump back to step 2.
7) Check the matching status at s and output $\{s, p\}$ if s represents a word match.
8) Jump back to step 5.

However, since the number of *failure* transitions taken is limited by the number of input characters consumed [13], TAFA will always have an *amortized* throughput of at least 0.5 characters per clock cycle. In practice, real-world dictionaries for deep packet inspection usually result in an amortized throughput of at least 0.8–0.96 characters per cycle.

Lemma 1: TAFA can ignore any { tail root, input pointer } = $\{s_r, p_r\}$ reported by PASR if the input character at p_r has been processed in a valid state in TAFA.

Proof: Recall from Section III-A2 that an $\{s_r, p_r\}$ reported by PASR indicates that a state associated with s_r can be reached by the input stream up to p_r. On the other hand, a valid state s in TAFA processing the input character at p_r also means that the state s can be reached by the input stream up to p_r. Being a tail root, s_r must have a lower depth than s; the Aho-Corasick algorithm (AC-alg) used to construct TAFA ensures that s_r can be ignored when a valid state s is reached. ■

Figure 4 shows the circuit architecture of TAFA. The "suffix tries" of TAFA is stored in blocks of 128 bits. Each 128-bit block is called a *branch* and contains from 1 to 8 trie paths of various lengths:



`next_br` Branch number of the first child branch
`fail_br` Branch number of the failure branch
`count` (n) Number of valid children branches (0 to 7)
`roll` (r) Number of characters to roll back when failure branch is taken (0 to 7)
`mode` (m) Operation mode of the branch (0 to 3)
`ma_msk` Mask for reporting word matches (one bit per byte in `path_labels`)
`in_msk` Mask for filtering input characters (one bit per byte in the input vector)
`path_labels` Path pattern(s) stored in the branch

Depending on the value of m (`mode`), TAFA compares 2^m input characters (bytes) with the `path_labels` in the way defined by Algorithm 1 with multiple and branch-specific strides. If a TAFA state has more than 8 *goto* targets, it is broken into multiple branches of $m = 0$ connected in a sequence of `fail_br`. While this can reduce the minimum throughput of TAFA to below 0.5 input characters per cycle, for dictionaries used in both ClamAV and Snort, this situation does not occur above depths 5 and 18, respectively.

IV. Performance Evaluation

A. Memory Efficiency

Due to the highly compact data structure of *p*BST, PASR achieves a memory usage ~1.2 bytes per dictionary character. A small (typically 10%–15%) overhead is needed to store the containment bitmap and the padding bit described in Definition 2. Memory usage of TAFA is usually ~2 bytes per character. An 8-byte branch may not be fully populated at the end of a word. If most of the word "tails" are much longer than 8 characters (*e.g.* both Snort and ClamAV), then the percentage of branches that are not fully populated will be low.

B. Implementation and Comparison

We implemented the proposed architecture in Verilog, using Synplify Pro 9.6.2 and Xilinx ISE 10.1.3, with Virtex-5

TABLE II
PERFORMANCE COMPARISON OF PASTA WITH OTHER STATE-OF-THE-ART DBSM SOLUTIONS.

	AG-Tput (Gbps)	Capacity (Kch)	(Gbps × Kch)	PS-Tput (Gbps)	Updatability	Platform
PASTA [1]	**3.2**	**≥700**	**2240**	**1.6**	**Flush BRAM**	**Virtex 5 LX**
Pattern-parallel[14] [2]	8.8	100 [6]	880	8.8	Re-synthesis	Virtex II Pro
Bit-split[15] [2]	3.2	28	90	1.6	Rebuild DFA	Virtex 4 FX
Field-merge[11] [1]	4.56	200	920	1.14	$O(\delta)$ updates	Virtex 5 LX
DFA-FPGA[7] [3]	~ 3.2	~ 180 [7]	~ 576	1.6	Rebuild DFA	Virtex 5 LX
DFA-MC[5] [3]	3.5~4.5	~ 150 [7]	~ 680	< 0.02	Rebuild DFA	2P Cell/B.E.
DFA-Cray[16]	7.5	400	3000	< 0.02	Rebuild DFA	32P Cray XMT
DFA-GPU[6]	2.3	100 [7]	230	0.072	Rebuild DFA	nVidia G80
DFA-hash[8] [4,5]	8.5	200 [6]	1700	4.25	Rebuild DFA	0.18 μm ASIC [5]

Note: **1.** Use FPGAs with 10 Mb BRAM. **2.** Earlier reported performance scaled up linearly to 10 Mb BRAM or 200K LUTs. **3.** Requires input training to identify the "hot states." **4.** Assumes 10 Mb 2-way set assoc. SRAM at 1.0625 GHz. **5.** Performance estimated from synthesis only. **6.** Capacity highly depends on effective dictionary compaction by the proposed data structures. **7.** A much larger capacity is possible with some impact on throughput.

XC5VLX330 as the target. Our PASTA implementation is configured as dual pipelines (processing two individual input streams in parallel) to take advantage of the dual-ported on-chip block RAM (BRAM).

The PASR consists of 4 *p*BSTs. Each *p*BST module is internally pipelined along the tree level, with the amount of memory in each level twice of the previous level. Each node (affix) in a *p*BST is 89 bits wide, including a 80-bit affix value, an 8-bit containment bitmap and a single padding bit. The affix value consists of a 16-bit affix index from the previous *p*BST stage plus the 64-bit (8-byte) pattern of the current affix. [3] A single *p*BST with 15 pipeline stages requires 48 blocks of 36-Kb BRAMs (16%) and less than 4K LUTs (5%).

The tail roots and input buffers each occupies a single 36-Kb BRAM and is assisted by a small amount of logic (< 1K LUTs) to manage the ring accesses. The TAFA accesses to 4096 KB branch transition memory, where each branch occupies a block of 16 bytes. All branches are addressed directly without the use of any hash function. The post place-and-route clock rate is over 200 MHz, resulting in an aggregated throughput of 3.2 Gbps over two streams.

Table II compares PASTA with other state-of-the-art DBSM solutions using the following metrics:

- *Aggregated Throughput*. Total number of bits across all input streams matched per second. (AG-Tput)
- *Per-stream Throughput*. Number of bits per input stream matched per second. (PS-Tput)
- *Overall Performance*. Product of AG-Tput and Capacity, which often trade off with each other. (Gbps × Kch)
- *Capacity*. Total number of characters in the dictionary.
- *Updatability*. Complexity to update the dictionary.

As shown in Table II, PASTA out-performs other state-of-the-art DBSM solutions on FPGAs by at least 50%. In addition, the performance of PASTA is neither input- nor dictionary-dependent.

V. CONCLUSION AND FUTURE WORK

Our hybrid PASTA architecture offers guaranteed high performance under the worst-case input for DBSM in both Snort and ClamAV. This guaranteed performance can greatly improve the robustness of the DPI system. Our approach can be further tailored for low-power operations on both FPGAs and ASICs; or it can be optimized to run on the emerging many-core architectures; or it can be designed to accept incremental updates of the dictionary.

[3]The 16-bit affix index allows PASR to handle more than 64K words. This length can be easily extended when more on-chip memory is available.

REFERENCES

[1] "Clam AntiVirus," http://www.clamav.net/.
[2] "SNORT," http://www.snort.org/.
[3] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.
[4] 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.
[5] D. P. Scarpazza, O. Villa, and F. Petrini, "High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor," in *IEEE Intl. Parallel & Distributed Proc. Sym.*, 2008.
[6] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Recent Advances in Intrusion Detection*, vol. 5230, 2008, pp. 116–134.
[7] Q. Wang and V. K. Prasanna, "Pipelined Multi-Core Architecture on FPGA for Large Dictionary String Matching," in *IEEE Sym. on Field Programmable Custom Computing Machines*, 2009.
[8] T. Song, W. Zhang, D. Wang, and Y. Xue, "A Memory Efficient Multiple Pattern Matching Architecture for Network Security," in *IEEE INFOCOM*, 2008.
[9] J. van Lunteren, "High-performance Pattern-matching for Intrusion Detection," in *IEEE INFOCOM*, 2006.
[10] D. Pao, W. Lin, and B. Liu, "Pipelined Architecture for Multi-String Matching," in *IEEE Computer Architecture Letters*, vol. 7, 2008.
[11] Y.-H. E. Yang and V. K. Prasanna, "Memory-Efficient Pipelined Architecture for Large-Scale String Matching," in *IEEE Sym. on Field Programmable Custom Computing Machines*, 2009.
[12] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High Speed Pattern Matching for Network IDS/IPS," in *IEEE Intl. Conf. on Network Protocols, ICNP*, 2006.
[13] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," in *IEEE Sym. on Field Programmable Custom Computing Machines*. ACM, 2004, pp. 223–232.
[14] ——, "A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs," in *IEEE Sym. on Field Programmable Custom Computing Machines*, April 2004.
[15] H.-J. Jung, Z. Baker, and V. Prasanna, "Performance of FPGA Implementation of Bit-Split Architecture for Intrusion Detection Systems," in *IEEE Intl. Parallel & Distributed Proc. Sym.*, April 2006.
[16] O. Villa, D. Chavarria, and K. Maschhoff, "Input-independent, Scalable and Fast String Matching on the Cray XMT," in *IEEE Intl. Parallel & Distributed Proc. Sym.*, 2009.