STRING LOOKUP USING THREE-TREE STRUCTURE

Inventors: Robert Wing Pong Luk, Hong Kong (CN); Wai Tak Wong, Hong Kong (CN)

Assignee: The Hong Kong Polytechnic University, Kowloon (HK)

(51) Int. Cl.
G06F 17/30 (2006.01)

Abstract
A tree data structure for storing strings of searchable data divided into substrings has at least one first type node that has a substring associated with it and at least one second type node that does not have a substring associated with it. A second type node is directly linked to a first type node by a fail transition. A first type node is directly linked to a second type node by a match transition. An IP address lookup system has memory for storing a tree data structure and a controller for storing and loading an incoming IP address. The system has a finite state machine having three stages including a first stage for loading the IP address and performing a length calculation, a second stage for searching the data structure to find a match, and a third stage for determining the next-hop address index.
### References Cited

**U.S. PATENT DOCUMENTS**

<table>
<thead>
<tr>
<th>Patent Number</th>
<th>Year</th>
<th>Inventor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,103,597 B2</td>
<td>2006</td>
<td>McGovern</td>
</tr>
<tr>
<td>7,155,427 B1</td>
<td>2006</td>
<td>Prothia et al.</td>
</tr>
<tr>
<td>7,359,860 B1</td>
<td>2008</td>
<td>Greenblatt</td>
</tr>
<tr>
<td>7,716,240 B2</td>
<td>2010</td>
<td>Lim</td>
</tr>
<tr>
<td>7,797,348 B2</td>
<td>2010</td>
<td>Luk et al.</td>
</tr>
<tr>
<td>8,150,815 B2</td>
<td>2012</td>
<td>Lim</td>
</tr>
<tr>
<td>8,156,566 B2</td>
<td>2012</td>
<td>Lim</td>
</tr>
<tr>
<td>2002/021838 A1</td>
<td>2002</td>
<td>Richardson et al.</td>
</tr>
<tr>
<td>2002/006101 A1</td>
<td>2002</td>
<td>Wan</td>
</tr>
<tr>
<td>2004/0367732 A1</td>
<td>2004</td>
<td>Luk et al.</td>
</tr>
<tr>
<td>2007/0156559 A1</td>
<td>2007</td>
<td>Lim</td>
</tr>
<tr>
<td>2007/0156560 A1</td>
<td>2007</td>
<td>Lim</td>
</tr>
<tr>
<td>2007/0156694 A1</td>
<td>2007</td>
<td>Lim</td>
</tr>
<tr>
<td>2007/0156695 A1</td>
<td>2007</td>
<td>Lim</td>
</tr>
<tr>
<td>2007/0156696 A1</td>
<td>2007</td>
<td>Lim</td>
</tr>
<tr>
<td>2007/0157287 A1</td>
<td>2007</td>
<td>Lim</td>
</tr>
</tbody>
</table>

---

**OTHER PUBLICATIONS**


* cited by examiner
Figure 1

(a)

Figure 2

root node 011 d
match e
fail
Tree Node

Tree Node

000 a
match
1

110 f

Figure 3

root node 011 d
match e
fail
Tree Node

Tree Node

101

0
Figure 10

Figure 11
<table>
<thead>
<tr>
<th>State</th>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Reset: update begins</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1-4: Start Searching (Algorithm B)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4: Appropriate Initialization</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6-9: Wait for search to halt (Figure 10)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>10-16: If IPFinish or ExactEnd then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If IPFinish then PrefixPtr ← hop register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elseif ExactEnd then Hop ← hop register</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>17-22: If RAMAdr = 0 then root ← NewNodeAdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Elseif PrefixPtr = NULL then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PrefixPtr ← NewNodeAdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Elseif P[i] = 0 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZeroPtr ← NewNodeAdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i ← i + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Elseif P[i] = 1 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OnePtr ← NewNodeAdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i ← i + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAMAdr ← NewNodeAdr</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>23-25: If IPProcess then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StrideChar ←</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Len ← k;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hop ← hop register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PrefixPtr ← NewAdr;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAMAdr ← NewAdr;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NewAdr ← NewAdr + 1;</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>End</td>
</tr>
</tbody>
</table>

Figure 12
<table>
<thead>
<tr>
<th>State</th>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Reset delete</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1-4 Start Searching (Algorithm B) Appropriate Initialization</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4 Wait for search to halt (Figure 10) Maintain previous hop (phop) and stack</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>If IP not Finish and not ExactEnd then Goto state 3 and Halt</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9 If (IP Finish or ExactEnd) and not leave node then Hop ← phop</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>10 If (IP Finish or ExactEnd) and leave node then Hop ← phop</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11-13 If RAMAdr ≠ 0 and ZeroPtr ≠ 0 and OnePtr ≠ 0 and Hop = hregister then OldReg ← RAMAdr; FreeNode(RAMAdr); RAMAdr ← TopOfStack; TopOfStack ← TopOfStack - 1;</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>14-19 Else If ZeroPtr = OldReg then RAMAdr.ZeroPtr ← 0; Else If OnePtr = OldReg then RAMAdr.OnePtr ← 0; Else If PrefixPtr = OldReg then RAMAdr.PrefixPtr ← 0;</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Halt</td>
</tr>
</tbody>
</table>

Figure 13
STRING LOOKUP USING THREE-TRANSITION TREE STRUCTURE

CROSS-REFERENCES TO RELATED APPLICATIONS

The present application is a continuation application of U.S. patent application Ser. No. 12/869,332, filed Aug. 26, 2010, entitled "SYSTEM FOR IP ADDRESS LOOKUP USING SUBSTRING AND PREFIX MATCHING," which is a divisional application of U.S. patent application Ser. No. 10/836,577, filed Apr. 30, 2004, entitled "DATA STRUCTURE AND SYSTEM FOR IP ADDRESS LOOKUP AND IP ADDRESS LOOKUP SYSTEM," which claims priority to U.S. Provisional Application No. 60/467,223, filed May 1, 2003, entitled "DATA STRUCTURES FOR IP ADDRESS LOOKUP AND IP ADDRESS LOOKUP SYSTEM." The applications are hereby incorporated by reference in their entirety for all purposes except for those sections, if any, that are inconsistent with the specification.

BACKGROUND TO THE INVENTION

1. Field of the Invention

The current invention relates to data structures from use in IP address lookup systems and to IP address lookup systems.

2. Background Information

In IP forwarding a lookup is needed for every data packet received by the router. The router uses the lookup to identify a next-hop address and exit port in a routing table. IP forwarding speed is becoming a major problem with increasing demand on the Internet due to multiple protocols converging, faster and more affordable data links, increased traffic due to more users and novel applications such as SMS and multimedia. The problem is further compounded by the use of the classless inter-domain routing (CIDR) address assignment scheme, and the introduction of IP version 6, in which IP addresses can be as long as 128 bits.

To reduce the demanding requirements of fast IP lookup it has been suggested that additional information such as tags and labels be added to the new version of the IP protocol. The additional information is intended to improve the forwarding rate and to simplify the forwarding process. The wide deployment of these schemes still needs to be tested and backward compatibility with IP version 6 would still demand high performance forwarding machines. Other efforts to reduce the demand requirements of fast IP address lookup include reducing the size of routing tables, since smaller routing tables can enhance lookup speed, and the use of caches.

SUMMARY OF THE INVENTION

It is an object of the present invention to provide a data structure and system for IP address lookup which overcomes or ameliorates the above mentioned disadvantages.

According to a first aspect of the invention there is provided a tree data structure for storing two or more strings of searchable data, said strings being divided into a plurality of substrings, said data structure comprising a plurality of data keys each holding a substring, a plurality of nodes including at least one first type node having a substring associated with it and at least one second type node not having a substring associated with it, a plurality of transitions including at least one fail transition, and wherein the second type node is directly linked to the first type node by the fail transition.

According to a second aspect of the invention there is provided a method of searching a tree structure having at least two levels for storing two or more strings of searchable data, the strings being divided into a plurality of sub strings, the data structure comprising at least one first level node having a first field for storing a substring, at least three second level nodes and at least three transitions including a match transition, a first fail transition and a second fail transition, wherein the second level nodes are directly linked to the first level node by the three transitions, said method including comparing the key to the substring and when a match is found transitioning the match transition to the second level of the tree, and when a match is not found comparing a character of the key with the first and second fail transitions and when the character matches the first fail transition transitioning the first fail transition to the next level of the tree and when the character matches the second fail transition transitioning the second transition to the next level of the tree.

According to a third aspect of the invention there is provided an IP address lookup system comprising memory for storing a searchable data structure relating an IP address to a next-hop address, a controller for storing and looking up an IP address, a finite state machine having three stages including a first stage for looking the IP address and performing a length calculation, a second stage for searching the data structure to find a match, and a third stage for determining the next-hop address index, and a buffer to hold the next hop address index.

Further aspects of the invention will become apparent from the following description which is given by way of example only.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will now be described with reference to the accompanying drawings in which:

FIG. 1 illustrates a Trie and its corresponding child-sibling representation of the IP addresses 110 and 100;
FIG. 2 is a Binary hybrid search tree for the IP addresses 110 and 100;
FIG. 3 is a Hybrid search tree for the IP addresses 110 and 100;
FIG. 4 is a ternary hybrid search tree for the IP addresses 110 and 100;
FIG. 5 illustrates path packing for the ternary hybrid search tree in FIG. 5;
FIG. 6 is an Algorithmic State Machine (ASM) Chart for a search process;
FIG. 7 illustrates memory organization;
FIG. 8 illustrates Combinational logic for parallel matching between two characters;
FIG. 9 illustrates Hardware implementation of a ternary hybrid search tree;
FIG. 10 illustrates Cascade architecture;
FIG. 11 illustrates MIMD architecture;
FIG. 12 is an ASM Chart for inserting a single IP address into an existing ternary hybrid search tree, and
FIG. 13 is an ASM Chart for deleting a single IP address in the ternary hybrid search tree.

DESCRIPTION OF THE PREFERRED EXAMPLE

An IP forwarding system comprises a customized SRAM circuit design for IP forwarding. The search function is implemented as a finite state machine (FSM) with less than 10 states and the insertion function is carried out using less than 20 states.

Simple uniform data structures are used that add little logic circuitry to the standard SRAM circuits. This allows easy verification, fast design turn around time and customization.
to both static and dynamic router requirements. Before the hardware
design is described the data types and data structures
used are discussed.
String Abstract Data Type
Applicants earlier U.S. patent application Ser. No. 10/112,
043, the contents of which are considered included herein,
discloses an abstract data type called string that can repre-
sent a string in a storage efficient manner. Sprising is a string
over an infinite triple alphabet $A = \{\text{\Sigma}, \{x\}, \{y\}\}$, where
$\text{\Sigma}$ is a finite alphabet, $\text{\#}$ is a special character outside $\text{\Sigma}$, $x$ is the
null character and $y$ is the set of integers. A character in $\text{\Sigma}$ is
represented as a tuple $(a, p)$ where $a$ is a character in $\text{\Sigma}\setminus\{\text{\#}\}$
and $p$ is called the skip pointer. A trie is linearized into a string (Sprising) $s$ with skip pointers augmented at relevant posi-
tions in $s$ to skip certain part of the sub-strings in $s$ during search and insertion of keywords.
IP addresses can be represented in the string. An IP address,
say 110, is copied to an empty string $s$ resulting in: $s\leftarrow(1.0.0.1)$
where the second value in each group is the skip pointer. If the value of the skip pointer is zero, the skip
pointer is omitted. In this case, $s\leftarrow110$. To insert a second IP
address, say 100, in to $s$, searching is carried out between $s$ and
the second IP address. The search starts at the beginning of
$s$ and matches the first character 1. However, the second
character of s and that of the IP address do not match. Since a
mismatch is found, the remainder of the IP address (i.e. 00)
is appended to $s$ and the skip pointer at the mismatch position
(i.e. 2) is updated with the original length of $s$. Hence, the
updated Sprising $s$ becomes: $s\leftarrow(1.1.5.0.0.0.0.0)$, where 5 is the
value of the skip pointer.
A trie representation of $s$ is shown in FIG. 1. The Sprising
corresponds to the child-sibling representation of the trie,
where the child pointers are erased and the skip pointers are
the sibling pointers. The number of non-zero sibling pointers
is equal to the number of IP addresses represented by the trie
minus one. This represents a significant saving in storage if all
skip pointers with zero values are erased. No intermediate
node in the trie is associated with a character in $s$. The
characters in $s$ are associated with transitions between nodes. The
# character, that separates unmatched IP address suffixes in the
Sprising, corresponds to special trie nodes and leaf nodes,
represented by the double circle, where the prefix to that trie
node represents a complete IP address.
Alphabet Coding
In prefix expansion, multiple bits are packed as an entity or
a character for fast processing. For example, if 2 bits are
packed as a character then the equivalent alphabet size is 4.
The number of packed bits is called the stride $k$.
Prefix expansion is undesirable because of the need to
expand prefixes to the integral powers of the stride. Instead of
expanding the prefixes, a one bit prefix is added to each bit
string to indicate whether the end of the IP address is reached.
If this prefix bit is 0, then all the bits in the string represent the
packed bits of the IP address. Otherwise, the end of the string,
or IP address, is reached and number of packed bits is less
than the stride. In this case, where the number of packed bit is
less than the stride, it is necessary to know the number of bits
packed, as well as the bit pattern. One encoding that can serve
both purposes is to use a $2^k$ suffix appended to the packed
bits, where $2^k$ means zero or more zero-value digits such that
the total length is equal to the stride. If the prefix bit is not 0
then to detect the length, the bits are scanned backward until
1 is found and the rest of the bit patterns are the packed bits.
Alternatively, the suffix coding could use $0^k$ suffix, where $1^k$
means non-zero or more nonzero value digits such that the
total length is equal to the stride. We call this stride+$1$ coding
because it includes the stride bits plus a one bit prefix.

Let $p$ be a bit string of length stride plus 1 (i.e. $k+1$). The
first character in $p$ is the delimiter bit (i.e. $p[1]$). If $p[1]=0$,
then $p[2\ldots k+1]$ are the $k$ packed bits. If $p[1]=1$ and the
number of packed bits is $n$, which is less than $k$, then $p[2\ldots 
(n+1)]$ are the $n$ packed bits, $p[n+2] = 1$ and $p[n+3\ldots 
(k+1)] = 0$ (if $n+2<k$). The two IP addresses 110 and 100 packed using stride+$1$ coding are (011) (101) and (010) (101) respectively,
where the string between brackets is a stride+$1$ character. The
first bit is the prefix, 0 or 1, indicating whether all the bits in
the string represent the packed bits of the IP address and the
underlined bits are the packed bits.
Hybrid Search Tree
Sprising and alphabet coding can be used together to rep-
represent any trie. For example, the two IP addresses 110 and 100
are represented by the following Sprising: $s\leftarrow(3, 3, e, a, c, e$;
where $\Sigma_2\{a, b, c, d, e, f\}, a=000, b=001, c=-010, d=011,
e=101, f=110$.
For Sprising with stride+$1$ coding, there is no need for the \# character since the prefix bit is used to indicate the end of the
IP address. A disadvantage of using stride+$1$ coding with
Sprising is that the number of traversals is increased substan-
tially. In the worst case, all positions that are directly pointed
to by the skip pointer or pointed to via any number of indi-
rection of a sequence of skip pointers have to be visited, which
is not desirable.
To overcome this problem an alternating hybrid search tree
is used. Under some conditions the search tree takes advan-
tage of the stride+$1$ coding and in other conditions the search
tree is like a conventional trie for efficiently finding the long-
est prefix match. A simple hybrid search tree is a binary search
tree. When stride+$1$ coding is used, the tree node tests
whether the incoming IP address prefix is the same as
the stride+$1$ character stored in the tree node. If not, the failure
branch of that tree node is traversed and matching reverts to
conventional trie searching for the following trie node. If the
match is successful, the success branch is traversed, arriving
at another tree node. The root node is always a tree node since
a trie node is visited only after a failure to match in the parent
tree node.

FIG. 2 shows the binary hybrid search tree for the two IP
addresses 110 and 100 using the alphabet $\Sigma_2$. Since this
hybrid search tree is a trie it can be represented as a string
$s\leftarrow(3, 3, e, a, f)$; where the skip pointer represents the fail
branch of a tree node or the sibling pointer of a trie node.
For the best case, the number of traversal for a successful
search is reduced to $|IP/k|$ where $P$ is the keyword and $k$ is the
stride. For the worst case, since each tree node leads to a
match failure and the trie node always matches a successful
match, the number of traversals for a successful search is
2(|IP|+1). For the string representation each trie node requires
at most 2 additional skip pointers therefore the worst case
search time is 3(|IP|+1) traversals. If the number of additional
traversal due to match failure can be reduced, then the worst-
case search time can be reduced further.

With the hybrid search tree an additional traversal occurs for
every failure to match in the trie node. One special con-
ideration is that after each failure to match in the trie node, a
sub-trie of maximum depth $\epsilon$ equals to the stride $k$ follows.
Based on the example in FIG. 2, the hybrid search tree is
shown in FIG. 3. For this example, after the failure to match at
the root node, there are two trie nodes because the maxi-
mum depth $r=2$-$k$. When the maximum depth is reached, the
search reverted back to the tree node with character e. For the
worst-case search time with $r=k$, this case, there are at most
$|IP/k|$ additional traversals and there are |IP| binary trie tra
versals. In general, the worst-case number of traversal is bounded by:

\[ T(r) = \frac{\log m}{1 - \frac{1}{m}} + 2. \]

As \( r \) tends to infinity, the worst-case search time approaches that of the standard binary trie. However, if \( r \) is set arbitrary large, then the number of tree node is reduced and the reduction in storage by packing multiple bits together is also reduced.

A tree node has the following structure:

```
Struct TreeNode begin
    Character StrideChar;
    Integer hop; [Nexthop addr idx]
    Integer len; [Len of StrideChar]
    Pointer LeftPtr;
    Pointer RightPtr;
End
```

The following Algorithm is the search procedure for a binary hybrid search tree of arbitrary depth \( r \).

Method: Search_BHT
Input: stack \( i \), integer \( k \), integer \( w \)
Output: stack \( 
\begin{align*}
1: & \quad \text{integer } i \leftarrow 0; \quad \text{[Initialize]} \quad \\
2: & \quad \text{integer hop } \leftarrow 0; \quad \\
3: & \quad \text{integer len } \leftarrow 0; \quad \\
4: & \quad \text{integer rlen } \leftarrow 0; \quad \\
5: & \quad \text{BTnode } c \leftarrow \text{root;} \quad \\
6: & \quad \text{stack } c \leftarrow \text{NULL;} \quad \\
7: & \quad \text{Bool tree } \leftarrow \text{true;}
\end{align*}
\]

while \( c \neq \text{NULL do} \{ \text{Loop} \}
\]

\begin{align*}
8: & \quad \text{push}(c, 0); \quad \text{[stack delete only]} \\
9: & \quad \text{if } \text{tree then} \{ \text{tree node} \}
\end{align*}

\begin{align*}
10: & \quad \text{if } \text{c.StrideChar } \neq [\text{p} \ldots i+k-1] \text{ then} \\
11: & \quad \quad \text{i } \leftarrow \text{i }+ \text{c.len;}
12: & \quad \quad \text{len } \leftarrow \text{c.len;}
13: & \quad \quad \text{hop } \leftarrow \text{c.hop;}
14: & \quad \quad \text{rlen } \leftarrow \text{c.hop;}
15: & \quad \quad \text{if (rlen } \neq \text{c.i)} \text{return } (\text{o}, \text{hop}, 0); \\
16: & \quad \quad \text{else } c \leftarrow \text{c.LeftPtr;}
17: & \quad \quad \text{else } \{ \text{match fail: goto trie node} \}
18: & \quad \quad \quad \text{c } \leftarrow \text{c.RightPtr;}
19: & \quad \quad \quad \text{tree } \leftarrow \text{false;}
20: & \quad \quad \quad \text{rlen } \leftarrow 0;
21: & \quad \quad \text{else } \{ \text{tree node} \}
22: & \quad \quad \quad \text{rlen } \leftarrow \text{rlen } + 1;
23: & \quad \quad \quad \text{if } \text{rlen } \neq \text{tree then } \text{true;}
24: & \quad \quad \quad \text{if } \text{P[i] } \neq 0 \text{ then} \\
25: & \quad \quad \quad \quad \text{i } \leftarrow \text{i }+ 1;
26: & \quad \quad \quad \text{hop } \leftarrow \text{c.hop;}
27: & \quad \quad \quad \text{c } \leftarrow \text{c.RightPtr;}
28: & \quad \quad \quad \text{if } \text{P[i] } \neq 1 \text{ then} \\
29: & \quad \quad \quad \quad \text{i } \leftarrow \text{i }+ 1;
30: & \quad \quad \quad \text{hop } \leftarrow \text{c.hop;}
31: & \quad \quad \quad \text{c } \leftarrow \text{c.LeftPtr;}
32: & \quad \quad \quad \text{else return } (\text{o}, \text{hop}, 0); \\
33: & \quad \quad \text{return } (\text{o}, \text{hop}, 0)
\end{align*}

A trie node is a TreeNode with a NULL character for StrideChar. The stack \( i \) (Step 6) is used to keep track of the traversed nodes for the delete operation only and is not used for searching or insertion. If the current node is a trie node (Step 10) and the stride+1 character is matched, then Steps 12-14 update the relevant variables. Step 13 decides whether the search reaches the end. If not, the search continues. If the tree node failed to match, then traversal is made to a trie node (Step 17). The variable rlen is used to keep track of the depth or the number of traversals from the root of the sub-trie. Once the depth reaches the given maximum depth \( r \), the next traversed node is assumed to be a tree node (Step 23).

The number of failure to match at tree nodes should be at most \( N-1 \), where \( N \) is the number of IP addresses in the database. For each match failure the number of trie nodes is at most \( r \) and the number of trie nodes are at most \( (N-1) \). The number of tree nodes are at most \( (T-N)k+N-1 \) since each tree node represents \( k \) original trie nodes and the number of trie nodes is \( T \). Hence, the total number of nodes is bounded by:

\[ S_r = \frac{T - N}{k} + N - 1 + (N - 1)r \]

Hence, \( r \) should be minimized for storage and maximized for speed.

Ternary Hybrid Search Tree

The trie node of a binary hybrid tree is merged with the parent tree node of the binary hybrid tree to form a ternary tree node. FIG. 4 is a ternary hybrid search tree for the two IP addresses 110 and 100. Each node in the tree has 3 pointers and there are no trie nodes. Each node holds a stride+1 character. If the incoming stride+1 character matches with the current node, then the middle pointer, representing a successful search is traversed. Otherwise, if the current bit is on, then the left pointer is traversed. If the current bit is zero, then the right pointer is traversed. The current bit in this case is the most significant bit of the incoming stride+1 character. If this bit is a 1 the left (1) transition is followed to the next level of the tree and the bit is a 0 the right (0) transition is followed to the next level of the tree. The advantage of the ternary hybrid search tree is in saving the trie node of the hybrid search tree. Although each tree node has three pointers instead of two, the total number of nodes to represent the same set of keywords as the binary hybrid search tree may be less.

Spirstring is also used to represent ternary hybrid search trees. The (middle) pointer representing a match with the stride+1 character of the current tree node is implicitly represented by adjacent positions in the spirstring where position \( i \) represent the current tree node \( c \) and position \( i+1 \) represents the tree node, which is the child node of \( c \) after following the match or middle pointer of \( c \). The skip pointer at position \( i \) is the sibling pointer of the tree node \( c \). For example, the spirstring representation of the ternary hybrid search tree in FIG. 4 is:

\[ S=(d,3,e,a,g) \]

The worst-case search time is \( 3|P|+3 \) traversals and this turns out to be the same as the spirstring representation of the binary hybrid search tree with \( r-1 \).

The ternary hybrid search tree has the following data structure:

```
Struct TreeNode begin
    Character StrideChar;
    Integer hop; [Nexthop addr idx]
    Integer len; [Len of StrideChar]
    Pointer PrefixPtr;
    Pointer ZeroPtr;
    Pointer OnePtr;
End
```

The difference between the TreeNode data structure and the earlier BTreeNode data structure is the addition of the pointer PrefixPtr. The variable hop stores the next-hop
address of an IP address that reaches the current node. The delimiter bit of the strife+1 character is on the next-hop address is stored in PrefixPtr. There should not be any child nodes via PrefixPtr for that tree node.

The following algorithm finds the longest prefix match in the ternary hybrid search tree for the input string P.

```
Method: Search_THIT
Input: string P, integer k
Output: stack t, integer h, integer w
1: integer i影0; [Initialization]
2: integer hop影0;
3: TreeNode c影null;
4: stack o影null; [Loop]
5: while o影null do
6: pushc, o影[for delete only]
7: hop影hop; [Keep track of hop]
8: if c.StrideChar影P[i-i+k-1] then
9: i影i+k;
10: if c.len影k then [exact match]
11: return (o, c.PrefixPtr, 0);
12: else c影c.PrefixPtr;
13: else if P[i影0 then
14: c影c.ZeroPtr;
15: i影i+1;
16: else P[i影1 then
17: c影c.OnePtr;
18: i影i+1;
19: else out]
20: return (o, hop, P[i+k-1]影exact match]
21: return (o, hop, h) [prefix match]
```

The search begins with the root node (Step 4) and repeatedly tries to find the appropriate child node of the current node to traverse. Before the traversal, some housekeeping tasks are carried out (Steps 6 and 7), including maintaining the stack of traversed nodes, and updating the next-hop address index of the longest prefix matched. Step 8 tests whether the strife+1 character matches the input string P at position i. If there is a match, the child node corresponding to a successful strife+1 character matching is traversed via PrefixPtr. The string position i is updated by the length of the strife+1 character, which is stored in c.len. If c.len影k, then the delimiter bit of the strife+1 character is on and the search ends in success. In this case (Step 11), the stack of the traversed nodes, the next-hop address index stored in c.PrefixPtr and the match flag are returned. If the match flag is zero, then the match is an exact match. If the match flag is P[i影1, then it is also an exact match but the next-hop address index is stored in the hop variable, instead of the PrefixPtr. Otherwise, it is a prefix match.

If the strife+1 character does not match with the input string P at position i, then Search_THIT examines whether P[i影zero (step 13). If this is true, then the c.ZeroPtr is traversed and the string position i is advanced by unity. Likewise, if P[i影1, then c.OnePtr is traversed and i is incremented. Otherwise, the end of the input string P is reached. In this case, the stack, next-hop address index and the successful match flag are returned (Step 20).

The following algorithm inserts one IP address into the ternary hybrid search tree.

```
Method: Insert_THIT
Input: string P, integer k
Output: NULL
1: integer i;
2: integer hop;
3: stack t;
4: (i, hop, i)影Search_THIT(P, k);
```

Initially, Search_THIT( ) is called to determine whether the IP address exists in the ternary hybrid search tree. If there is exactly the same IP address in the ternary hybrid search tree, then the next-hop address index is updated (Steps 7 and 8). Otherwise, a prefix match is found and a node is created (Step 10). If the current node is null, then the ternary hybrid search tree must have been the empty tree or null. Hence, the root node is updated (Step 11). If the current node has a non-null prefix pointer, then either the OnePtr or the ZeroPtr of the current node is null and needs to be updated (Steps 13 to 15).

Since the traversal of the pointer to the child node consumes one bit of the IP address, i is incremented (Step 15). Otherwise, the PrefixPtr is updated (Step 16). Steps 17 to 22 repeatedly append a strife+1 character to the node o. Steps 23 to 25 add a strife+1 character to the last node. It is assumed that CreateNode( ) will initialize the pointers of the new node to null and MakeChar( ) will automatically create a strife+1 character based on k or less than k bits in the IP address.

The following algorithm deletes one IP address from the ternary hybrid search tree.

```
Method: Delete_THIT
Input: string P, integer k
Output: NULL
1: integer i;
2: integer hop;
3: stack t;
4: TreeNode o;
5: (i, hop, i)影Search_THIT(P, k);
6: TreeNode c影pop();
7: integer i影pop();
8: integer hop影pop();
9: if i影1 then c.PrefixPtr影null;
10: else if c.OnePtr影null then c.ZeroPtr影null;
11: else c.ZeroPtr影null;
12: else c.OnePtr影null;
13: if c影null then
14: return;
node is not a leaf node, then only the hop address indexes are
updated (Step 8 and 9) and the current node c is not deleted.
Otherwise, backtracking of the ternary hybrid search tree is
implemented as popping the stack t, until the node has a
non-null Zeroptr or a non-null OnePtr or the next-hop address
index is not the same as the next-hop address index of
the deleted IP address. Finally (Step 15-19), the child pointer
of the undeleted node is reset to NULL.
Path Packing in Pointers
FIG. 5 shows the path packed ternary hybrid search tree of
FIG. 4. In path packing, a sequence of nodes of out-degree is
stored as a string in unused pointers of a node. For example,
the packed bits of a leaf node are stored in the pointer field of
its parent node instead of the pointer to the leaf node. In FIG.
5 path packing is used in the bottom right node, where the path
is stored in the OnePtr of the node. For stride+1 character, the
prefix bit can be used to indicate the last character of the path.
Path packing is a general technique that can be used for any
(sparse) tree structure.
Two type of path packing are used. If the path to be packed
requires storage of more than two pointers plus a stride+1
character, then the PrefixPtr can be used to store a pointer
pointing to the next node, and the stride+1 character, OnePtr
and ZeroPtr of the node are used to store the path. Alterna-
tively, the path can be stored in a single node using the
stride+1 character, PrefixPtr, ZeroPtr and OnePtr.
The variable len of a node can be used to indicate whether
there is any path packing and if there so, what type of path
packing is used. Following Algorithm is an example.

\[
\begin{align*}
\text{if } 0 < \text{len} \leq k & \text{ then no path packing} \\
\text{else if len } \leq k + 2C_1/C_2 & \text{ then} \\
\text{the string is packed in the stride+1 character, the OnePtr and the} \\
\text{ZeroPtr of the current node} \\
\text{else the string is packed in the stride+1 character, the PrefixPtr, the} \\
\text{OnePtr and the ZeroPtr of the current node} \\
\text{where } C_1 \text{ is the storage cost of a pointer and } C_2 \text{ is the storage} \\
\text{cost of a character.}
\end{align*}
\]

Hardware Design
In the first subsection, the basic IP forwarding machine that
implements the search and update algorithms using sparsing
or hybrid search tree ADT is discussed. In various embed-
diments activities of the IP forwarding machine, as well as of
other techniques described herein, may be performed through
execution, by a computing device, of instructions stored on
computer-readable storage media. Only selected components
of the basic forwarding machine are discussed to highlight
the implementation problems and to demonstrate how simple
the machines are.
Basic Forwarding Machine
The search and update algorithms for the sparsing and trie
data structures are implemented using finite state machines
(FSMs). The processing of FSMs can be divided into three
general stages where pipeline techniques can reduce the total
number of clock cycles further. The first stage is to load the
data and perform some pre-processing (e.g. length calcula-
tion). In the second stage searching or updating based on the
sparsing or trie data structure is carried out. The final stage is
to determine the next-hop address for output. The FSMs are
also designed to change state on the negative edge of the clock
and to process the combination logic on the positive edge of
the clock to reduce unwanted transient effect.

FIG. 6 shows the Algorithmic State Machine (ASM) chart
of the search process for the sparsing data structure for a
binary alphabet. In total, there are only 6 states, illustrating
that a very simple FSM can be built with very little circuitry
added to the SRAM. The control signals are SR-A, SR-B, etc.
This ASM chart serves as the basic structure for searching
whether stride+1 coding or hybrid search trees are used.

The search process is divided into three stages which can be
pipelined. Stage 1 (Init) loads the IP address and its length.
Stage 2 is the Spstring search. If the current address is com-
pletely matched (Adr Finish) then the search moves to stage 3.
Otherwise, the ASM will examine whether the current stride
character matches (Match-IP), signalling the Match-IP flag.
If there is a match (i.e., SR-A is true), then the next child
is obtained by incrementing the memory address for Spstring.
If the prefix bit is on or prefix character is found, the Word flag
is set and the current hop address is stored at the Track Adr
state. If the Match-IP is zero and the skip pointer value is non
zero (i.e., not NULL), then the sibling node is visited by
loading the skip pointer value of the Spstring into the address
decoder. In this case, if the delimiter bit is on or the word flag
is on, then the next hop address is updated (during the next sib
state). Otherwise if the skip pointer value is null, then search-
ing is finished and stage 3 begins. In stage 3, the prefix
character is sought, until it is found or until the skip pointer
value is zero (i.e., NULL). If the prefix is found, then the hop
address has to be loaded into the hop register during the load
hop state. If the skip pointer value is zero, then the content
of the hop register is the most updated. Finally, in the hop here
state, the content of the hop register is used as the next-hop
address index in order to lookup the specific next-hop address
for forwarding.

FIG. 7 illustrates the organisation of the Static Random
Access Memory (SRAM). The SRAM is organized into a
used area, an unused area and a released area. The used area
is memory that is currently used by spstring or the tries. It may
contain some freed memory. The unused area is the memory
region that is not used for any purpose. The released area is
the memory region that stores information about memory that is
freed dynamically. The freed memory may be in the used or
unused areas.

Each location in the unused or released area contains a
pointer and a block size. The pointer points to the area where
the memory is free in the used area and the block size indi-
cates the contiguous memory that is free starting from the
pointer location to the pointer location plus the block size.
If the block size is zero, then that addressable space is in the
unused area. The size of the pointer is m and the block size is
W−m, where W is the memory width. The released area can
be organized into a heap if the storage allocation is based on the
largest block of memory.

Initially, when the power is turned on, the size of the release
and used areas is zero and the entire memory space is the
unused area. Hence, the used size pointer is 0 and the released
size pointer is 2m−1. Memory is allocated using the SRAM_
Block_Alloc Algorithm.

Method: SRAM_Block_Alloc
Input: integer m, integer b
Output: integer adr
1: integer adr1 ← FindBestBlock(RP, b, h);
2: if RPtr = 2m−1 OR add = 1
3: if Uptr = b ≤ RPtr
4: adr ← UPtr;
5: Uptr = Uptr + b;
6: else return −1; [No free memory]
7: else
8: if adr1 ≥ 0
9: adr ← SRAM[adr1].Pointer;
The released area is checked to see if a large enough or best-fit contiguous block is available. If there are no deletions, then memory in the unused area is used and the used size pointer is updated (Steps 4 and 5).

This simple memory organization can lead to memory fragmentation for sprring since large enough contiguous blocks of memory may not be available even though some smaller free memory blocks are available. The sprring is rebult if the released area is minus one (Step 6). Since sprring is designed for static solutions and dynamic with insertions, deletions are expected to be seldom and rebuilding the sprring from scratch would be infrequent. For fast insertion, if there is sufficient memory in the unused area for insertion. If the unused area is not large enough, the released area is used and the memory allocation speed would be much slower since the time complexity of SortReleasedArea(RPrr) and RemoveBlock(RPrr, adr) may be linear with respect to the size of the released area.

If there is a memory release, then an extra location in memory is used to point to the block of de-allocated memory. If there is any memory in the released area that is adjacent to the de-allocated memory to form a large contiguous block of released memory, then it is sufficient to update the block size of the adjacent block of memory in the released area.

The memory allocation and de-allocation even for our simple machine is quite complicated. A simpler solution is to allow for insertions and allow for deletions without memory de-allocation. When the memory is not enough, the sprring is rebult completely. However, this simple scheme requires additional storage to hold the address during the rebult process. With a Multiple Instruction Multiple Data (MINTD) architecture with multiple machines, one machine could be used to rebuild the sprring for the other machine. Unfortunately, two machines would be used for rebuilding the sprring instead of one.

The memory organization for trees is much simpler because there is no requirement a block of contiguous memory. This memory organization does not have any memory fragmentation problem if each addressable space can hold one tree node. Since there is no need to find the best match block of memory, memory allocation and de-allocation can be performed efficiently (i.e. in one clock cycle).

When the system is powered on, the memory is organized into a linked list of records of free space. The following SRAM_Trie_Init Algorithm achieves this.

```
Method: SRAM_Trie_Init
Input: integer m
Output: Nil
1: for integer i ← 0 to 2^4−1 do
2: if i = 2m − 1 [Last Free Memory]
```

Each record is associated with a unique address in memory and it has a pointer that points to the next location where free memory is available. When the system is initialized, the pointer points to the current location plus one (Step 6 and 7). When the end of memory is encountered (Step 2) the pointer is assigned the value minus one to indicate that the end of the linked list or stack is encountered with no more free memory available. This initialization is algorithmically simple but it requires 2^4+1 clock cycles to initialize.

An alternative method uses an additional pointer, say UPtr (or the UsedSize pointer) to indicate where the memory is used up to. Initially, the UPtr is initialized to zero and RPrr to minus one. This only requires two clock cycles for initialization (c.f. 2^4+1 clock cycles for the previous method). When allocating memory for a trie node only one address is returned. The following SRAM_Trie_Alloc Algorithm is used to allocate memory a trie node.

```
Method: SRAM_Trie_Alloc
Input: integer m
Output: integer adr
1: if RPrr = 0 [No Released RAM]
2: if UPtr = 2^4−1
3: adr ← −1 [No memory]
4: else
5: adr ← UPtr;
6: UPtr← UPtr + 1;
7: else
8: adr ← RPrr;
9: RPrr ← SRAM[RPrr].Pointer;
10: return adr;
```

If there is any freed memory (i.e., RPrr=0) then freed memory is allocated (Steps 8 and 9). If there is unused memory (i.e., UPtr=2^4−1) then this unused memory is used (Steps 5 and 6). If there is neither freed memory nor any unused memory, then an error value −1 is returned (Step 3).

Since there is no freed memory after initialization the unused memory is used since RPrr is initialized to minus one.

Memory de-allocation is simpler than memory allocation. The following SRAM_Trie_Free Algorithm is used to de-allocate memory from tie node.

```
Method: SRAM_Trie_Free (FreeNode)
Input: integer adr
Output: Nil
1: if SRAM[adr].Pointer ← RPrr;
2: SRAM[adr].BlockSize ← 1;
3: RPrr ← adr;
```

The freed trie node is simply pushed into the stack of freed memory, updating the top of the stack pointer (i.e. RPrr). The amount of available space for the stack of freed memory is limited by the used memory space. Since the amount of freed memory can never be larger than the used memory, there is no need to consider the use of unused memory to maintain the stack of freed memory (i.e., no need to update UPtr).
Clearly, memory initialization, allocation and de-allocation are very efficient to carry out, which requires only one or two clock cycle(s).

The width W of the SRAM memory depends on a number of factors. Firstly, k bits are reserved for strides+1 coding and 1 bit for maintaining the next-hop address. Secondly, an spsring has one pointer and a trie node has two pointers. The pointer size of m bits depends on the addressable space, which in turn depends on the maximum number of IP addresses that needs to be stored. If 256K IP addresses are stored m is at least 18 bits. Further, if every IP address requires 4 address spaces then m should be set to 20 bits. Thirdly, the next-hop address index can be stored in every address space or it can be stored after the delimiter character or after the character where the delimiter bit is on, in which case it occupies an address space. Storing the next-hop address index in every addressable space is preferred if the size h of the next-hop address index is relatively small. In general, this method is used when:

\[
h \leq \frac{NW}{2^m}
\]

where \( N \) is the maximum number of IP addresses. Even if the above becomes an equality it is preferable to assign the next-hop address index to every addressable space because it can reduce the usage of address space and because the next-hop address index can be loaded in parallel with the other information (e.g. pointer and strides+1 character), saving an additional clock.

If \( NW \geq 256K \), h=8 and m=20 then the next-hop address index is assigned to every address space when \( W \geq 32 \) bits. \( W \geq 32 \) bits for every trie node because it has 2 pointers (i.e. 2x20 bits). Hence, each trie node should store a next-hop address index. For spsring the memory width W is \( k+1+4m \). The next-hop address index is assigned to each addressable space if \( k=W-m-1 \).

When strides+1 coding is used the Match-IP decision is implemented based on parallel comparison logic. For strides k bits from the IP address prefix are compared with the k bits in memory. In addition, the delimiter bit is placed at the back (least significant bit) of the IP address prefix and the delimiter bit is placed at the front (most significant bit), which have to be processed as well. There are two outputs from the comparison: the exact match and the prefix match. The exact match requires the bit patterns between the strides+1 character of the IP address prefix and the character in memory to match exactly. The prefix match is the same as the exact match when the delimiter bit is zero. Otherwise, when the delimiter bit is one a match is found if the prefix of strides+1 character in memory is matched with the strides+1 character of the IP address. This prefix match is used for keep track of the next-hop address for the longest prefix match. FIG. 8 shows parallel comparison logic for the stride k=4.

Hardware Implementation of the Hybrid Search Tree

The ASM chart corresponding to the Search_THT Algorithm for the ternary hybrid search tree comprises only two states: state one for initialization, which corresponds to Step 1 to 4 of the Search_THT Algorithm; and state two for looping, which corresponds to Step 5 to 21 of the Search_THT.

FIG. 9 shows the hardware implementation of the Search_THT Algorithm, assuming that the values in the registers are appropriately initialized after state one.

The hardware compares packing multiple bits (called IPChar) from the IP address with the StrideChar in the memory. The result of the comparison indicates whether there is an exact match. If there is an exact match the value of the prefix pointer (PrefixPtr) is the next address and the current IP address position is moved by one stride k. Otherwise, the current IP address position is advanced by one position. Also, the next address depends on whether the current bit of the IP address is a zero or a one, where the value of the ZeroPtr or the OnePtr would be routed to the next address, respectively. The memory lookup terminates when the next address is zero (Step 5 in search_TNT). The index of the next-hop address is stored in the hop register, which is updated either by the hop field of the accessed memory or the PrefixPtr field of the accessed memory, depending on whether the delimiter bit of the StrideChar is on and whether the end of the IP address is reached.

The procedure for inserting an IP address into the ternary hybrid search tree is more complicated. The state transitions and the associated actions of the ASM chart for the Insert_TNT Algorithm are shown in FIG. 12.

The update starts with state zero. The transition from state zero to state one will initiate the search logic, FIG. 9, and initialize different variables for Insert_TNT. The transition from state one to itself is simply a polling mechanism to wait for the search logic to halt.

The state transitions and the associated actions of the ASM chart for the Delete_TNT Algorithm are shown in FIG. 13.

The deletion procedure needs to searches to check whether the IP address is in the ternary hybrid search tree. If it is not the delete operation stops immediately (state 1 to state 3 transition). Otherwise, the previous next-hop address is stored in the last node of the search traversal (state 1 to 2 transition). Next, the current last node of the search traversal is deleted (state 2 self-transition) until the current last node is the root node, or has a non-null ZeroPtr or non-null OnePtr or has a different next-hop address value.

Level Folding

Level compression has been applied in LC-tries and in hardware for IP forwarding to speed up the search process. However, if level compression is applied to different trie nodes as well as the root node memory management becomes complex. For this reason level compression is only applied at the root node. In addition, instead of enumerating all the levels from 0 to d, as in level compression, contiguous memory is only allocated for certain levels, e.g. level \( d_1 \) and level \( d_2 \) only. This can save half the storage cost for each level and it serves the purpose of bounding the search time to \( d_2 - d_1 \) (assuming that \( d_2 > d_1 \)). This modified technique is called level folding to distinguish it from level compression. Another advantage of the level folding technique is that update is simplified because there is no need to expand prefixes. Instead only the highest level is identified for update.

Each entry at a particular level \( d \) in level folding corresponds to an IP address prefix bit pattern of length \( d \). In this way, there is no need to implement any matching logic. Instead, the first \( d \) bits of the IP address are used as a memory address to lookup information relating to the entry. Each entry contains a pointer to the memory where the search or update process begins and a next-hop address index for the first \( d \)-bits matched. Assuming that the next-hop address index needs 8 bits and the pointer needs 20 bits, each entry requires 28 bits to store information.

Due to the concentration of IP addresses of length 8, 16 and 24 \( d_i \) is set to 8 and \( d_{i+1} \) is set to 16 for the hardware design. IP addresses of length 24 are too large to be accommodated in the chip, which originally only holds 32 Mbytes of RAM. Specifically, 24 bits implies 16 million address spaces. Assuming each entry needs 28 bits, the total storage needed is
56 Mbytes. Even if each entry contains only the next-hop address index, the level folding technique would consume 16 Mbytes, with 16 Mbytes left for the other machines. This implies about half of the number of individual forwarding machines can be accommodated and the search speed would be halved.

Instead of enumerating all the d-bit patterns of the dth level folding, which requires $2^d$ address space, only the most commonly used d-bits IP address prefixes are stored and used in a binary content addressable memory (CAM) cache. This is dynamic level folding, as compared to the previous static level folding. It is effective in two respects. Firstly, caching the IP addresses in the database is more effective than caching the incoming IP addresses. And secondly, using a standard binary CAM to cache IP address prefixes of fixed length d enables caching of a group of IP addresses that share the same prefix of length d, reducing the likelihood of a miss even though the incoming IP addresses may differ as long as the address prefix has the same d bits. The use of a binary CAM also saves half the number of memory cells and reduces gate counts for comparison for each entry. The net effect is that a larger size binary CAM can be used, increasing the likelihood of a cache hit, or alternatively more levels of dynamic level folding can be supported.

As levels become higher and higher, the number of IP addresses of longer length becomes less and less, and the required cache size can be reduced further and further. Dynamic level folding at level 20, 24 and 28 can be implemented using a 1 bit binary CAM for each level. When the cache is empty and an IP address of length d or longer is inserted the CAM for the dynamic level folding at the dth level is updated. Hence, if the number of IP addresses of length larger than or equal to d is less than the size of the CAM dynamic level folding is emulating static level folding. This is because all IP addresses of length d or longer are stored in the CAM. For every long IP addresses this mode of operation is the norm rather than exception as the number of long IP addresses is small. If there is more than 1 level of dynamic level folding the deepest level that is shorter than or equal to the length of the inserted IP address is updated.

The number of bits in a cache line is the number m of bits for memory addresses plus the number h of bits of the next-hop address index (i.e. m+h). Typically, d is at most 32 for IP version 4 (IPv4) addresses, m is 20 and h is 8. Thus, the cache line for IPv4 addresses is just 28 bits, with at least 16 tag bits and at most 32 tag bits.

Pipeline Architecture

The pipeline/cascade architecture enables simple algorithms to operate at 1 clock cycle per IP address lookup in a uniform manner. It is also scalable to longer prefixes by cascading more stages without affecting the throughput. However, not every search algorithm can be put into pipeline or cascade architecture and two additional problems need to be resolved. The first problem is the amount of memory to allocate at each stage of the pipeline and the second problem is the priority to output the results.

FIG. 10 shows the cascade (or asynchronous pipeline) architecture for IP forwarding based on the FSM design. The IPM (In-Path Module) has a buffer that stores the incoming IP address and the (dynamic or static) level folding is executed in this module. The IPM holder partitions the incoming IP address into IP suffix the retrieves the address (start address) of the root node at which the FSM is to start the search. For each stage of the cascade the state of processing is passed to the next stage, with the unprocessed IP suffix, the next node address (called StartAdr) in the next FSM module and the remaining length of the IP address.

A potential problem with cascade architecture arises in passing this amount of information from one stage to the other. For IPv4, IP Suffix length can be as long as 32 bits, requiring 5 bits for the length information. If the address space is specified in 20 bits and 6 bits are used to represent different states, the total number of bits is 63 bits. For IP version 6 (IPv6) the number of bits to pass from one stage to the next is 162 bits. It is unlikely that such a large internal bus is used inside a chip. If several chips are placed in a cascade, then 162 pins are needed if the data is transferred in one external clock. For each FSM, there is a buffer to hold the next-hop address index. Assuming that the next-hop buffer can pass the next-hop address index in parallel with a 32 bit bus, it is possible to load 4 next-hop address index simultaneously if each next-hop address index needs 8 bits.

For trie based solutions, if each stage corresponds to the depth of the trie data structure then by the prefix property there are at most Ntrie nodes if the chip can hold at most NIIP addresses. Furthermore, the root trie node is at depth 0 the number of trie nodes is limited by the depth d to 2^d. Hence, at any stage d, the number Td of trie nodes is bounded as follows:

$$T_d \leq \min(2^d, N)$$

For a standard binary trie, assuming that N = 256K, each trie node has 2 pointers plus the next-hop address index. Assuming that each pointer uses 18 bits and the next-hop address index uses 8 bits, a trie node needs 44 bits, which can easily be contained within 6 bytes. From stage to stage, the total number of nodes is:

$$18 + 4 \times 44 = 256$$

Hence, the storage from stage 0 to 17 is 256K x 6 bytes = 1.5 Mbytes. Thereafter, each stage needs to maintain 1.5 Mbytes. For a 32 Mbyte SRAM the total number of stages that can be stored is 21 and the maximum length of the IP address represented in the pipeline is 37 bits (i.e. 17420). For IPv4 a single chip can accommodate 256K IP addresses. For IPv6 additional chips are necessary. If a chip holds 32 Mbytes another 5 chips are necessary for IP addresses of length 128 bits.

Apart from memory allocation, the other problem with cascade architecture is the priority to output results when there is more than one stage of the pipeline that has the next-hop address index. To overcome this the next-hop address is output only at the end of the pipeline. This means that the order of the incoming packets is the same as the order of the outgoing packets.

Multiple Instruction Multiple Data Architecture

FIG. 11 shows a generic MIMD architecture for IP forwarding based on the FSM design. It is possible for MIMD architecture to achieve an IP lookup average of less than one clock cycle. The IPM Holder is involved in loading and holding the IP addresses. In addition, once the IP address is loaded, level folding is applied to determine the start address of the IP lookup FSM. Since level folding is carried out in one clock cycle it can be used within a pipeline to sharing level folding between different forwarding machines, which saves storage space.

A problem with MIMD architecture is whether loading of each forwarding machine can be scheduled effectively. A simple mechanism using a ring buffer for holding the identifiers of idle forwarding machines is used. A pointer keeps track of the front of the queue stored in the ring buffer and another pointer keeps track of the back of the queue. Whenever a forwarding machine is assigned to the current packet the (front) pointer is advanced to the next position in the ring buffer.
If the end of the buffer is encountered the next position is the zeroth position of the ring buffer. If a forwarding machine has completed a task the (back) pointer adds the identifier of the new idle forwarding machine at the back of the queue and the pointer is moved backward by one position. If the pointer reached the zeroth position of the buffer the pointer is set to the end of the buffer. When the front and back pointers are pointing at the same location all forwarding machines are in operation and incoming packets have to wait before processing can begin. The size of the ring buffer is the same as the number of forwarding machines that can be packed in a chip since the total number of idling forwarding machines is the total number of available machines in a chip.

When two more or more forwarding machines are idled simultaneously and their identifiers need to be placed in the ring buffer, a conflict as to which machine should be placed first. Since each forwarding machine can forward identical packets at the same rate, only serialization of the queue up process is needed since there are no priorities to determine which idle forwarding machine to choose.

Another issue with MIMD architecture is whether the forwarding machines can be updated simultaneously. If not, the worst-case update time can be increased by a multiplicative factor of the number of forwarding machines in a chip. One solution to this problem is to update asynchronously but by means of scheduling. The updated forwarding machines are used for searching and the not updated machines are simultaneously and asynchronously updated. The drawback of this solution is that it reduces the number of available forwarding machines during update of the database and therefore reduces the forwarding rate during updates.

Another solution is to suspend the current forwarding operation of all machines while the update if completed. For insertions and modifications, all forwarding machines synchronously update the database. For deletions, synchronous update is more complicated.

Where in the foregoing description reference has been made to integers or elements have known equivalents then such are included as if individually set forth herein.

Embodiments of the invention have been described, however it is understood that variations, improvement or modifications can take place without departure from the spirit of the invention.

What is claimed is:

1. A computer-implemented method comprising:
   storing, by a computing device, a data structure with a tree form having at least two levels for storing two or more strings and at least one associated output string, wherein the data structure comprises at least one node at a first level of the tree form data structure having a field for storing a first substring of the two or more stored strings, at least three nodes at a second level of the tree form data structure respectively storing three substrings of the two or more stored strings, and at least three transitions, a match transition, a first fail transition, and a second fail transition, respectively governing transition from the one node at the first level to a first, a second and a third of the at least three nodes at the second level;
   receiving, by the computing device, a string;
   comparing, by the computing device, a first portion of the string to the substring of the one node of the first level; in response to the first portion of the string matching the substring of the one node of the first level, transitioning, by the computing device, from the one node of the first level, using a match transition, to the first node of the second level; in response to the first portion of the string not matching the substring of the one node of the first level, transitioning, by the computing device, from the one node of the first level, using a selected one of the first and second fail transitions, to a corresponding selected one of the second and third node of the second level.

2. The method of claim 1, further comprising identifying one of the at least one associated output strings based at least in part on a matched one of the two or more stored strings having the substrings stored with the one node of the first level, and the first node of the second level, in response to the first node of the second level being a leaf node of the tree form data structure.

3. The method of claim 1, wherein transitioning, by the computing device, from the one node of the first level, using a selected one of the first and second failed transitions, to a corresponding selected one of the second and third node of the second level of the tree form data structure, comprises:
   comparing, by the computing device, an initial character of the first portion of the string with a character associated with the first fail transition;
   in response to the initial character matching the character associated with the first fail transition, transitioning, by the computing device, from the one node of the first level, using the first fail transition, to the second node of the second level;
   and in response to the initial character not matching the character associated with the first fail transition, transitioning, by the computing device, from the one node of the first level, using the second fail transition, to the third node of the second level.

4. The method of claim 1, wherein:
   the at least three nodes at the second level comprise first at least three nodes;
   the three substrings of the two or more stored strings comprise first three substrings;
   the at least three transitions comprise first at least three transitions;
   the match transition comprises a first match transition;
   the data structure further comprises:
   second at least three nodes at a third level of the tree form data structure respectively storing second three substrings of the two or more stored strings; and
   second at least three transitions, a second match transition, a third fail transition, and a fourth fail transition, respectively governing transition from the first of the at least three nodes at the second level to fourth, fifth and sixth node of the second at least three nodes at the third level; and
   the method further comprises:
   on transitioning to the first node of the at least three nodes at the second level, and the first node is not a leaf node of the tree form data structure:
   comparing, by the computing device, a second portion of the string immediately following the matched first portion of the string to the substring of the first node of the second level;
   in response to the second portion of the string matching the substring of the first node of the second level, transitioning, by the computing device, from the first node of the second level, using the second match transition, to the fourth node of the third level; and
   in response to the second portion of the string not matching the substring of the first node of the second level, transitioning, by the computing device, from the first node of the second level, using a
19
selected one of the third and fourth failed transitions, to a corresponding selected one of the fifth and sixth node of the third level.

5. The method of claim 2, wherein:
the string comprises a received IP address;
the stored two or more strings comprise two or more stored IP addresses;
the at least one associated output string comprises at least one next-hop address; and
identifying one of the at least one output string based at least in part on a matched one of the two or more stored strings identifying one of the at least one next-hop IP address based at least in part on one of the two or more stored IP addresses that matches the received IP address.

6. The method of claim 5, wherein:
the method further comprises receiving, by the computing device, an IP packet having a destination IP address;
the received IP address comprises the destination IP address; and
the method further comprises routing, by the computing device, the received packet based on the identified next-hop address.

7. The method of claim 5, wherein the character associated with the first fail transition is a 1 and the character associated with the second fail transition is a 0.

8. The method of claim 1, wherein the at least three nodes at the second level comprise at least a tree node and two trie nodes.

9. The method of claim 8, wherein the first of at least three nodes at the second level comprises the trie node, and transitioning to the first of the at least three nodes at the second level comprises transitioning to the tree node.

10. The method of claim 8, wherein the second and third nodes of at least three nodes at the second level comprise two trie nodes, and transitioning to a selected one of the second and third nodes at the second level comprises transitioning to a selected one of the two trie nodes.

11. One or more computer-readable storage media including instructions stored thereon which, in response to execution by a computing device, cause the computing device to:
store a data structure with a tree form having at least two levels for storing two or more strings and at least one associated output string, wherein the data structure comprises at least one node at a first level of the tree form data structure having a field for storing a first substring of the two or more stored strings, at least three nodes at a second level of the tree form data structure respectively storing three substrings of the two or more stored strings, and at least three transitions, a match transition, a first fail transition, and a second fail transition, respectively governing transition from the one node at the first level to a first, a second, and a third of the at least three nodes at the second level of the tree form data structure;
receive a string;
compare a first portion of the string to the substring of the one node of the first level;
in response to a match of the first portion of the string to the substring of the one node of the first level, use the match transition to transition from the one node of the first level to the first node of the second level;
in response to a mismatch of the first portion of the string to the substring of the one node of the first level, use a selected one of the first and second fail transitions to transition from the one node of the first level to a corresponding selected one of the second and third node of the second level.

12. The one or more computer-readable storage media of claim 11, wherein the instructions are further configured, in response to execution, to cause the computing device to identify one of the at least one associated output strings based at least in part on a matched one of the two or more stored strings having the substrings stored with the one node of the first level, and the first node of the second level, in response to the first node of the second level being a leaf node of the data structure.

13. The one or more computer-readable storage media of claim 11, wherein the instructions are configured, in response to execution, to cause the computing device to transition, from the one node of the first level, using a selected one of the first and second failed transitions, through:
comparison of an initial character of the first portion of the received string with characters associated with the first and second fail transitions;
in response to a match of the initial character to the character associated with the first fail transition, use of the first fail transition to transition to the second level of the tree form data structure; and
in response to a match of the initial character to the character associated with the second fail transition, use of the second fail transition to transition to the second level of the tree form data structure.

14. The one or more computer-readable storage media of claim 12, wherein:
the string comprises a received IP address;
the stored two or more strings comprise two or more stored IP addresses;
the at least one associated output string comprises at least one next-hop address; and
the instructions are further configured to cause the computing device, in response to execution of the instructions, to identify one of the at least one output string based at least in part on a matched one of the two or more stored strings through identification of one of the at least one next-hop IP address based on at least in part on one of the two or more stored IP addresses that matches the received IP address.

15. The one or more computer-readable storage media of claim 14, wherein:
the instructions are further configured to cause the computing device, in response to execution of the instructions, to receive an IP packet having a destination IP address;
the received IP address comprises the destination IP address; and
the instructions are further configured to cause the computing device, in response to execution of the instructions, to route the received packet based on the identified next-hop address.

16. The one or more computer-readable storage media of claim 11, wherein:
the at least three nodes at the second level comprise at least a trie node and two trie nodes;
the first of the at least three nodes at the second level comprises a trie node;
the second and third nodes at the second level comprise two trie nodes;
the instructions are configured to cause the computing device, in response to execution of the instructions, to transition to the first of the at least three nodes at the second level through transition to the trie node; and
the instructions are configured to cause the computing device, in response to execution of the instructions, to
17. An apparatus, comprising:
a memory configured to store strings;
a searchable data structure with trie form, stored in the
memory, and having at least two levels for storing two or
more strings and at least one associated output string,
wherein the data structure comprises at least one node at
a first level of the trie form data structure having a field
for storing a first substring of the two or more stored
strings, a trie node and two trie nodes at a second level
of the trie form data structure respectively storing three
substrings of the two or more stored strings, and at least
three transitions, a match transition, a first fail transition,
and a second fail transition, to the trie node and two trie
nodes at the second level; and
one or more processors coupled to the memory and con-
figured to execute logic to:
store the trie form data structure;
receive a string;
compare an initial portion of the string to the substring
of the one node of the first level;
in response to a match of the initial portion of the string
to the substring of the one node of the first level, use
the match transition to transition from the one node of
the first level to the trie node of the second level;
in response to a mismatch of the initial portion of the
received string to the substring:
compare an initial character of the initial portion of
the received string with characters associated with
the first and second fail transitions;
in response to a match of the initial character to the
character associated with the first fail transition,
use the first fail transition to transition to a first trie
node of the trie nodes of the second level of the trie
form data structure; and
in response to a match of the initial character to the
character associated with the second fail transition,
use the second fail transition to transition to a sec-
ond trie node of the trie nodes of the second level of
the trie form data structure.

18. The apparatus of claim 17, wherein:
the string comprises a received IP address;
the stored two or more strings comprise two or more stored
IP addresses;
the at least one associated output string comprises at least
one next-hop address; and
the logic is further to be executed by the processors to
identify one of the at least one next-hop address based at
least in part on a matched one of the two or more stored
IP addresses having the substrings stored with the one
node of the first level, and the trie node of the second
level, in response to the trie node of the second level
being a leaf node of the data structure.

19. The apparatus of claim 18, wherein:
the logic is further to be executed by the processors to
receive an IP packet having a destination IP address;
the received IP address comprises the destination IP
address; and
the logic is further to be executed by the processors to route
the received packet based on the identified next-hop
address.

20. The apparatus of claim 19, wherein the character asso-
ciated with the first fail transition is a 1 and the character
associated with the second fail transition is a 0.

***