A Memory-Efficient and Modular Approach for String Matching on FPGAs

Hoang Le and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
{hoangle, prasanna}@usc.edu

***Abstract*—In Network Intrusion Detection Systems (NIDSs), string matching demands exceptionally high performance to match the content of network traffic against a predefined database of malicious patterns. Much work has been done in this field; however, they result in low memory efficiency[1]. Due to the available on-chip memory and the number of I/O pins of Field Programmable Gate Arrays (FPGAs), state-of-the-art designs cannot support large dictionaries without using *high-latency* external DRAM. We propose a novel Memory efficient Architecture for large-scale String Matching (MASM), based on pipelined binary search tree. With memory efficiency close to 1 byte/char, MASM can support a dictionary[2] of over 4 MBytes, using a single FPGA device. The architecture can also be easily partitioned, so as to use external SRAM to handle even larger dictionaries of over 8 MBytes. Our implementation results show a sustained throughput of 3.5 Gbps, even when external SRAM is used. The MASM module can be simply duplicated to accept multiple characters per cycle, leading to scalable throughput with respect to the number of characters processed in each cycle. Dictionary update involves only rewriting the memory content, which can be done quickly without reconfiguring the chip.**

I. Introduction

State-of-the-art FPGAs offer high operating frequency, unprecedented logic density and a host of other features. Additionally, FPGAs are programmed specifically for the problem to be solved rather than to solve all problems, they can achieve higher performance than the general-purpose processors. The advantage is more beneficial for applications with a regular structure and abundant parallelism. Therefore, FPGA is a promising implementation technology for many areas. Applications using FPGAs range from network flow analysis [7], Hybrid FPGA-CPU Computational Components [8], to FPGA supercomputing [12], among others.

With the rapid expansion of the Internet and the explosion in the number of attacks, design of high speed applications, especially Network Intrusion Detection Systems (NIDS), has been a big challenge. Advances in optical networking technology are pushing link rates beyond OC-768 (40 Gbps). This throughput is impossible to achieve using existing software-based solutions [9], and thus, must be performed in hardware. Most hardware-based solutions for high speed string matching in NIDS fall into two main categories: ternary content addressable memory (TCAM)-based and dynamic/static random access memory (DRAM/SRAM)-based. Although TCAM-based engines can retrieve result of a match in just one clock cycle, they are power-hungry and their throughput is limited by the relatively low speed of TCAMs. On the other hand, SRAM-based solutions require multiple cycles to perform a search. Therefore, pipelining techniques are commonly used to improve the throughput. The SRAM-based approaches, which are memory-intensive, often result in an inefficient memory utilization. This inefficiency limits the size of the supported dictionary. In addition, it is not feasible to use external SRAM in these architectures due to the limited number of I/O pins. This constraint restricts the number of external stages, while the amount of on-chip memory constrains the size of the memory for each pipeline stage. Due to these two limitations, state-of-the-art SRAM-based solutions do not scale well to support larger dictionaries. This scalability has been a dominant issue for implementations on FPGAs.

We propose and implement a scalable, high-throughput, **M**emory efficient **A**rchitecture for large-scale **S**tring **M**atching (MASM). This architecture utilizes binary search tree (BST) structure to improve the storage efficiency. In a complete binary tree structure, the amount of memory of the next level doubles that of the current one. Therefore, the last levels (further from the root) can be moved onto external SRAMs, as the amount of available SRAMs is much larger than that of the on-chip memory. MASM also provides a fixed-latency due to the linear pipelined architecture. Furthermore, the number of pipeline stages is determined by the size of the supported dictionary, as discussed in detail in Section IV-D.

This paper makes the following **contributions**:

1) MASM achieves a memory efficiency close to 1 B/char (Section III-D). State-of-the-art designs can only achieve the memory efficiency of over 2 B/char.
2) It scales very well as the size of the dictionary

[1]The memory efficiency (in bytes/char) is defined as the ratio of the amount of the required storage memory (in bytes), and the size of the dictionary (number of characters).

[2]The size of a dictionary is the total number of characters in all the patterns in the dictionary.

grows, due to linear storage complexity and resource requirements (Section IV-D).
3) The architecture can easily utilize external SRAM to handle larger dictionaries (Section V-B).
4) The implementation on FPGA shows a sustained aggregated throughput of 3.5 Gbps (Section VI).
5) The design can be duplicated to improve throughput by exploiting its simple architecture (Section VI).

The rest of the paper is organized as follows. Section II covers the background and related work. Section III introduces the proposed string matching algorithm. Section IV and V describe the architecture and its implementation. Section VI presents implementation results. Section VII concludes the paper.

II. Background And Related Work

A. Definitions and Notations

The following definitions and notations are used throughout this paper:

1) $|S|$ is the length (in *bits*) of string S (e.g. $|abcd| = 32$).
2) S_1 is a prefix of S_2 *iff* $S_2 = S_1.*$ (e.g. $S_1 = ab$, $S_2 = abcd$).
3) $S_1 < S_2$ *iff* the first non-matching character (from the left) in S_1 has a smaller ASCII value than the corresponding character in S_2.
4) The binary representation of string S, denoted as B_S, is obtained by converting all characters of S into their corresponding ASCII values.
5) $*$ $(star)$: indicates any string including the empty one.
6) $\{n\}$: repeat the previous item exactly n times, where n is a positive integer.
7) The *range* representation of string S, using L bits ($L > |S|$), is denoted as $R_S = [R_{S_L}, R_{S_H}]$, where $R_{S_L} = B_S0\{n\}$ and $R_{S_H} = B_S1\{n\}$ such that $n = L - |S|$. In the case of $L = |S|$, $R_S = [B_S, B_S]$.
8) S_1 and S_2 are said to be non-overlapping *iff* their corresponding ranges (using L bits), R_{S_1} and R_{S_2}, are disjoint. Note that $L \geq \max(|S_1|, |S_2|)$.

B. String (Pattern) Matching

String matching is one of the most important functions of the NIDSs, as it provides the content-search capability. A string matching algorithm compares all the string patterns in a given database to the traffic passing through the device. Note that the string matching is also referred to as exact string matching.

The string matching problem can be formulated as follows. Given an input string of length K, an alphabet Σ consisting of all ASCII characters ($|\Sigma| = 256$), and a dictionary consisting of N string patterns $S_1, S_2, ..., S_N$, whose characters belong to the alphabet, our goal is to find and return all occurrences, if any, of every pattern in the given input string.

C. Related Work

In general, architectures for string matching on FPGA can be classified into the following categories: pipelined non-deterministic finite automaton (NFA) based approach and pipelined deterministic finite automaton (DFA) based approach. Each has its own advantages and disadvantages.

In TCAM-based architecture presented in [17], the entire dictionary is stored onto TCAM, which has deterministic lookup time for any input. To reduce wastage, TCAM is chosen such that its width is shorter than the length of the longest pattern. Patterns, whose length are greater than the width of TCAM, are divided into shorter patterns (sub-patterns). The first sub-pattern is called prefix-pattern, and the subsequent ones are called suffix-patterns. The short suffix-patterns are padded with "do not care" states up to the TCAM's width. Dictionary updates can be done simply by adding or removing patterns from TCAM, whenever needed. Hence, the advantages of this approach are high memory efficiency and simple updating mechanism. The achieved throughput of 4 Gbps is also reasonably high (scaled to current technology).

In a pipelined NFA string matching architecture [5], [6], [13], all of the given string patterns are combined to build an NFA on FPGA. By accepting multiple characters per cycle, these designs achieved a throughput of $8 - 10$ Gbps. However, the main drawbacks of this approach are the inflexibility and relatively small size of the supported dictionary. This approach also has a potentially high resource utilization. Since these designs are bounded by the FPGA's logic resource rather than memory, the memory efficiency was not reported.

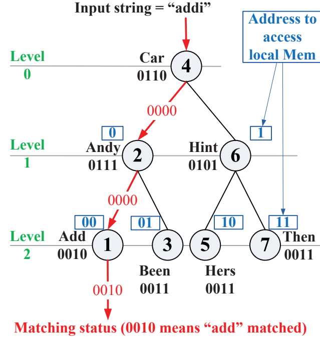
Bit-split DFA architectures [11], [14] are based on the Aho-Corasick (AC-DFA) algorithm [4]. The original algorithm converts a dictionary tree of N patterns to a DFA with $O(N)$ states. The AC-DFA requires $O(1)$ computations per character, regardless of the size of the dictionary. However, the valid (non-zero) states in the state transition table (STT) are usually very sparsely populated. This problem leads to a large memory wastage [15], and makes memory bandwidth and latency the bottlenecks for string matching based on AC-DFA. The memory efficiency was reported as 23 and 33 byte/char in [11] and [14], respectively.

The field-merge NFA architecture [16] is similar to the bit-split NFA discussed above. The algorithm partitions each input character into k bit-fields, and constructs a tree-structured NFA, called the full state machine (FSM). The field-merge NFA is a fully memory-based approach, where dynamic dictionary updates can be performed easily by updating the memory content. Yet, the memory efficiency of this architecture is dictionary dependent, with a large variance among different dictionaries (2.16 byte/char for the Rogets and 6.33 byte/char for the Snort dictionary).

The variable-stride (VS) multi-pattern matching architec-

Pattern	Converted Pattern	Bitmap	Node
An And Andy	Andy	0111	2
Add	Add	0010	1
Bee Been	Been	0011	3
The Then	Then	0011	7
Her Hers	Hers	0011	5
Hi Hint	Hint	0101	6
Car Ca	Car	0110	4

(a) Sample dictionary



(b) Sample BST

Figure 1. A sample dictionary and its corresponding BST

ture [10] is also a DFA-based approach. Even though this is an ASIC implementation, the work is presented as it achieved high memory efficiency and can be ported to FPGA platform. The main idea is that a variable number of characters can be scanned in one step. In the pre-processing step, all patterns are first segmented into a sequence of blocks. While this design achieved a good memory efficiency of 2.3 byte/char, the throughput is difficult to estimate, as it is highly dictionary dependent.

III. String Matching Algorithm

A. Prefix Properties

The string matching can be formalized as a prefix matching problem, with the exception that *all* matches are returned, not only the *longest* one. Consider a L-bit number space Ω and 2 strings S_A, S_B such that $L \geq max(|S_A|, |S_B|)$, we denote $\Omega = [0\{L\}, 1\{L\}]$. Let $n_A = L - |S_A|$, $n_B = L - |S_B|$, and $n_{BA} = abs(|S_B| - |S_A|) = abs(n_B - n_A)$. The following properties are used as the foundation of our work.

Property 1: Given two distinct strings, S_A and S_B, if $|S_A| = |S_B|$ then S_A and S_B do not overlap.

Proof: Each string represents a range of numbers in Ω. Without loss of generality, assume that $S_A < S_B$. Let $n = n_A = n_B$. The range of S_A and S_B are $[S_A 0\{n\}, S_A 1\{n\}]$ and $[S_B 0\{n\}, S_B 1\{n\}]$, respectively. Since $S_A < S_B$, we have $S_A 1\{n\} < S_B 0\{n\}$; hence, the two ranges do not overlap. ■

Property 2: Given two distinct strings, S_A and S_B, if S_A is a prefix of S_B then $S_A < S_B$.

Proof: If S_A is a prefix of S_B then $S_B = S_A X\{n_{BA}\}$, where $X \subseteq \{0, 1\}$. Thus, $S_B 0\{n_B\} = S_A X\{n_{BA}\} 0\{n_B\}$, but $S_A X\{n_{BA}\} 0\{n_B\} > S_A 0\{n_{BA}\} 0\{n_B\} \Rightarrow S_B 0\{n_B\} > S_A 0\{n_A\}$. Therefore, $S_A < S_B$. ■

B. Fixed-Length String Matching

We propose a memory efficient data structure based on a binary search tree. BST is a special binary tree data structure with the following properties: (1) each node has a value, (2) the left subtree of a node contains only values less than or equal to the node's value, and (3) the right subtree of a node contains only values greater than the node's value. Binary search algorithm is a technique to find a specific element in a sorted list. The algorithm runs in $O(\log_2 N)$ time, where N is the total number of elements in the tree.

Three dictionaries are analyzed: the Roget's English, the Snort [3] and the ClamAV [2]. In the Rogets dictionary, most of the patterns are of the length between $1 - 16$ bytes and the longest pattern has a length of 24 bytes. Whereas in the Snort dictionary, the distribution is stretched out with the maximum length of 232 bytes. The ClamAV dictionary also has patterns whose length has thousands of bytes. These distributions suggest that a modular approach can be used. In this method, long patterns are cut into segments, which are matched against different chunks of input data. The final match result is determined based on these partial match results. The algorithm is introduced in Section III-C.

In order to use the BST as the data structure to perform fixed-length string matching, we need to preprocess the given (original) dictionary. Let L be the maximum length of the patterns. Each processed pattern has a bitmap vector of length L bits attached to it. The bitmap vector is a binary string, which indicates how many child-patterns are included in the parent pattern and what they are. A value of 1 at position i implies that there is a child-pattern with length i bytes, starting from the beginning of the parent pattern. For instance, if the parent pattern is "*andy*" and its bit-map vector is "0111", then there are 3 child-patterns included: "*an*", "*and*", and "*andy*", corresponding to the "1s" at position $2, 3$, and 4, respectively. A sample dictionary is shown in Figure 1(a). Note that a pattern can be the child (prefix) of more than one parent pattern.

A given dictionary can be simply preprocessed by merging the child patterns with their parents. Let N be the number of patterns and M be the size of the dictionary (in bytes). This preprocessing step has the computational complexity of $O(M \times N)$. Other data structure can be used to reduce the complexity to $O(M)$. Basically, we build the lexical tree from the original dictionary. The leaves of the tree are the patterns of the post-processing dictionary. The path from the root to each leaf represents the bit-map vector of the corresponding leaf (pattern).

The processed patterns are used to build the BST. Note that the bit-map vectors are only used for our matching purpose and not for building the BST. With the corresponding BST built, the string matching is performed by traversing left or right, depending on the result of the comparison at each node. If the input string is less than or equal to the node's value, it is forwarded to the left subtree of the node, and to the right subtree otherwise. These comparison results are calculated based on *Definition 3*.

A sample dictionary and its corresponding BST are illustrated in Figure 1. In this sample, patterns with maximum length of 4 characters are considered. Input stream

Table I
A SAMPLE DICTIONARY AND ITS MATCHING TABLE

(a) Split patterns

Pattern	Split pattern
beautiful	beau tifu l
plenary	plen ary
plentiful	plen tifu l
land	land
landing	land ing
bombbomb	bomb

(b) Patterns in the processed dictionary

Merged pattern	Bitmap	Label	Prefix flag	Suffix flag
ary	0010	1 (0001)	0	1
beau	0001	2 (0010)	1	0
bomb	0001	3 (0011)	1	1
ing	0010	4 (0100)	0	1
land	1001	5 (0101)	1	1
plen	0001	6 (0110)	1	0
tifu	0001	7 (0111)	0	1
beautifu		8 (1000)	1	
plentifu		9 (1001)	1	

(c) Matching table

Prefix label	Suffix label	Bitmap	New prefix label	Matched
0 (0000)	2 (0010)	0001	2 (0010)	0
0 (0000)	3 (0011)	0001	3 (0011)	0
0 (0000)	5 (0101)	1001	5 (0101)	1
0 (0000)	6 (0110)	0001	6 (0110)	0
2 (0010)	7 (0111)	0001	8 (1000)	0
3 (0011)	3 (0011)	0001	3 (0011)	1
5 (0101)	4 (0100)	0010	0 (0000)	1
6 (0110)	1 (0001)	0010	0 (0000)	1
6 (0110)	7 (0111)	0001	9 (1001)	0
8 (1000)	5 (0101)	1001	5 (0101)	1
9 (1001)	5 (0101)	1001	5 (0101)	1

is processed using a window of 4 characters, advancing 1 character at a time. Assume that 4-character input string of "*addi*" arrives. At the root of the tree, the input is compared with pattern "*car*" of the node to yield no match and a "*less than*" result. The input string traverses to the left. At *node 2*, it is compared with pattern "*andy*" to again yield the same result. The input string is then forwarded to the left. At *node 1*, the input string matches the node's pattern "*add*", which is the final result.

We must ensure that the proposed algorithm actually finds all the matching patterns for each input string, if any. Note that all matching patterns for an input string must be the prefix of the longest matching pattern. Hence, the preprocessing step ensures that these patterns are included in the same node with the longest one. Therefore, if the input string S_I reaches that node, then all the matching patterns should be found. We must prove that S_I does actually reach that node.

Claim: The input string S_I reaches the node that contains the longest matching pattern.

Proof: Assume that the input string S_I arrives at *node A*. Further assume that the longest matching pattern is located at *node B*, which belongs to the right subtree of *node A*. Note that *node B* is not necessarily the immediate child of *node A*. Let S_A and S_B denote the patterns stored at *node A* and *B*, respectively. By the property of BST, we have $S_A < S_B$. Let S'_B be the longest matching pattern of S_I, located at *node B*. We have $S_B = S'_B.*$. If S'_B is a prefix of S_A then it is already matched at *node A*. Otherwise, assume that S'_B is not a prefix of S_A. We have $S_A < S_B$ and $S_B = S'_B.*$ $\Rightarrow S_A < S'_B.*$. Since S'_B is not a substring of S_A, if $S_A > S'_B$ then $S_A > S_B$, which contradicts the property of BST. Hence, $S_A < S'_B$. By *Property 2*, $S'_B < S_I$, as S'_B is a prefix of S_I. Thus, $S_A < S_I$. The comparison result indicates that S_I is forwarded to the right subtree of *node A*. Similarly, if *node B* belongs to the left subtree of *node A*, the same result can be derived. This reasoning can be applied recursively until S_I visits *node B*. The result also implies that the last matched pattern is the longest one. ■

C. Arbitrary-Length String Matching

Consider a pattern matching module that can process fixed-length patterns of L bytes. This module can be used as the building block to process patterns with arbitrary length. String matching, in which patterns have variable lengths, can be performed as follows. Long patterns,whose widths are *longer* than L, are partitioned into segments of L bytes in length (the last segment may be shorter). The first sub-pattern is called prefix-pattern, and the subsequent ones are called suffix-patterns. The concatenation of a prefix-pattern and its following suffix-pattern yields a new (or intermediate) prefix-pattern. These new prefix-patterns, however, are not stored in the pattern database. For instance, with $L = 4$, pattern "*beautiful*" is split into a prefix-pattern "*beau*", an intermediate prefix-pattern "*beautifu*", and a suffix-pattern "*l*". All patterns in the given dictionary are processed similarly. The resulting prefix-patterns and suffix-patterns are merged into one database using the same preprocessing step as in the case of fixed-length string matching. Note that a processed pattern can either be a prefix, a suffix, or both. Hence, in addition to the bitmap vector, each processed pattern has: (1) a prefix flag, (2) a suffix flag, and (3) a unique label (or index to identify this pattern in the dictionary). A short pattern is also considered as a suffix-pattern. A sample dictionary and its split patterns are shown in Table I(a). The processed patterns are shown in Table I(b). The intermediate prefix-patterns (not colored) are only used to construct the matching table. Note that pattern "*l*" is merged with pattern"*land*".

The matching table is built by assembling the prefix with the corresponding suffixes to form valid long patterns. Given the matching table, the overall matching process is as follows. If a prefix-pattern is matched, we store the matched pattern's label. L cycles later, if a suffix-pattern is matched, the concatenation of the prefix-pattern's label and the suffix-pattern's label is checked if it forms a long pattern. The corresponding matching table of a sample dictionary is depicted in Table I(c). An example is used to walk through the operation of the algorithm. Suppose the input string is

Table II
16-CYCLE WALK-THROUGH MATCHING EXAMPLE

Cycle	Prefix label	Suffix label	Matched bitmap	New prefix label	Matched
1	0 (0000)	2 (0010)	0001	2 (0010)	0
2	0 (0000)	0 (0000)	0000	0 (0000)	0
3	0 (0000)	0 (0000)	0000	0 (0000)	0
4	0 (0000)	0 (0000)	0000	0 (0000)	0
5	2 (0010)	7 (0111)	0001	8 (1000)	0
6	0 (0000)	0 (0000)	0000	0 (0000)	0
7	0 (0000)	0 (0000)	0000	0 (0000)	0
8	0 (0000)	0 (0000)	0000	0 (0000)	0
9	8 (1000)	5 (0101)	1001	5 (0101)	1
10	0 (0000)	0 (0000)	0000	0 (0000)	0
11	0 (0000)	0 (0000)	0000	0 (0000)	0
12	0 (0000)	0 (0000)	0000	0 (0000)	0
13	5 (0101)	4 (0100)	0010	0 (0000)	1
14	0 (0000)	0 (0000)	0000	0 (0000)	0
15	0 (0000)	0 (0000)	0000	0 (0000)	0
16	0 (0000)	0 (0000)	0000	0 (0000)	0

"*beautifulanding*" and we want to search for patterns in the sample dictionary in Table I(c). We illustrate the steps in the following.

Table II describes a 16-cycle walk-through matching example of the input string "*beautifulanding*". Since $L = 4$ we need a buffer of size 4 to store the prefix labels. The buffer is initially empty (containing all $0s$). The algorithm looks up the 4-byte window of the input string in the dictionary and then shifts one byte at a time. At each position, the output matched-label and bitmap are prefixed with the output label of the buffer to form a *prefix-suffix-bitmap* concatenation. This combination is checked against the matching table to determine the next prefix label and any matching result. The new prefix label is used to update the corresponding entry in the buffer. Going back to the example, in cycle 1, there is a matched pattern "*beau*" with label 2 and matched bitmap "0001". Searching the combination "0000|0010|0001" in the matching table results in the new prefix label of 2 and no matched pattern. We update the first prefix label in the buffer. The next 3 cycles do not have any match. At cycle 5, there is a matched pattern "*tifu*" with label 7 and matched bitmap "0001". The matching table gives the new prefix label of 8 for intermediate prefix-pattern "*beautifu*". We update the first prefix label in the buffer. The next 3 cycles do not have any match. At cycle 9, there is a matched pattern "*land*" with label 5 and matched bitmap "1001". A search in the matching table yields the new prefix label of 5 and there are also 2 matched patterns: "*beautiful*" and "*land*". The process continues until the end of the input string is reached. There are a total of 3 matches reported at cycle 9 and 13 corresponding for "*beautiful*", "*land*", and "*landing*".

D. Arbitrary-Length String Matching Algorithm Analysis

Let K be the length of the input stream, L be the length of the longest pattern, and N be the total number of patterns. The proposed algorithm has $O(KL \log N)$ time complexity. Note that the time complexity is defined as the number of 1-char comparison. The Brute Force and the Aho-Corasick algorithm have time complexity of $O(KLN)$ and $O(K)$, respectively. As FPGAs provide large amount of parallelism, we can improve the complexity of our algorithm to $O(K \log N)$, by comparing L characters at a time. Note that the proposed algorithm does more work in order to improve time performance. The additional work can be performed by exploiting the intrinsic parallelism and computational power of FPGAs. Furthermore, by using pipelining techniques, we can reduce the complexity to $O(K)$, which is asymptotically comparable to that of the Aho-Corasick algorithm.

Let M denote the total number of characters in the original dictionary and N' denote the number of patterns in the processed dictionary ($N' \leq N$). Recall that each processed pattern needs a bit-map vector of length L bits. Each pattern also has a matched label of l bits so that it can be used to match arbitrary-length patterns. The total amount of required memory (in bits) is $(N'(9L+l))$. Therefore, the memory efficiency (in bytes/char) is $\frac{N'(9L+l)}{8M}$.

Note that this efficiency is calculated using the full size of the alphabet (256 characters). An Aho-Corasick (AC) design would have the worst-case memory storage of $(M|Ptr||\Sigma|)$, where $|Ptr|$ and $|\Sigma|$ are the size of the address pointer and the symbol set ($|Ptr| \leq \log_2 M$; $|\Sigma| = 256$ for ASCII character set). The size of Σ is one of the main reasons for the Aho-Corasick-based algorithm to have low memory efficiency.

IV. MASM Architecture

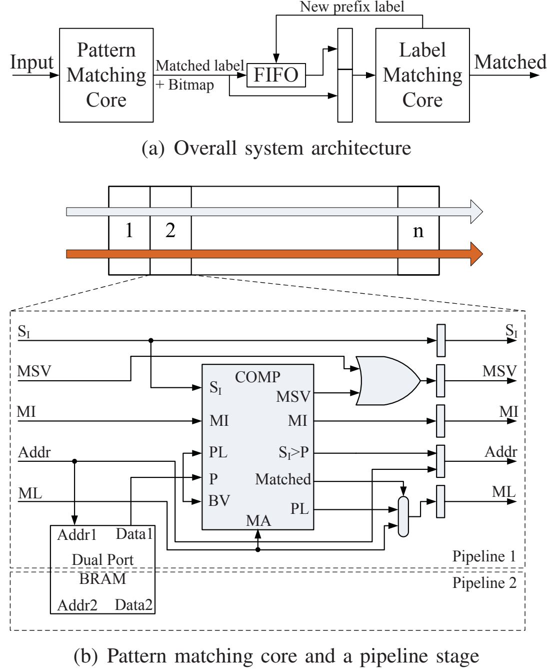
A. Overall Architecture

The overall architecture is depicted in Figure 2(a). As mentioned above, there are 2 matching cores: (1) pattern matching and (2) label matching. The first core matches the input string against the pattern database, while the second core matches the *prefix-suffix-bitmap* combination to validate the long pattern and output the matching result. The FIFO length is determined by the width of the matching window.

In the label matching core, all entries are uniquely defined. Hence, any matching mechanism can be utilized. The critical point is the relationship between the buffer length (or the size of the input window L) and the number of entries in the second core. *The buffer length should be at least* log *of the number of the entries*. For this reason, L should be chosen according to the size of the dictionary.

B. Pattern Matching Core Architecture

To ensure that every pattern matching results in the same number of operations or cycles, the input string continues with all the comparisons, even though matches may have already occurred. Pipelining is used to increase the throughput. The number of pipeline stages is determined by the height of the BST. Each level of the tree is mapped onto a pipeline stage, which has its own memory (or table). The size of table



(a) Overall system architecture

(b) Pattern matching core and a pipeline stage

Note: S_I - Input string; MSV - Matching status vector; MI - Matched info; $Addr$ - Address; ML - Matched label; P - Pattern's value; PL- Pattern's label; BV - Bitmap vector; $COMP$ - Comparator.

Figure 2. Block diagram of MASM architecture

doubles from one stage to the next. The number of patterns in a stage is determined by 2^i, where i is the stage index.

The block diagram of the basic pipeline and a single stage are shown in Figure 2(b). The on-chip memory of FPGA (BRAM) has dual-ported feature. To take advantage of it, the architecture is configured as dual-linear pipelines. This configuration doubles the matching rate. At each stage, the memory has 2 sets of Read/Write ports so that two strings can be input every clock cycle. The content of each entry in the memory includes: the pattern, its bit-map vector, the matched info, and the pattern's label. In each pipeline stage, there are 5 data fields forwarded from the previous stage: (1) the input string, (2) the matching status vector, (3) the matched info, (4) the memory access address, and (5) the matched label. The forwarded memory address is used to retrieve the pattern stored at the, which is compared with the input string to determine the matching status. In case of a match, the matched label (ML) and the matching status vector (MSV) are updated. The comparison result (1 if the input string is greater than the node's pattern, 0 otherwise) is appended to the current memory address and forwarded to the next stage. For instance, in the sample BST in Figure 1 (b), the input string ("*addi*") is less than the pattern of the root node ("*car*"); hence a "0" is appended to form the address of "0". At *node 2* (in level 1), the input string is again less than the node's pattern ("*andy*"). Thus, the memory address is "00", which is the address to access *node 1* in level 2. In the last comparison, if the input string were greater, then the memory address would be "01", which is the address of *node 3* in level 2.

C. Label Matching Core Architecture

The architecture of the label matching core is almost identical to that of the pattern matching core. The differences are (1) each entry does not have the bitmap vector, and (2) the matched info field is no longer required. The matching operation is, in fact, much simpler as it does not have to deal with the overlapping problem. Additionally, the size of the search key in each node is substantially shorter, making the comparison much faster and the size of the label table is significantly smaller than that of the prefix table. For these reasons, a small associative array or content-addressable memory (CAM) or some hashing architecture can be used to make the overall architecture simpler and more memory efficient.

D. Scalability of MASM

In the pre-processing step, one or more patterns can be merged in processed pattern. Therefore, the use of the complete BST structure leads to a sub-linear storage complexity in our design, as each node contains one *processed* pattern. The height of a complete BST is $(\lfloor \log_2 N \rfloor + 1)$, where N is the number of nodes of the tree. Since each level of the BST is mapped to a pipeline stage, the height of the BST determines the number of stages. Our proposed architecture is simple, and as shown in Section VI, utilizes a small amount of logic resource. Hence, the major constraint that dictates the number of pipeline stages, and in turn the size of supported dictionary, is the amount of on-chip memory. The memory size doubles as we go from one level to the next of a complete BST. Therefore, we can move the lower levels of the BST onto external SRAM stage(s). Consequently, for each additional stage on the external SRAM, the size of the supported dictionary is doubled. However, due to the limitation on the number of I/O pins of FPGA devices, we can only fit a certain number of stages on the external SRAM, as described in detail in Section V-B. The scalability of our architecture relies on the close relationship between the size of the supported dictionary and the number of pipeline stages. As the number of patterns increases, we simply add more stages.

V. Implementation

A. MASM without external SRAMs

As shown in Table III, the longer the segment length, the lower the memory efficiency. The system clock frequency of the design with longer segment length is also lower, due to longer propagation delay in the comparator. In this paper, we analyze 4 cases: $8, 12, 16, 24$-byte segments. The upper part of Table III shows the number of patterns of the Rogets and Snort dictionaries after the preprocessing step.

As previously stated, the number of nodes in the next level doubles that of the current level. Stage i has 2^i entries,

Table III
STATISTICS OF ROGETS AND SNORT DICTIONARIES WITH THE NUMBER OF PIPELINE STAGES AND THE REQUIRED BRAM

Dictionary	Rogets				Snort			
Dictionary size (byte)	$178K$				$146K$			
Segment length (byte)	8	12	16	24	8	12	16	24
# Merged patterns	17395	17720	17714	17716	15149	11924	10583	8718
# Pipeline stages	15	15	15	15	13	13	13	13
Total amount of memory (Mb)	1.48	2.19	2.83	4.11	0.51	0.77	1.01	2.02
# BRAM (18 Kb)	83	122	158	223	29	43	57	110
# BRAM (36 Kb)	42	61	79	112	15	22	29	55
Efficiency (byte/char)	1.05	1.51	1.94	2.82	1.07	1.17	1.32	1.69

$0 \leq i \leq height_of_BST$. Each entry includes: a pattern (length of L bytes, or $8L$ bits), a bit-map vector (of L bits), and the pattern label. The length of the label field is chosen to be 16 bits, to handle up to 64K nodes. That makes a total of $(9L + 16)$ bits per entry. Hence, each node needs $88, 124, 160, 232$ bits for segment lengths of $8, 12, 16, 24$ bytes, respectively. On a Xilinx FPGA, BRAM is available in blocks of 18 Kb (Virtex4), or 36 Kb (Virtex5-6). The distribution of the number of pipeline stages, and the required BRAMs for each segment length are shown in the lower part of Table III. A Virtex-4 FX140 (with 9936 Kb of on-chip memory, 552 blocks) can easily process up to 1.17M characters in a dictionary with a similar distribution as the Snort. A more recent device, such as Virtex-6, can handle up to 4.21-MB Snort-like dictionary.

B. MASM with external SRAMs

In our design, external SRAMs can be used to handle even larger dictionaries, by moving the storage of the last stage(s) of the pipelines onto external SRAMs. Currently, SRAM is available as $2-32$ Mb chips [1], with data widths of $18, 32, 36$ bits, and an access frequency of 550 MHz. Each stage uses dual port memory, which requires two address and two data ports. Let S_{node} denote the size (in bits) of each node (entry) of the BST. In general, the pipeline requires $2 \times (address_width + data_width) = 2 \times (i + S_{node})$ pins to connect stage i to its memory on the external SRAM. For instance, for segment length of 8 bytes, $S_{node} = 88$ (with 16-bit label field). Thus, stage 16 needs 208 pins, stage 17 needs 210 pins, and so on. The largest Virtex package, which has 1517 I/O pins, can interface easily with 6 banks of dual-ported SRAMs. Using this package, we can accommodate up to 6 external stages. For each additional external stage, the size of the supported dictionary is doubled. Moreover, since the access frequency of SRAM is twice that of our target frequency (250 MHz), the use of external SRAM should not adversely affect the performance of our design.

C. Modular Extensibility

Our architecture can be expanded either horizontally, vertically, or both. In the horizontal direction, more pipeline stages can be added to support larger dictionaries. If the amount of on-chip memory is not sufficient, we can extend the pipeline stages onto external SRAMs. In the vertical expansion, we duplicate the architecture multiple times. All the pipelines have the same memory content, more data streams can be matched simultaneously. Since matching in each pipeline is performed independently, the input stream can be interleaved to match different characters in each pipeline (intra-stream). This configuration leads to increase in the per-stream throughput by a factor equal to the number of replicated pipelines. Each pipeline can also match independent input streams (inter-stream), to increase the aggregated throughput. If the entire duplicated design cannot be fit onto one FPGA, then multiple chips can be utilized. In addition to these two scenarios, we can also expand the architecture in both directions to support larger dictionaries and achieve higher throughput at the same time.

VI. IMPLEMENTATION RESULTS

A. Throughput

The proposed pattern matching core was implemented in Verilog, using Synplify Pro 9.6.2 and Xilinx ISE 11.3, with Virtex-5 FX200T as the target. We assume that the label matching core is implemented using a small CAM or some type of hashing. As shown in Table III, even with segment size of 8 bytes, the total number of patterns in the processed dictionary is less than 20K, for both the Rogets and Snort dictionaries. A pipeline consisting of 15 stages can accommodate up to 32K patterns, and hence, is sufficient to hold the entire Rogets and Snort dictionaries. The implementation results for different pattern lengths are reported in Table IV. In all the cases, the amount of logic resource is very low (less than 7% of the total resource). For segment length of 24 bytes, the implementation achieved a frequency of 197 MHz. This result translates to a throughput of 1.6 Gbps per pipeline. Therefore, with a dual-pipeline architecture, this design achieves a throughput of 3.2 Gbps.

B. Performance Comparison

Two key comparisons were performed with respect to the memory efficiency and throughput. Table V shows the comparisons of MASM and the state-of-the-art designs. These candidates are the variable-stride [10], the bit-split [11], and the field-merge [16] approaches. The memory efficiency of MASM is calculated for the entire Rogets and Snort dictionaries with the segment size of 8 bytes. The performance results of these candidates that are used in the comparisons

Table IV
IMPLEMENTATION RESULTS FOR DIFFERENT PATTERN LENGTHS

Segment length (B)	8	12	16	24
Frequency (MHz)	220	217	210	197
Throughput (Gbps)	3.5	3.5	3.4	3.2

Table V
PERFORMANCE COMPARISON OF MASM, VARIABLE-STRIDE (V.S.), BIT-SPLIT (B.S.), AND FIELD-MERGE (F.M.) APPROACHES

Architecture	Memory efficiency (byte/char)	Throughput (GBps/stream)
MASM/Snort	1.07	3.5
MASM/Rogets	1.05	3.5
F.M./Snort	6.33	2.28
F.M./Rogets	2.16	2.28
B.S./Snort	34.1	1.76
B.S./Rogets	28.5	1.76
V.S./Snort	2.4	NA

were reported in [16]. We did not compare our design directly with other bloom filter or hashed AC-DFA approaches. The reasons are (1) the bloom filter approach can cause false positives, and (2) the AC-DFA implementation does not often take into account the implementation complexity of the hash functions. Hence, it is difficult to make a fair and meaningful comparison with these approaches.

The comparisons show that for the same dictionary, our design outperforms the state-of-the-art in terms of memory efficiency and per-stream throughput. MASM achieved over $2\times$ improvement in memory efficiency, compared with the state-of-the-art designs (the field-merge [16]). Our architecture requires only $\frac{1}{2}$ to $\frac{1}{6}$ the memory of the field-merge, while achieving $1.5\times$ the per-stream throughput. Compared with the bit-split design, our approach needs only $\frac{1}{28}$ to $\frac{1}{34}$ the memory of the bit-split design while achieving $1.8\times$ the per-stream throughput. With respect to scalability, our design can be partitioned to use BRAM+external SRAM, as discussed in Section V-B, to support larger dictionaries of over 8 MB. This can be done without sacrificing the sustained throughput.

VII. CONCLUDING REMARKS

This paper proposed and implemented a high-throughput, memory-efficient, and modular SRAM-based pipeline architecture for string matching (MASM), that does not use TCAM. By utilizing a binary search tree algorithm, the architecture achieves high memory efficiency close to 1 byte/char. Consequently, our architecture can support large dictionaries of over 4 MB, using only on-chip BRAM in a state-of-the-art FPGA. Employing external SRAM, this architecture can handle even larger dictionaries of over 8 MB. Our design sustained a lookup rate of 3.5 Gbps even when external SRAM is used. We are planning to improve our algorithm to match multiple characters per clock cycle and also looking at various techniques to reduce power consumption.

REFERENCES

[1] SAMSUNG SRAMs [Online]. [http://www.samsung.com].

[2] The Open Source Anti-Virus Database [Online]. [http://www.clamav.net/].

[3] The Open Source Network Intrusion Detection System [Online]. [http://www.snort.org].

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

[5] Z. K. Baker and V. K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on fpgas. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 135–144, Washington, DC, USA, 2004. IEEE Computer Society.

[6] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–257, Washington, DC, USA, 2004. IEEE Computer Society.

[7] J. Dharmapurikar, S. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24(10):1781–1792, 2006.

[8] M. French, E. Anderson, and D.-I. Kang. Autonomous system on a chip adaptation through partial runtime reconfiguration. In *FCCM '08: Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[9] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[10] 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.

[11] H.-J. Jung, Z. Baker, and V. Prasanna. Performance of fpga implementation of bit-split architecture for intrusion detection systems. *Parallel and Distributed Processing Symposium, International*, 0:177, 2006.

[12] R. Scrofano, M. B. Gokhale, F. Trouw, and V. K. Prasanna. Accelerating molecular dynamics simulations with reconfigurable computers. *IEEE Trans. Parallel Distrib. Syst.*, 19(6):764–778, 2008.

[13] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion. *FPL*, 2003:880–889, 2003.

[14] L. Tan, B. Brotherton, and T. Sherwood. Bit-split string-matching engines for intrusion detection and prevention. *ACM Trans. Archit. Code Optim.*, 3(1):3–34, 2006.

[15] 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*, pages 112–122, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Y.-H. E. Yang and V. K. Prasanna. Memory-efficient pipelined architecture for large-scale string matching. In *FCCM '09: Proceedings of the 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2009. IEEE Computer Society.

[17] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP '04: Proceedings of the 12th IEEE International Conference on Network Protocols*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.