A Memory-Efficient and Modular Approach for Large-Scale String Pattern Matching

Hoang Le, Student Member, IEEE, and Viktor K. Prasanna, Fellow, IEEE

Abstract—In Network Intrusion Detection Systems (NIDSs), string pattern matching demands exceptionally high performance to match the content of network traffic against a predefined database (or dictionary) of malicious patterns. Much work has been done in this field; however, most of the prior work results in low memory efficiency (defined as the ratio of the amount of the required storage in bytes and the size of the dictionary in number of characters). Due to such inefficiency, state-of-the-art designs cannot support large dictionaries without using high-latency external DRAM. We propose an algorithm called “leaf-attaching” to preprocess a given dictionary without increasing the number of patterns. The resulting set of post-processed patterns can be searched using any tree-search data structure. We also present a scalable, high-throughput, Memory efficient Architecture for large-scale String Matching (MASM) based on a pipelined binary search tree. The proposed algorithm and architecture achieve a memory efficiency of 0.56 (for the Rogets dictionary) and 1.32 (for the Snort dictionary). As a result, our design scales well to support larger dictionaries. Implementations on 45 nm ASIC and a state-of-the-art FPGA device (for latest Rogets and Snort dictionaries) show that our architecture achieves 24 Gbps and 3.2 Gbps, respectively. The MASM module can simply be 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 simply rewriting the content of the memory, which can be done quickly without reconfiguring the chip.

Index Terms—String Matching, Reconfigurable, Field Programmable Gate Array (FPGA), ASIC, Pipeline, Leaf Attaching, Aho-Corasick, DFA, Snort, Rogets.

1 INTRODUCTION

The power of the internet has grown explosively to a giant open network. Internet attacks require little efforts and monetary investment to create, are difficult to trace, and can be launched from virtually anywhere in the world. Therefore, computer networks are constantly assailed by attacks and scams, ranging from nuisance hacking to more nefarious probes and attacks. The most commonly used network protection systems are firewall and Network Intrusion Detection System (NIDS). They are critical network security tools that help protect high speed computer networks from malicious users. Firewall and NIDS are installed at the border of a network to inspect and monitor the incoming and out-going network traffic. Firewall, which performs only layer 3 or 4 filtering, processes packets based on their headers. NIDS, in contrast, provides not only layer-3 or 4, but also layer-7 filtering. NIDS searches both packet headers and payloads to identify attack patterns (or signatures). Hence, NIDS can detect and prevent harmful content, such as computer worms, malicious codes, or attacks being transmitted over the network. Such systems examine network communications, identify patterns of computer attacks, and then take action to either terminate the connections or alert system administrators.

With the rapid expansion of the Internet and the explosion in the number of attacks, design of 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 three main categories: ternary content addressable memory (TCAM)-based, dynamic/static random access memory (DRAM/SRAM)-based, and SRAM-logic-based solutions. Although TCAM-based engines can retrieve results 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 and SRAM-logic based solutions require multiple cycles to perform a search. Therefore, pipelining techniques are commonly used to improve the throughput. In the SRAM-logic based approach, a portion of the dictionary is implemented using logic resource, making this approach logic-bound and hard to scale to support larger dictionaries. The SRAM-based approaches, which are memory-bound, result in an inefficient memory utilization. This inefficiency limits the size of the supported dictionary. In addition, it is difficult to use external SRAM in these architectures, due to the constraint on the number of I/O pins. This constraint restricts the number of external stages, while the amount of on-chip memory upper-bounds 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 dictionary. This scalability has been a dominant issue for implementation of NIDs in hardware.

The key issues to be addressed in designing an architecture for string pattern matching engines are (1) size of the supported dictionary, (2) throughput, (3) scalability with respect to the size of the dictionary, and (4) dictionary update. To address these challenges, we propose a pre-processing algorithm and a scalable, high-throughput, Memory efficient Architecture for large-scale String Matching (MASM). This architecture utilizes binary search tree (BST) structure to improve the storage efficiency. MASM also provides a fixed-latency due to the linear pipelined architecture.

This paper makes the following contributions:
- An algorithm called leaf-attaching to efficiently disjoint a given dictionary without increasing the number of patterns (Section 5.1).
- An architecture that achieves a memory efficiency of 0.56 (for Rogets) and 1.32 byte/char (for Snort) (Section 8). State-of-the-art designs can only achieve the memory efficiency of over 2 byte/char in the best case.
- The implementation on ASIC and FPGA shows a sustained aggregated throughput of 24 Gbps and 3.2 Gbps, respectively (Section 8).
- The design can be duplicated to improve the throughput by exploiting its simple architecture (Section 7.5).

The rest of the paper is organized as follows. Section 2 introduces the background and related work. Section 3 lists the definitions and notations used in the paper. Section 4 presents the prefix properties used in our work. Section 5 and 6 detail the fixed-length and arbitrary-length string matching algorithms. Section 7 describes the proposed architectures. Section 8 presents the performance evaluations. Section 9 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 String Pattern Matching

String pattern matching (or simply 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 dictionary (or database) to the traffic passing through the device. Note that the string matching is also referred to as exact string matching.

Among currently available NIDS solutions, Snort [2] is a popular open source and cross-platform NIDS. Snort uses signatures and packet headers to detect malicious internet activities. As an open source system, Snort rules are contributed by the network security community to make widely-accepted and effective rule-sets. These rule-sets, which include both packet headers and signatures (strings and regular expressions), have grown quite rapidly, as rules are added as soon as they are extracted by the network security experts. The string patterns constitute the largest portion of the signatures in a Snort database. There are over 8K string signatures in the current Snort database (as of 03/29/2011). Though the sizes of these rule-sets are still moderate, it is not efficient to perform direct search on them. As a lightweight NIDS, Snort can easily be deployed on almost any node of a network, with minimal disruption to operations. However, when the traffic rate exceeds the throughput of this software-based system, additional incoming traffic is dropped. Rogets dictionary is another useful database of signatures. The dictionary often provides system hackers with viable passwords and wordlists, which conversely is used in the network security community to prevent the servers/systems/passwords from being hacked.

2.2 Related Work

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

In a TCAM-based architecture presented in [19], the entire ClamAV dictionary is stored onto TCAM, which has deterministic lookup time for any input. To reduce storage 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. The algorithm for matching long patterns is as follows. If a prefix-pattern is matched at the i-th position of the packet, it is recorded in memory. Later, if a suffix pattern is matched at position i + j (0 < j ≤ w), the concatenation of this suffix pattern and the previous prefix pattern is checked to see if it forms a long pattern. Dictionary updates can be done simply by adding or removing patterns from TCAM, whenever needed. Therefore, the advantages of this approach are high memory efficiency (close to 1 byte/char) and simple updating mechanism. Yet, the achieved throughput of 2 Gbps is reasonably low.

In a pipelined NFA string matching architecture [4], [7], [14], all the given string patterns are combined to build a NFA. The NFA, which is implemented as a pipelined network, matches multiple string patterns in parallel. The architectures allow matching more than one character per clock cycle, either by replicating a match pipeline [14], or by using a multi-character decoder [4]. By accepting multiple characters per cycle, these designs were able to achieve a throughput of 8-10 Gbps. However, the main drawbacks of this approach are the inflexibility and relatively small size of the supported dictionaries. This approach also has a potentially high resource utilization. The logic cell saving method, which
was proposed in [4], is based on the idea that patterns sharing characters do not need to be redundantly compared. By pushing all character-level comparisons to the beginning of the comparator pipelines, the single character match operation is simply reduced to the inspection of a single bit. While this approach can improve the resource utilization, it is highly dictionary dependent; hence, it requires the re-synthesis of the entire circuit for any dictionary updates. This makes on-line dictionary updates impossible, and also makes it difficult to predict the implemented performance. Since these designs are bounded by the logic resource rather than memory resource, the memory efficiency was not reported.

The field-merge NFA architecture [18] is similar to the bit-split NFA discussed above. The algorithm partitions every input character into \( k \) bit-fields, and constructs a tree-structured NFA, called the full state machine (FSM). The dictionary is then divided “vertically” into orthogonal bit-fields. Each bit-field is used to construct a tree-structured NFA, called the partial state machine (PSM), with fewer bits per transition label. Every match state in the FSM, called a full match state, corresponds to exactly one partial match state in every PSM. In addition, during the construction of the PSMs, a per-level auxiliary table (ATB) is maintained to map every full match state uniquely into a set of partial match states on the same level, one from each PSM. The novelty of this approach is that by using a relatively small ATB, both pipeline traversals and output merging of the PSMs can be made simple. 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 large variances among different dictionaries (2.16 byte/char for the Rogets and 6.33 byte/char for the Snort dictionary).

The bit-split DFA architectures [11], [16] are based on Aho-Corasick (AC-DFA) algorithm [3]. 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 amount of memory wastage [17], and makes memory bandwidth and latency the bottlenecks for string matching based on AC-DFA. To improve the memory efficiency, Tan et. al. proposed the bit-split algorithm [17], which splits a full AC-DFA into several split state machines. Each machine accepts a small portion (1 or 2 bits) of the input as the transition labels. A partial match vector (PMV), one bit per pattern in the dictionary, is maintained at every state in the split state machine to map the state to a set of possible matches. At every clock cycle, the PMVs from all split state machines are AND-ed together to produce a full match vector (FMV) to generate the actual matches. While this approach increases the memory efficiency, the construction of the split state machines from the AC-DFA can be very expensive for large dictionaries with thousands of states. Moreover, in every clock cycle, the FMV results need to be aggregated, incurring a tree-like structure whose routing delay is likely to be costly. Even though it is possible to optimize the performance of the bit-split approach, such an optimization requires resource planning for each group, and makes dynamic dictionary updates more difficult. The memory efficiency was reported as 23 and 33 byte/char in [11] and [16], respectively.

The variable-stride (VS) multi-pattern matching architecture [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, which is similar to the Winnowing idea developed for document fingerprint matching. In the pre-processing step, all patterns are first segmented into a sequence of blocks. The first and last blocks are called the head block and tail block, respectively. The core blocks of patterns are extracted and used as alphabet symbols to build the VS-DFA. When all the core blocks match, the head and tail need to be compared to ensure the correct pattern is matched. 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.

The major problems in these SRAM-based prior work are low memory efficiency and relatively small size of supported dictionaries. Besides, these designs more or less depend on the size of the alphabet, which lowers the memory efficiency as the size increases. It is also unclear how well these designs can be scaled to support larger dictionaries. On the other hand, TCAM solutions have good memory efficiency and high scalability, but low throughput and high power dissipation. These problems become more severe as the pattern database grows. Our goal is to find a good solution to leverage and take advantages from the best of both approaches.

### 2.3 Problem Definition

The string pattern matching problem can be formulated as follows. Given an input string of length \( K \), an alphabet \( \Sigma \) consisting of all the ASCII characters (\( |\Sigma| = 256 \)), and a dictionary consisting of \( N \) string patterns, \( \{S_0, S_1, \ldots, S_{N-1}\} \), whose characters belong to the alphabet \( \Sigma \), find and return all occurrences, if any, of every pattern in the given input string.

### 3 Definitions and Notations

Fig. 1 shows the corresponding prefix tree [13] representation of the sample dictionary shown in Table 1. Given a dictionary and the corresponding prefix tree representation, the following terms and notations are defined:

1. **String, pattern, or signature**: a distinctive characteristic that identifies any malicious content.
TABLE 1: A sample pattern database (dictionary) (maximum pattern length is 8)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>an</td>
</tr>
<tr>
<td>$P_2$</td>
<td>and</td>
</tr>
<tr>
<td>$P_3$</td>
<td>andy</td>
</tr>
<tr>
<td>$P_4$</td>
<td>ant</td>
</tr>
<tr>
<td>$P_5$</td>
<td>anchor</td>
</tr>
<tr>
<td>$P_6$</td>
<td>bee</td>
</tr>
<tr>
<td>$P_7$</td>
<td>be</td>
</tr>
<tr>
<td>$P_8$</td>
<td>been</td>
</tr>
</tbody>
</table>

2) * (star): any string including the empty one$^1$.
3) $\{n\}$: repeat the previous item exactly $n$ times, where $n$ is a positive integer$^1$.
4) $|S|$ is the length (in bits) of string $S$ (e.g. $|abcd| = 32$ as each ASCII character occupies 8 bits).
5) $S_1$ is a prefix of $S_2$ if and only if $S_2 = S_1*$. If $S_1 = ab$, $S_2 = abcd$. $S_1$ is said to be a prefix-child of $S_2$, and $S_2$ is a prefix-parent of $S_1$. For simplicity, the terms child and parent are used to indicate the prefix-child and prefix-parent, respectively. In the special case when $S_1 = S_2$, then $S_1$ is the prefix of $S_2$, and vice versa.
6) $S_1$ and $S_2$ are said to be distinct if and only if $S_1$ is not a prefix of $S_2$, and $S_2$ is not a prefix of $S_1$. Otherwise, they are said to be overlapped. A set of patterns such that all pairs of patterns are disjoint is called a disjoint pattern set.
7) $S_1 < S_2$ if and only if the first non-matching character (from the left) in $S_1$ has a smaller ASCII value than the corresponding character in $S_2$ (e.g. $S_1 = abcd$, $S_2 = abef$). $S_1 > S_2$ since $c < e$.
8) The binary representation of string $S$, denoted as $B_S$, is obtained by converting all characters of $S$ into their corresponding ASCII values.
9) The range representation of string $S$, using $L$ bits $(L > |S|)$, is denoted as $R_S = [R_{S_L}, R_{S_R}]$, where $R_{S_L} = B_S(0)$ and $R_{S_R} = B_S(1)$ such that $n = L - |S|$. In the case of $L = |S|$, $R_S = [B_S, B_S]$.5
10) Any node, for which the path from the root of the prefix tree to that node represents a pattern in the dictionary, is called a pattern node. If a pattern is at a leaf-node, then it is called a leaf-pattern node. Otherwise, it is called a non-leaf-pattern node.

4 PREFIX PROPERTIES

The string matching can be formalized as a prefix matching problem, with the exception that all the matches are returned, not only the longest one. Consider two strings, $S_A$ and $S_B$, in a $L$-bit number space $\Omega = \{0(L), 1(L)\}$, $L \geq \max(|S_A|, |S_B|)$. Let $n_A = L - |S_A|$, $n_B = L - |S_B|$, and $n_{BA} = \min(|S_B| - |S_A|)$. We first introduce the properties that 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.

Fig. 1: Prefix tree for the sample dictionary in Table 1

Proof: In $\Omega$, each string represents a number range. 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 $[S_A0(n_A), S_A1(n_A)]$ and $[S_B0(n_B), S_B1(n_B)]$, respectively. Since $S_A < S_B$, we have $S_A1(n_A) < S_B0(n_B)$; therefore, the two ranges do not overlap.

Property 2: Given two distinct strings, $S_A$ and $S_B$, if they overlap then one must be a prefix of the other.

Proof: In $\Omega$, the range of $S_A$ and $S_B$ are $[S_A0(n_A), S_A1(n_A)]$ and $[S_B0(n_B), S_B1(n_B)]$, respectively. Without loss of generality, assume that $S_B0(n_B) \subseteq [S_A0(n_A), S_A1(n_A)]$. This result implies that $S_A$ is a prefix of $S_B$.

Property 3: 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 $S_B = S_AX(n_{BA})$, where $X = \{0, 1\}$. Thus, $S_B0(n_B) = S_A0(n_{BA})0(n_B)$, but $S_AX(n_{BA})0(n_B) > S_A0(n_{BA})0(n_B) \Rightarrow S_B0(n_B) > S_A0(n_A)$. Therefore, $S_A < S_B$.

5 FIXED-LENGTH STRING MATCHING ALGORITHM

5.1 Leaf-Attaching Algorithm

Tree search algorithm is a good choice for a string matching engine as the lookup latency does not depend on the length of the pattern, but on the number of patterns in the dictionary. In the case of the tree search, the latency is proportional to $\log$ of the number of patterns. However, in order to use tree search algorithms, the given set of patterns needs to be processed to eliminate the overlap between patterns.

Leaf-pushing [15] can be used to eliminate overlap. Leaf-pushing by itself is a simple idea. In this approach, the initial step is to build a prefix tree from the given dictionary. Leaf pushing first grows the prefix tree to a full tree, and then pushes all the non-leaf patterns to the leaf nodes. While leaf pushing eliminates the overlap between patterns, it has the negative effect of exponentially expanding the size of the dictionary. This
is due to the sparseness of the prefix trie. Although each node of the prefix tree can potentially have up to 256 children (corresponding to the number of ASCII characters), the actual number of children of each node is often much less.

We propose an algorithm to disjoint the patterns, called leaf-attaching. The algorithm takes a prefix tree as the input, and outputs a set of disjoint patterns. All the child (or non-leaf) patterns are merged with their parent (or leaf) patterns. Let \( L \) be the maximum length of the patterns. Each parent pattern has a match vector of length \( L \) bits attached to it. The match vector is a binary string, which indicates how many child-patterns are included in a 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 \textit{andy} and its match vector is 0111, then there are 3 child-patterns included: \textit{an}, \textit{and}, and \textit{andy}, corresponding to the 1 at positions 2, 3, and 4, respectively. Note that a pattern can be the child (prefix) of more than one parent pattern. Fig. 2 shows the sample merging for 2 parent patterns, \textit{andy} and \textit{between}.

![Fig. 2: Sample merging](image)

A prefix tree is built from a given set of patterns. Each node of the tree includes: (1) a leaf bit, (2) a pattern bit, (3) a pattern (if it is a pattern-node), and (4) a match vector (if it is a leaf-node). The leaf bit and pattern bit determine if the node is a leaf node or a pattern node, respectively. The leaf-attaching algorithm performs an in-order traverse of the prefix tree and pushes the non-leaf patterns to their leaf-patterns. The pseudo-code is given in Algorithm 1. Fig. 3 shows the result of the prefix tree in Fig. 1 after being leaf-attached. Once the prefix tree is leaf-attached, the leaves, along with their associated match vector, are collected to form a disjoint pattern set. Table 2 shows the sample dictionary (in Table 1) after leaf-attaching.

![Fig. 3: Leaf-attached tree](image)

### Algorithm 1 LEAF_ATTACH(node, match_vector)

**Input:** A prefix tree  
**Output:** A prefix tree with all the non-leaf patterns attached to the leaf-patterns

1: if (node is NULL) then  
2: return  
3: end if  
4: if (node is a leaf node) then  
5: \[ \text{node.match.vector} = \text{CONCATENATE(match.vector, 1)} \]  
6: return  
7: end if  
8: if (node is not root) then  
9: if (node is a pattern-node) then  
10: \[ \text{new.match.vector} = \text{CONCATENATE(match.vector, 1)} \]  
11: Set node to non-pattern node  
12: else  
13: \[ \text{new.match.vector} = \text{CONCATENATE(match.vector, 0)} \]  
14: end if  
15: else  
16: \[ \text{new.match.vector} = \text{match.vector} \]  
17: end if  
18: \[ i = 0 \]  
19: while \( i < 256 \) do  
20: LEAF_ATTACH(node.children[i], new.match.vector)  
21: \[ i = i + 1 \]  
22: end while  
23: return

### Complexity

Let \( N \) denote the number of patterns in the given dictionary and \( L \) be the maximum length of the patterns. The step to build the prefix tree for all the patterns has a computational complexity of \( O(N \times L) \), since each character of the patterns needs to be processed. The proposed algorithm then traverses to each node of the prefix tree and pushes the patterns to the leaves. This step also has a complexity of \( O(N \times L) \). Therefore, the overall complexity of the leaf-attaching algorithm is \( O(N \times L) \).

### 5.2 Binary Search Tree (BST) String Matching Algorithm

In this section, a memory efficient data structure based on a complete binary search tree (BST) [8] is presented. Complete 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 that are less than the node’s value.
3) The right subtree of a node contains only values that are greater than the node’s value.
4) All the levels are fully occupied except possibly the last level. If the last level is not full, then all the nodes are as far left as possible.

The binary search algorithm is a technique to find a specific element in a sorted list. In a balanced binary search tree, an element if it exists, can be found in at most \([\log_2 (N + 1)]\) operations, where \(N\) is the total number of nodes in the tree.

The given dictionary is leaf-attached and the leaf patterns along with their match vector are extracted. The leaf patterns are used to build the BST. Each node in the BST includes a pattern and a match vector. Note that the match vectors are only used for our matching purpose and not for building the complete 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, or to the right subtree otherwise.

A sample complete BST built for the set of disjoint patterns in Table 2 is illustrated in Fig. 4. In this sample, patterns with the maximum length of 8 characters are considered. The complete BST can be mapped as follows. Nodes in each level are stored in a contiguous memory block. Let \(x\) denote the index (in base-2) of node \(A\). \(A\) has 2 child nodes with the index of \(2x\) (left child) and \(2x + 1\) (right child). In base-2 number system, \(2x = \{x, 0\}\) and \(2x + 1 = \{x, 1\}\), where \(\{\}\) represents the concatenation (in base-2). Using this memory mapping, there is no need to store the child pointers at each node. Instead, these pointers are calculated explicitly on-the-fly by simply concatenating the address of the current node and the comparison result bit.

Input stream is processed using a window of 8 characters, advancing 1 character at a time. Assume that an 8-character input string \(\text{anchor}\) arrives. The matching status vector is reset (\(MSV = 00000000\)). At the root of the tree (\(P_{12}\)), the input is compared with pattern \(\text{car}\) of the node to yield no match and a “less than” result. The input string traverses to the left. At Node \(P_{11}\), it is compared with pattern \(\text{beat}\) to again yield the same result. The input string is then forwarded to the left. At Node \(P_9\), the input string matches the node’s pattern \(\text{andy}\) to yield a match and a “less than” result. As the input string matches with the child pattern \(\text{an}\), the second bit of the \(MSV\) is set, \(MSV = 01000000\). The input string then traverses to the left. At Node \(P_{5}\), the input string matches the node’s pattern \(\text{anchor}\) and \(MSV = 01000010\), which is the final result.

We must ensure that the search algorithm actually finds all the matching patterns for each input string, if any. Note that all the matching patterns for an input string must be the prefix of the longest matching pattern. Hence, the leaf-attaching 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 the input string \(S_I\) 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 \(S_B = S_B^* = S_A < S_B^*\). 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.

**5.3 Memory Efficiency of The BST String Matching Algorithm**

In this section, the set of disjoint patterns \((S_D)\) generated by the proposed leaf-attaching algorithm is considered. The following notations are used in our analysis:

- \(L\): Maximum length of the disjoint patterns (in bytes)
- \(\bar{L}\): Average length of the disjoint patterns, \(1 \leq \bar{L} \leq L\)

The memory efficiency of the proposed algorithm is given by the following equation:

\[
\text{Memory Efficiency} = \frac{\text{Size of proposed algorithm}}{\text{Size of conventional algorithm}}
\]


- $N_D$: Number of disjoint patterns
- $M_D$: Size of the set of disjoint patterns (in bytes)
- $M_{BST}$: Size of the memory required to store $S_D$ (in bytes) using the BST data structure

Each disjoint pattern has a match vector with the length of $L$ bits (or $\frac{L}{2}$ bytes). $M_D$ and $M_{BST}$ are calculated using (1) and (2), respectively. The memory efficiency of the BST string matching algorithm is defined in (3).

\[
M_D = N_D \times L \tag{1}
\]

\[
M_{BST} = N_D \times \left( L + \frac{L}{8} \right) = \frac{9}{8} N_D \times L \tag{2}
\]

\[
\frac{M_{BST}}{M_D} = \frac{9}{8} \times \frac{L}{L} \tag{3}
\]

It is observed from (3) that the memory efficiency of the BST string matching algorithm depends on the ratio $L/L$. The best-case memory efficiency is achieved when $L = L$, and the worst case happens when $L = 1$. The analysis results of the Roget’s dictionary and the current Snort database along with the memory efficiency of the BST string matching algorithm are shown in Table 3. The results agree with our observation.

**TABLE 3:** Memory efficiency (in byte/char) of the BST string matching algorithm for the Rogets and Snort dictionary

<table>
<thead>
<tr>
<th>$L$ (characters)</th>
<th>Rogets</th>
<th>Snort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>323</td>
</tr>
<tr>
<td>$L$ (characters)</td>
<td>7.92</td>
<td>27.61</td>
</tr>
<tr>
<td>$L/L$</td>
<td>2.78</td>
<td>11.70</td>
</tr>
<tr>
<td>$M_{BST}/M$ (byte/char)</td>
<td>3.13</td>
<td>13.16</td>
</tr>
</tbody>
</table>

### 5.4 Cascading approach

As previously stated, the memory efficiency of the BST string matching algorithm can be improved by reducing the ratio of $L/L$. This can be done by splitting the patterns into segments of length $L_S$. The set of disjoint patterns ($S_D$) is divided into a collection of segmented sets. Segmented set $i$ consists of segments $i$ of all the disjoint patterns in set $S_D$. For each segmented set $i$, a prefix tree $i$ is built and then leaf-attached. The leaf-patterns of the leaf-attached tree $i$ are collected into set $S_D^0$, $0 \leq i < k$, $k = \lceil L/L_S \rceil$.

Each disjoint set $S_D^i$ is matched by a BST module (or a segment BST). These BSTs are cascaded as shown in Fig. 5. Each node in the BSTs includes: (1) a search key, (2) a match vector, and (3) a label. A label with non-zero value (or simply non-zero label) indicates that there is subsequent segment(s). In this case, the label is used to verify the match in the next BST.

$BST_0$ uses the first segment of the patterns as the search key, while $BST_i$ ($i > 0$) uses the concatenation (in base-2) of the label of the previous segment and segment $i$ of the patterns as the search key. Each $BST$ outputs 2 data fields: (1) a matching status vector ($MSV$) and (2) a match label ($ML$). The match label is the label of the node that matches the entire input string. Note that search in $BST_i$ is necessary only if $BST_{i-1}$ returns a non-zero label.

The set of disjoint pattern ($S_D$) in Table 2 is used to illustrate the idea. In this example, the patterns are cut into segments of length 4 characters. Since $L = 8$, $L_S = 4$, Set $S_D^i$ is cut into $k = \lceil L/L_S \rceil$, $L^2$ segmented sets. For each set, a prefix tree is built and then leaf-attached. The leaf-patterns of the leaf-attached trees are collected into sets $S_D^0$ and $S_D^1$. The patterns of these 2 sets are shown in Table 4. Fig. 6 describes the cascaded BSTs, the $BST_0$ built for set $S_D^0$, and the $BST_i$ for Set $S_D^1$.

Input stream is processed using a window of 8 characters, advancing 1 character at a time. Assume that an 8-character input string anch orer arrives. The input string is split into 2 substrings: anch and orer. The first substring is matched in $BST_0$. The second substring waits for the results from $BST_0$ and will be matched in $BST_1$.

At the beginning of $BST_0$, the matching status vector is reset ($MSV_0 = 0000$). The input string $S_I = anch$ traverses the BST to find the match, using the similar search operation described in the previous section. $S_I$ fully matches the node with pattern anch and $MSV_0 = 0100$. Thus, the label (0001) of the matching node is also returned. The match label is then concatenated with the second substring orer. This combination is used as the search key in $BST_1$. Search in $BST_1$ should yield a match at the node with the search key {0001, org} and $MSV_1 = 0100$. The final matching status vector is the concatenation (in base-2) of all the partial matching vectors.
Therefore, $\text{MSV} = \{\text{MSV}_0, \text{MSV}_1\} = 01000100$, which indicates that patterns an and anchor are matched. This is the final result.

### 5.5 Comparison of memory efficiency for the single BST and cascaded BST approaches

The set of disjoint patterns $S_D$ shown in Table 2 is used in our analysis. Let $M$ denote the size of $S_D$ (in bytes) shown in Table 2, $M_{\text{BST}}$ be the size of the memory required to store $S_D$ (in bytes) using the single BST data structure, and $M_{\sum \text{BST}}$ be the size of the memory required to store $S_D$ (in bytes) using the cascaded BST data structure. $M$ is calculated by summing up the size of all the patterns. $M_{\text{BST}}$ is calculated using (2). $M_{\sum \text{BST}}$ is calculated by summing up the size of the 2 BSTs. Each node of the BSTs has an additional label field. There are a total of 10 nodes in $\text{BST}_0$; hence, 4 bits are sufficient to identify these labels. Therefore, $M = 42, M_{\text{BST}} = 90, M_{\sum \text{BST}} = 61$. The memory efficiency for the single BST approach and the cascaded BST approach is 2.14 and 1.45 byte/char, respectively. Note that the memory efficiency is calculated for the BST only. For the overall memory efficiency, $M$ should be the size of the given dictionary. In this case, $M = 57$, and the memory efficiency for the single BST approach and the cascaded BST approach is 1.45 and 1.07 byte/char, respectively. The analysis results for the Rogets and Snort dictionaries with various segment lengths also show improved memory efficiency in the cascaded BST approach, compared with that of single BST approach. These results are shown in Table 5.

### Table 5: Memory efficiency (in byte/char) of the cascaded-BST string matching algorithm with various segment lengths for the Rogets and Snort dictionary

<table>
<thead>
<tr>
<th>$L_S$ (character)</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rogets (byte/char)</td>
<td>0.56</td>
<td>1.21</td>
<td>1.70</td>
<td>2.19</td>
<td>2.69</td>
</tr>
<tr>
<td>Snort (byte/char)</td>
<td>1.11</td>
<td>1.27</td>
<td>1.29</td>
<td>1.43</td>
<td>1.40</td>
</tr>
</tbody>
</table>

### 6 Arbitrary-Length String Matching Algorithm

#### 6.1 Arbitrary-Length Pattern Splitting

The cascaded BST approach has 2 drawbacks: (1) the maximum length of the patterns must be fixed and (2) a dictionary with long patterns may potentially require many segmented BSTs. To overcome these issues, we propose a string matching algorithm for arbitrary-length patterns. Consider a pattern matching module that can process fixed-length patterns of $L$ bytes. This module is used as the building block to process patterns with arbitrary length. Note that this module can be a single-BST module or a cascaded-BST module. 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 segment is called prefix-pattern, and the subsequent segments are called suffix-patterns. The concatenation (in base-2) of a prefix-pattern and its following suffix-pattern yields a new (or intermediate) prefix-pattern. However, these new prefix-patterns are not stored in the split-pattern database. For instance, with $L = 3$, pattern between is split into a prefix-pattern bet, an intermediate prefix-pattern betwee, and a suffix-pattern n. All patterns in the given dictionary are processed similarly. The resulting prefix-patterns and suffix-patterns are merged into a split-pattern database, or dictionary. A prefix tree is built for this database. The prefix tree is then leaf-attached and the leaf-patterns are extracted into a set of disjoint patterns $S_D$. The preprocessing steps are similar to the case of fixed-length string matching. Note that a pattern in $S_D$ can either be a short pattern, a segment of a
long pattern, or both. Hence, in addition to the match vector, each pattern has: (1) an unique label and (2) a short match vector. The unique label is used to identify the pattern in the merged dictionary, whereas the short match vector is to keep track of the short patterns included in that pattern. For example, split pattern *anchor* has the short match vector 0100, indicating that pattern *an* is a short pattern (corresponding to the 1 at the second position of the vector). A sample dictionary and its split patterns of length 4 are shown in Table 6(a). The disjoint patterns of the corresponding split-pattern dictionary are shown in Table 6(b). Note that the construction of the prefix tree for the split-pattern dictionary and the leaf-attaching steps are skipped due to space limitation.

The label matching table is built by assembling the prefix with the corresponding suffixes to form valid long patterns. The overall matching process consists of 2 steps. Input stream is processed using a window of *L* characters, advancing 1 character at a time. In the first step, *L* input characters are matched against the split-pattern table. If there is a match, the corresponding match vector and match label are returned. In the second step, if a prefix-pattern is matched, we store the label of the matched pattern. *L* cycles later, if a suffix-pattern is matched, the concatenation (in base-2) of the prefix-pattern label and the suffix-pattern label is checked (against the label matching table) if it forms a long pattern. Note that the short patterns, whose length is shorter than or equal to the segment length, are reported in the first step. The corresponding matching table for the split-pattern dictionary in Table 6(b) is depicted in Table 6(c). An example is used to walk through the operation of the algorithm. Suppose the input string is *anchorinbetween* and we want to search for patterns in the sample dictionary in Table 6(c). We illustrate the steps in the following.

### 6.2 Arbitrary-Length Pattern Matching Algorithm

Table 7 describes a 16-cycle walk-through matching example of the input string *anchorinbetween*. 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 advances the input stream one byte at a time. At each position, the output matched label (or suffix label) and match vector are prefixed with the output label of the buffer (or prefix label) to form a \{prefix_label, suffix_label, match_vector\} concatenation (in base-2). This combination is checked against the label 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, the input is *anchor*, there is a matched split-pattern *anchor* with label 3 and a matched vector 0101. The short match vector 0100 indicates that there is a short pattern (an) that matches the input stream. Therefore, a match result is returned. Searching the concatenation of \{0, 3, 0101\} in the matching table results in the new prefix label of 3. We update the first prefix label in the buffer. The next 3 cycles do not have any match. At cycle 5, the input is *orin*, there is a matched split-pattern *orin* with label 12 and matched vector 0100. Searching the concatenation of \{3, 12, 0101\} in the matching table gives a match result for the complete pattern *anchor*, and the matched label 0. 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 split-pattern *between* with label 6 and matched bitmap 0101. A match result is returned for short pattern *between*. Search in the matching table yields the new prefix label of 6. The process continues until the end of the input string is reached. There are a total of 4 matches reported at cycle 1, 5, 9, 13, corresponding to matched patterns *an*, *anchor*, *be*, and *between*, respectively.

### 6.3 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 \(O(K \times L \times \log N)\) 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 a time complexity of \(O(K \times L \times N)\) and \(O(K)\), respectively. We can improve...
the complexity of our algorithm to $O(K \times \log N)$, 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 parallel hardware platforms, such as ASIC and 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.

### 7 Architecture

#### 7.1 Overall architecture

We propose a scalable, high-throughput, Memory efficient Architecture for arbitrary-length String Matching (MASM) based on pipelined binary search tree. The block diagram of the architecture is depicted in Figure 7. As mentioned above, there are 2 matching steps: (1) pattern matching and (2) label matching, handled by the pattern matching module (PMM) and label matching module (LMM), respectively. Input data stream is fed into PMM $L$ bytes at a time. This input window is advanced 1 byte per clock cycle. PMM then matches the input string against the pattern database, while LMM matches the \{prefix, suffix, match_vector\} combination to validate the long pattern and outputs the matching result.

In LMM, all entries are uniquely defined. Hence, any matching mechanism can be utilized. The critical point is the relationship between the size of the input window $L$ and the number of entries in the LMM. The window size $L$ should be greater than or equal to the matching latency of the LMM. For this reason, $L$ should be chosen according to the size of the dictionary.

#### 7.2 Pattern Matching Module (PMM) Architecture

Pipelining is used to produce one lookup operation per clock cycle to increase the throughput. The number of pipeline stages is determined by the height of the search tree. Each level of the tree is mapped onto a pipeline stage, which has its own memory (or table). The cascading BST approach is utilized to improve the memory efficiency. Therefore, multiple BSTs are used in the PMM architecture.

The block diagram of the basic pipeline and a single stage of a BST are shown in Figure 8. To take advantage of the dual-port feature offered by SRAM, 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: (1) a pattern $P$, (2) a match vector $MV$, and (3) a pattern label $PL$. In each pipeline stage, there are 4 data fields forwarded from the previous stage: (1) the input string $S_i$, (2) the matching status vector $MSV$, (3) the memory access address $Addr$, and (4) the matched label $ML$. The forwarded memory address is used to retrieve the pattern and its associated data stored in the local memory. This information 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 6(b), the input string anch is less than the pattern of the root node (car); hence, a 0 is appended to form the address of 0. At level 1, the input string is again less than the node’s pattern beat. Thus, the memory address is 00, which is the address to access the memory in level 2. Note that if the input string were greater, then the memory address would be 01.

There is one comparator in each stage of the pipeline. It compares the input string with the node’s pattern, and uses the node’s match vector ($MV$) to produce the matching status vector ($MSV$). Figure 9 depicts the block diagram of an 8-byte comparator. The inputs include: (1) an input string $S_i$, (2) a pattern $P$, (3) a match vector $MV$, (4) a pattern label $PL$, and (5) a match label. The input string $S_i$ and the pattern $P$ go into the “byte comparator”, which performs byte-wise comparisons of the 2 inputs. The results ($M_i - M_0$) are fed into the “matching vector decoder”, which operates based on the

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Prefix label</th>
<th>Suffix label</th>
<th>Matched vector</th>
<th>New prefix label</th>
<th>Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0000</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>12</td>
<td>0000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>10</td>
<td>0000</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>9</td>
<td>0000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
does not have the match vector, and (3) the match label field is no longer required. The matching operation is in fact much simpler, as it does not have to handle the overlapping problem. Additionally, the size of the search key in each node is substantially shorter, making the comparison much faster. In our design, a complete BST architecture and its basic pipeline stage are illustrated in Figure 10.

Let $N_l$ denote the number of entries in the LMM table. The latency of LMM is $\lceil \log_2(N_l + 1) \rceil$. As mentioned before, the size of the input window $L$ must be greater than or equal to latency of LMM, or $L \geq \lceil \log_2(N_l + 1) \rceil$. The impact of $L$ on the number of entries of the LMM and the overall memory efficiency are presented in detail in Section 8.2.

7.4 Dictionary Update

Dictionary update includes two operations: (1) pattern deletion and (2) new pattern insertion. The first type of update requires deleting an existing pattern. Each pattern can include (a) more than one pattern and (b) only one pattern. In case (a), deletion can be done by resetting the bit $i$ of the match vector of the processed pattern, where $i$ is the length of the to-be-deleted pattern. For instance, consider the pattern andy with match vector 0111. To delete pattern andy, the 3rd bit is set to 0, and the match vector becomes 0101. In case (b), there are two possible methods: lazy deletion and complete deletion. The lazy approach can be performed using the same updating mechanism described in case (a). In the complete deletion, if the structure of the tree changes, the BST must be rebuilt, and the entire memory content of each pipeline stage must be reloaded.

In the second type of update, new patterns are inserted into the existing dictionary. If the new pattern has a parent pattern, then only the parent match vector needs to be updated to include the new pattern. Note that all

7.3 Label Matching Module (LMM) Architecture

The architecture of the label matching module is almost identical to that of the pattern matching module. The differences are (1) there is only one BST, (2) each entry

truth table shown in Figure 9 (b). The output of the decoder is AND-ed with the node’s match vector. The result is then AND-ed with the output of the “string comparator”, which compares the pattern label and the match label to produce an 8-bit matching vector (MSV).

Fig. 9: Block diagram of an 8-byte comparator and the operation of its matching vector decoder.

<table>
<thead>
<tr>
<th>$M_7$</th>
<th>$M_6$</th>
<th>$M_5$</th>
<th>$M_4$</th>
<th>$M_3$</th>
<th>$M_2$</th>
<th>$M_1$</th>
<th>$M_0$</th>
<th>Matching Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>10000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>11000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>11100000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>11110000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>11111000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>11111100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111111</td>
</tr>
</tbody>
</table>

(a) Block diagram of a 8-byte comparator.

(b) Operation table of the 8-bit matching vector decoder.

Fig. 8: Architecture of the pattern matching module and its basic pipeline stage.

Fig. 10: Architecture of the label matching module and its basic pipeline stage.
8.2 Memory Efficiency

Memory efficiency of the proposed arbitrary-length string matching algorithm is evaluated. The number of entries in the label matching table for various values of segment length is shown in Table 10.

As mentioned in Section 7.1, length $L$ of the segment needs to be greater than or equal to the matching latency of the label matching module (LMM). In our design, a complete BST is used to implement the LMM. Therefore, the number of entries in the label matching table should not exceed $2^L - 1$. Hence, 3 values of $L$ (16, 20, 24) are used in our analysis. For each value of $L$, the pattern matching module (PMM) is implemented using the cascading BST approach with various values of $L_S \in \{4, 8, 12, 16\}$. Table 8 shows: (1) the memory requirement (in bits) of the pattern matching module ($M_{PMM}$), (2) the memory requirement (in bits) of the label matching module ($M_{LMM}$), (3) the overall memory efficiency ($M_{eff}$) of the entire design for each combination of $L$ and $L_S$. When $L = 24$ and $L_S = 4$, the best memory efficiency is achieved for both Rogets (0.56 byte/char) and Snort (1.32 byte/char). Therefore, the maximum size of supported dictionaries mainly depends on the amount on-chip + off-chip memories. The largest Rogets and Snort dictionaries require only 100KB and 300KB of on-chip memory, respectively, compared with over 4MB of on-chip memory available in a state-of-the-art FPGA device. Therefore, the proposed architecture should suffice for practical implementations of NiDs. Note that the Rogets dictionary has better memory efficiency due to the smaller size of the alphabet (26 in the Rogets and 256 in the Snort).
TABLE 8: Memory efficiency for the Rogets and Snort dictionaries with various values of segment length

<table>
<thead>
<tr>
<th>Segment length (B)</th>
<th>L = 16</th>
<th>L = 20</th>
<th>L = 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>161664</td>
<td>1692069</td>
<td>178988</td>
</tr>
<tr>
<td>8</td>
<td>2101728</td>
<td>2165246</td>
<td>2249280</td>
</tr>
<tr>
<td>12</td>
<td>21463490</td>
<td>21869550</td>
<td>22354990</td>
</tr>
<tr>
<td>16</td>
<td>2354990</td>
<td>2462830</td>
<td>2562465</td>
</tr>
</tbody>
</table>

8.3 Throughput

FPGA Implementation: The proposed pattern matching core was implemented in Verilog, using Xilinx ISE 12.4, with Virtex-5 FX200T as the target. The implementation results for different pattern lengths are reported in Table 11. In all the cases, the amount of logic resource is very low (less than 15% 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.

TABLE 11: Implementation results for different segment lengths

<table>
<thead>
<tr>
<th>Segment length (B)</th>
<th>Frequency (MHz)</th>
<th>Throughput (Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>270</td>
<td>3.5</td>
</tr>
<tr>
<td>8</td>
<td>1518986</td>
<td>2.12</td>
</tr>
<tr>
<td>12</td>
<td>2180955</td>
<td>1.70</td>
</tr>
<tr>
<td>16</td>
<td>2818626</td>
<td>1.90</td>
</tr>
<tr>
<td>20</td>
<td>3454990</td>
<td>2.69</td>
</tr>
</tbody>
</table>

ASIC Implementation: Implementation in ASIC is quite simple. Given a dictionary, the number of pipeline stages and the memory size in each stage can be calculated. Hence, an ASIC chip can be fabricated and is ready to be used. However, since the ASIC chip is not reconfigurable, future expansion of the dictionaries must be taken into consideration when designing the chip. Due to the simple architecture, our designs are memory-bound. Therefore, the amount of memory requirement determines the clock rate of the chip. The memory size of the largest stage is at most 600 Kbit. CACTI 5.3 [6] was used to estimate the memory access time on ASIC. A 600 Kbit dual-ported SRAM using 45 nm technology needs 0.657 ns to access. The maximum clock rate of the above architectures in ASIC implementation can be 1.52 GHz. This clock rate translates to 24 Gbps for a dual-pipeline architecture.

8.4 Performance Comparison

Two key comparisons were performed with respect to the memory efficiency and throughput. While memory efficiency affects the size of the largest supported dictionaries, throughput determines the processing rate of the entire design. For memory efficiency, smaller values indicate better design, whereas for throughput, larger values indicate better design. Table 12 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 [18] 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 the candidates that were used in the comparisons were collected from [18]. These state-of-the-art designs were implemented using the same target FPGA device for fair comparisons. 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. Therefore, 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 4× improvement in memory efficiency, compared with the state-of-the-art designs (the field-merge [18]). Our architecture requires only 1/4 (for both Rogets and Snort) the memory of the field-merge, while achieving 1.4× the per-stream throughput. Compared with the bit-split design, our approach needs only 1/27 (for Rogets) and 1/25 (for Snort) the memory of the bit-split.
design while achieving $1.8 \times$ the per-stream throughput. Additionally, the performance (memory efficiency and throughput) of our algorithm and architecture do not depend on the size of the set of symbols (or alphabets). In other designs, when the set of symbols increases, their performance degrades significantly.

9 CONCLUDING REMARKS

In this paper, we have described a leaf-attaching algorithm that can disjoint a given set of patterns without increasing the number of patterns. We have also presented fixed-length and arbitrary-length matching algorithms for string matching. These algorithms achieved better memory efficiency compared with that of the state-of-the-arts. One of the main drawbacks of our design is that the label matching module needs to match the timing of the input window. This limits the flexibility of choosing shorter segment lengths. Another drawback is the reloading of memory content if the update changes the structure of the BSTs. For future work, we plan to improve our algorithm to match multiple characters per clock cycle and examine at various techniques to reduce power consumption. We would also like to find alternative architectures for the label matching module to alleviate the above timing restriction.

ACKNOWLEDGMENT

The authors would like to thank Edward Yi-Hua Yang for his valuable inputs.

REFERENCES


