High-Performance Longest Prefix Matching supporting High-Speed Incremental Updates and Guaranteed Compression

Mikael Sundström*
*Department of Computer Science and Electrical Engineering
Luleå University of Technology
SE-971 87 Luleå
Sweden
E-mail: msm@csee.ltu.se

Lars-Åke Larzon*†
†Department of Information Technology
Uppsala University
SE-751 05 Uppsala
Sweden
E-mail: lln@it.uu.se

***Abstract*— Longest prefix matching is frequently used for IP forwarding in the Internet. Data structures used must be not only efficient, but also robust against pathological entries caused by an adversary or misconfiguration.**

In this paper, we attack the longest prefix matching problem by presenting a new algorithm supporting high lookup performance, fast incremental updates and guaranteed compression ratio. High lookup performance is achieved by using only four memory accesses. Guaranteed compression ratio is achieved by combining direct indexing with an implicit tree structure and carefully choosing which construct to use when updating the forwarding table. Fast incremental updates are achieved by a new memory management technique featuring fast variable size allocation and deallocation while maintaining zero fragmentation.

An IPv4 forwarding table data structure can be implemented in software or hardware within 2.7 Mb of memory to represent 2^{18} routing entries. Incremental updates require only 752 memory accesses in worst case for the current guaranteed compression ratio. For a hardware implementation, we can use 300 MHz SRAM organized in four memory banks and four pipeline stages to achieve a guaranteed performance of 300 million lookups per second, corresponding to ~ 100 Gbit/s wire speed forwarding, and 400,000 incremental updates per second.

In measurements performed on a 3.0 Ghz Pentium 4 machine using a routing table with more than 2^{17} entries, we can forward over 27 million IPv4 packets per second, which is equivalent to wire speeds exceeding 10 Gbit/s. On the same machine and with the same routing table, we can perform over 230,000 incremental updates/second.

I. Introduction

When the Internet addressing architecture changed from class-based to classless in 1993, the complexity of forwarding lookups increased. It became necessary to locate the *longest matching prefix* for the destination address of every single packet that traverses an Internet router. This operation, called *longest prefix matching* (LPM) is not only performed by routers, but most devices that need to map an IP address into the smallest possible set of aggregated prefixes that includes the address itself. For long, it was assumed to be hard to do routing lookups in software to match the increasing line speeds. But since 1997, a number of new LPM algorithms that support routing at gigabit speeds on a standard PC have been introduced, e.g. [1] [2] [3] [4] [5] [6].

For most proposed algorithms, it is possible to create *pathological entries* in the prefix list, causing the data structures to consume significantly more memory space than in the normally reported average case. Such entries could be caused by route flapping as observed in [7], but also by an adversary with perfect knowledge about the algorithm in use, possibility to access it and a malicious mind, or simply by misconfiguration.

We argue, that in order for an LPM algorithm to be usable in an application, e.g., an Internet router, there must be strict guarantees regarding how much system resources it will consume in a worst-case scenario. We want to be able to guarantee support of at least N_{max} prefixes within a fixed amount of available memory, e.g., in a hardware implementation. This requires a guaranteed compression ratio, i.e., the maximum amount of space required per prefix. At the same time, we want to guarantee the performance of lookups and incremental updates.

We present a high-performance LPM algorithm with guaranteed boundaries for memory consumption and number of memory accesses per lookup that supports incremental updates at high speeds without exceeding the guaranteed size. On a 3.0 Ghz Pentium machine, a reference implementation of our LPM algorithm can perform over 27 million lookups per second in a routing table containing 131,227 routes within 4 memory accesses per lookup and supports more than 230,000 worst-case incremental updates per second.

The paper is organized as follows. Section II describes how the construction of, lookup in and update of the data structure are carried out. In section III, we evaluate the performance of our proposed LPM algorithm using a reference implementation. Section IV contains a discussion of the performance in relation to alternative LPM algorithms similar to the one proposed, followed by a conclusion in section V.

II. The Forwarding Table Data Structure

In this section we present our core result – an IPv4 forwarding table data structure with the following characteristics:

- At most $t_{\text{lookup}} = 4$ memory blocks of size $b = 256$ bits need to be accessed to lookup the $d = 13$ bits next-hop index of a given IP-address.
- In addition to a fixed cost of $4 \cdot 2^{16}$ bytes, at most 10 bytes per routing entry is required to represent the data structure.
- At most $t_{\text{update}} = 752$ memory accesses are required to update the data structure while maintaining the compression ratio of 10 bytes per route.
- The data structure in the current configuration can grow to handle $419,430$ routing entries without modifications.

All the techniques applied in achieving our core result and the building blocks of the data structure are generic and can be applied to other configurations. For example, it is straightforward to implement support for more than 2^{13} different next-hops by sacrificing a small portion of the compression ratio. It is also possible to increase the performance of incremental updates by decreasing the performance for lookup (increasing t_{lookup}) and/or by decreasing the compression ratio.

A. Overview

The forwarding table data structure is essentially a two level 16-8 *variable stride trie* [8] [9] on top of a set of tree structures called *block trees* where the remaining 2 to 3 memory accesses are spent to complete the lookup. The data structure is outlined in Fig. 1.

A block tree, or more precisely a (t, w) block tree, is an $O(n)$ space *implicit* tree structure for representing a partition of a set of w bits non-negative integers that supports search operations using at most t memory accesses. For given values of t and w there are a maximum number of intervals $n_{\max}(t, w)$ that can be represented. If the number of intervals at the second level is $\leq n_{\max}(3, 16)$ we can use a $(3, 16)$ block tree as an alternative to the 2^8-ary trie node. Thereby, we can reduce the worst case amortized space per interval and achieve a better guaranteed compression ratio for the whole data structure.

B. Block Trees

The idea behind block trees is similar to the multiway binary search used by Lampson et. al. in [3] to improve performance when searching among interval endpoints, where the sorted list of interval endpoints and next-hop indices is reorganized into *nodes* and *leaves* stored in b bits blocks. With block trees, however, we take this further. To obtain the best compression ratio and lookup performance for a given input size we use an implicit data structure where the sub-tree pointers in the nodes are skipped altogether. Instead, we store the sub-trees of a node in order in the blocks immediately after the node. Moreover, we make sure that each sub-tree, except possibly the last, is *complete* or *full* so that we know its size in advance. We can then search the node to obtain an *index*. The index is then multiplied with the size of the sub-tree to obtain an offset ahead from the node to the sub-tree where to continue the search.

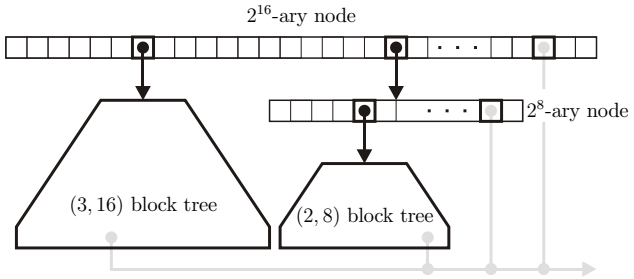

Fig. 1. Outline of the data structure. The first level is implemented as an array of pointers, referring to a $(3, 16)$ block tree, a 2^8-ary node or a next-hop index (shaded arrows). At the second level, each pointer contains either a next-hop index or a reference to a $(2, 8)$ block tree.

A general block tree is characterized by the block size b and the maximum number of blocks t needed to be accessed during lookup. We will sometimes refer to t as the number of *levels* or the *height* of the block tree. Besides b and t, a block tree is also characterized by the size of the data d and the number of bits required to represent a range boundary w. For given values of t, w, b, and d the maximum number of intervals that can be represented by a (t, w, b, d) block tree is

$$n_{\max}(t, w, b, d) = \left(\left\lfloor \frac{b}{w} \right\rfloor + 1 \right)^{t-1} \cdot \left\lfloor \frac{b + w}{d + w} \right\rfloor .$$

For our fixed values of $b = 256$, $d = 13$, and $(t, w) = (3, 16)$ and $(2, 8)$ respectively, we then get

$$\begin{aligned} n_{\max}(3, 16) &= 17 \cdot 17 \cdot 9 = 2601, \text{ and} \\ n_{\max}(2, 8) &= 33 \cdot 12 = 396. \end{aligned}$$

Observe that $n_{\max}(2, 8) > 2^8$. This means that a $(2, 8)$ block tree can always handle the intervals below the two levels of trie nodes described in the previous section. By definition, a block tree of height 1 occupies one block. A complete block tree of height 2 requires one block for the node and one for each possible leaf. For a $(2, 16)$ block tree we get a total of 18 blocks since we can fit 16 interval endpoints into 256 bits to support up to 17 leafs. Storing a complete block tree in general requires

$$S(t, w, b, d) = \frac{\left(\left\lfloor \frac{b}{w} \right\rfloor + 1 \right)^t - 1}{\left(\left\lfloor \frac{b}{w} \right\rfloor + 1 \right) - 1}$$

blocks. Derivation of the expressions for $n_{max}(t, w, b, d)$ and $S(t, w, b, d)$ are straightforward. A more thorough description of block trees and their properties can be found in [10].

Constructing a *complete* $(t, 16)$ block tree from a sorted list of interval endpoints $r_1, r_2, \ldots, r_{n-1}$ is achieved by the following recursive procedure. Interval $R_i = \{r_{i-1}, r_{i-1} + 1, \ldots, r_i - 1\}$ is associated with next-hop index D_i and T represents a pointer to the first block available for

storing the resulting block tree. The procedure is expressed as a base case (leaf) and an inductive step (node).

$build_{(1,16)}\,(T, r_1 \ldots r_8, D_1 \ldots D_9) =$
  store $r_1 \ldots r_8$ and $D_1 \ldots D_9$ in block T
$build_{(t,16)}\,(T, r_1 \ldots r_{n-1}, D_1 \ldots D_n) =$
  $m \leftarrow n_{\max}\,(t-1, 16)$
  store $r_m, r_{2m}, r_{3m}, \ldots, r_{16m}$ in block T
  $T \leftarrow T + 1$
  for $i \leftarrow 0 \ldots 16$ **do**
    $R \leftarrow r_{im+1} \ldots r_{(i+1)\cdot m-1}$
    $D \leftarrow D_{im} \ldots D_{(i+1)\cdot m}$
    $build_{(t-1,16)}\,(T, R, D)$
    $T \leftarrow T + S\,(t-1, 16)$
  end

Observe that in each round in the loop the block tree pointer is moved forward according to the size of the sub-tree constructed recursively. To construct a *partial* $(t, 16)$ block tree we use essentially the same procedure except that we bail out when running out of intervals.

Looking up a query key q in a $(3, 16)$ block tree starting at block T is achieved by the following simple procedure. (Endpoints between parentheses are stored implicitly).

$lookup_{(3,16)}\,(T, q) =$
  find $\min r_i > q$ in $r_1, r_2, \ldots, r_{16}, (r_{17})$
  $T \leftarrow T + 1 + 18 \cdot (i-1)$
  find $\min r_i > q$ in $r_1, r_2, \ldots, r_{16}, (r_{17})$
  $T \leftarrow T + 1 + (i-1)$
  find $\min r_i > q$ in $r_1, r_2, \ldots, r_8, (r_9)$
  return D_i

There are corresponding but slightly simpler procedures for construction of and lookup in $(2, 8)$ block trees.

C. Lookup

We have now described the building blocks of the forwarding table – straightforward pointer arrays and block trees. Before moving on to discuss construction and incremental updates we describe in more detail how a lookup is performed and how the forwarding table data structure is used together with a next-hop table. The way we want to perform lookup will determine to some extent how to construct, update and maintain the data structure.

An overview of the complete forwarding table data structure, how it is organized in memory and how a lookup is performed is shown in Fig. 2. The memory is organized into four major areas: *level 1* containing *one* array of 2^{16} pointers, the *next-hop table* which contains up to 2^{13} next-hop entries, *level 2* containing an array of arrays of 2^8 pointers each, and *level 2...4* which contains $(3, 16)$ block trees and $(2, 8)$ block trees of of various sizes. We refer to the areas as levels since the LPM data structure consists of four levels where one memory access is spent in each level to compute the next-hop index into the fifth level – the next-hop table.

Looking up the next-hop index and retrieving the next-hop information is described by the procedure below.

$lookup\,(a) =$	step	line
$p \leftarrow L1\,[a_{31\ldots16}]$	1	1
$def \leftarrow p.def$		2
if $p.code = 00_{\text{bin}}$ **then return** $L5\,[def]$	2a	3
elsif $p.code = 10_{\text{bin}}$ **then**	2c	4
$ix \leftarrow lookup_{(3,16)}\,(L2 \ldots 4\,[p.ref], a_{15\ldots0})$	2c	5
if $ix \neq 0$ **then** $def \leftarrow ix$ **end**		6
return $L5\,[def]$	3c	7
end		8
$p \leftarrow L2\,[p.ref]\,[a_{15\ldots8}]$	2b	9
if $p.def \neq 0$ **then** $def \leftarrow p.def$ **end**		10
if $p.code = 00_{\text{bin}}$ **then**	3a	11
return $L5\,[def]$	3a	12
end		13
$ix \leftarrow lookup_{(2,8)}\,(L2 \ldots 4\,[p.ref], a_{7\ldots0})$	3b	14
if $ix \neq 0$ **then** $def \leftarrow index$ **end**		15
return $L5\,[def]$	4	16

The steps in the lookup procedure refers to the steps in Fig. 2. There are however a number of steps that are not obvious and we will now go through these. We have reserved next-hop index zero to represent lookup failure and this is used throughout the data structure. Initially during lookup, no prefix (range) has yet matched the address. In line 2, we record the default next-hop index field from the pointer in variable def. If the value is zero, it means that there is *no* prefix of length $0 \ldots 16$ that matches the IP-address. On the other hand, if the value is non-zero, def will contain the next-hop index of the longest matching prefix P of a among the prefixes of length $0 \ldots 16$. Observe that among the prefixes of length $0 \ldots 16$, P is the longest matching prefix of any IP-address a' where $a'_{31\ldots16} = a_{31\ldots16}$. In fact, def will contain the *default next-hop index* for sub-universe $a_{31\ldots16}$. After performing the block tree lookup in line 5 (14), we store the index in variable ix and in line 6 (15) we check is ix is non-zero update def accordingly if so. If ix is zero, it means that a does not match any prefix of length $17 \ldots 32$ and that the longest matching prefix of a is either P or that a does not have any matching prefix.

Slightly similar to the unconditional assignment of def in line 2, we update def in line 10 if and only if the default next-hop index field in the pointer extracted from level 2 is non-zero, i.e., if and only if a has a matching prefix of length $17 \ldots 24$. This technique for storing and managing default next-hop indices for sub-universes is one of the key components in achieving high performance incremental updates while simultaneously maintaining a high compression ratio without affecting the lookup performance. In effect, we create a seal between classes of prefixes of length $0 \ldots 16$ and prefixes of length $17 \ldots 32$ which allows us to insert or delete a prefix from one class without affecting the other class. In the high density case, when we use 8-bit trie nodes and $(2, 8)$ block trees, there is a corresponding seal (locally) between prefixes of lengths $17 \ldots 24$ and $25 \ldots 32$.

D. Construction and Updates

To use and maintain the forwarding table, a number of support structures and mechanisms are required. In the pre-

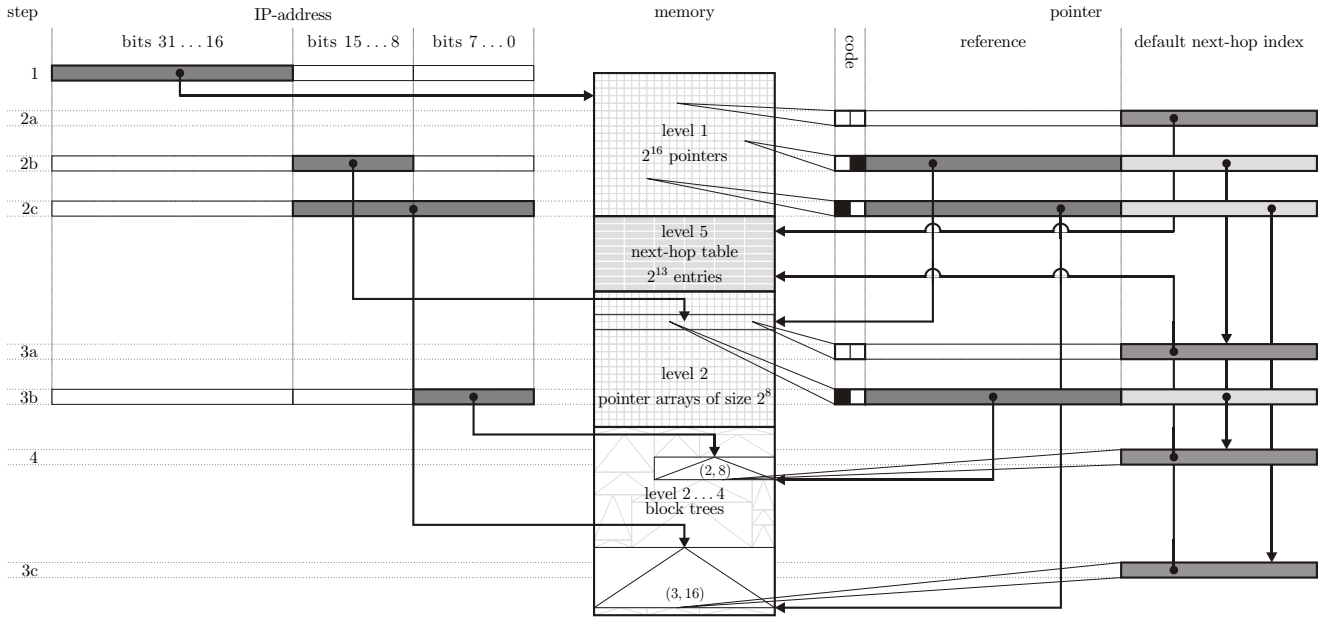

Fig. 2. Overview of the complete forwarding table lookup

vious section we mentioned the next-hop table, which is an array where each slot contains a piece of *next-hop information*, e.g., outward interface and IP-address of next-hop router. We will assume that there is a mechanism (for example a *hash table*) for managing next-hop information in the next-hop table and associating next-hop indices such that identical next-hop information have identical next-hop indices.

The most important support structure is the routing table which contains all routing entries and interfaces with the routing protocol software. Each routing entry is defined an *address prefix* and a piece of associated next-hop information. The address prefix is typically represented by a 32 bits *network address* and a *netmask*. Almost any kind of dynamic routing table data structure, for example a BSD Radix Tree, can be used together with our forwarding table but we will add some extra features to facilitate for fast incremental updates. When a new entry is inserted, the operations performed on the forwarding and next-hop table depends on the length of the prefix and therefore prefixes are classified according to their prefix length.

There are four different classes α, β, γ, and δ containing prefixes of length $0 \ldots 8$ bits, $9 \ldots 16$ bits, $17 \ldots 24$, and $17 \ldots 32$ bits respectively. Each prefix class has a custom insertion procedure and we will now go through these in detail.

Class α prefixes.
: Insert the prefix P into a separate table T_a containing only class α prefixes. For each IP-address of the form $a.0.0.0$ where P is the longest matching prefix in T_a, store the next-hop information associated with P in slot $a + 1$ in the next-hop table.

Class β prefixes.
: Insert the next-hop information in the next-hop table and obtain a next-hop index $index$. Insert the prefix P into a separate table T_β containing only class β prefixes. For each IP-address of the form $a.b.0.0$ where P is the longest matching prefix in T_β, store $index$ in the default next-hop index field of pointer $2^8 \cdot a + b$ in the first level trie node.

Class γ and δ prefixes (low density).
: *Applicable if the the current sub-universe is empty or represented by a block tree in the forwarding table.* Insert the next-hop information in the next-hop table and obtain a next-hop index $index$. Let u be the 16 most significant bits of the prefix P. (u is referred to as the sub-universe index of P.) Insert P in a table $T_{\gamma \text{ and } \delta}[u]$ containing only class γ and δ prefixes with the same sub-tree index as P. Compute the *density* n of sub-universe u, i.e. the number of basic intervals in the partition of the sub-universe defined by the prefixes in $T_{\gamma \text{ and } \delta}[u]$. If $n > 2295$[1] the current sub-universe must be handled as a *high density* case. Otherwise, we allocate space for constructing a new block tree (or reuse the space for the current one if possible) and then construct a new block tree from the list of intervals in the partition. Each interval that does not correspond to a matching class γ or δ prefix will be associated with next-hop index zero.

Class γ prefixes (high density).
: *Applicable if a 2^8-ary trie node that corresponds to*

[1]We could use $n_{\max}(3, 16) = 2601$ instead of 2295 if the data structure is static but we will need four bytes from the first block tree node later on to implement incremental updates.

the current sub-universe is in use in the forwarding table or if the density is too high for a low density case. Insert the next-hop information in the next-hop table and obtain a next-hop index $index$. As above, let the u be the sub-universe index of P. Insert P in a table $T_\gamma[u]$ containing only class γ with the same sub-tree index as P. For each IP-address of the form $a.b.c.0$ where P is the longest matching prefix in $T_\gamma[u]$, store $index$ in the next-hop index field of pointer c in the second level trie node associated with sub-universe u.

Class δ prefixes (high density).
Insert the next-hop information in the next-hop table and obtain a next-hop index $index$. Let u be the 24 most significant bits of the prefix P. (u is referred to as the sub-universe index of P.) Insert P in a table $T_\delta[u]$ containing only class δ prefixes with the same sub-tree index as P. Compute the *density* n of sub-universe u, i.e. the number of basic intervals in the partition of the sub-universe defined by the prefixes in $T_\gamma[u]$. The density can not exceed 2^8 and we can always use a $(2, 8)$ block tree to represent the partition. Therefore, allocate space for constructing a new block tree (or reuse the space of the current one if possible) and then construct a new block tree from the list of intervals in the partition. As above, uncovered intervals are associated with next-hop index zero.

For class α prefixes we use a technique not mentioned in the previous lookup section. The reason for treating these short class α prefixes specially instead treating all prefixes of length $0 \ldots 16$ the same is to avoid updates of large portions of the first level trie node in the forwarding table. If the result from the lookup described in the previous section is zero the query IP-address does not match any prefix of length $9 \ldots 32$ but there is still a possibility that a class α prefix is matched. By reserving slots $1 \ldots 256$ of the next-hop table for class α prefixes and using this technique, a copy of the next-hop information associated with the longest such prefix is stored in the slot with the same index as the 8 most significant bits of the query IP-address plus 1. The corresponding procedures for deleting prefixes of different classes are essentially the inverses of the insert procedures.

What is then the cost for incremental updates of the forwarding table? Since we consider *only* memory accesses in the forwarding table and the next-hop table, inserting or deleting a class α prefix may require that the first 256 slots in the next-hop table are accessed. Assuming that each piece of next-hop information occupies 8 bytes, we then need to access 64 memory blocks. For class β prefixes, the updates takes place in the first level trie node. We only update pointers that corresponds to the inserted or deleted prefix and since the prefix is at least 9 bits long, at most 128 pointers needs to be updated. Since each pointer occupies 4 bytes, the total number of blocks accessed is for class β updates is 16. For class γ and δ prefixes, the cost for incremental updates is directly related to the size of the block trees and second level trie nodes constructed since they must be filled with information. However, even more important is the costs for *allocating* and *deallocating* space. In the following subsection, we present a memory management algorithm which allows us to perform allocation and deallocation at virtually the same cost as the actual construction cost while maintaining zero fragmentation and maximum compression ratio. The memory management algorithm is then configured in section II-F to achieve an optimal memory management cost in relation to the promised compression ratio.

E. Stockpiling

Consider the general problem of allocating and deallocating memory areas of different sizes from a *heap* while maintaining zero fragmentation. In general, allocating a *contiguous* memory area of size s blocks is straightforward – we simply let the heap grow by s blocks. Dellocation is however not so straightforward. Typically, we end up with a hole somewhere in the middle of the heap and a substantial reorganization effort is required to fill the hole. An alternative would be to relax the requirement that memory areas need to be contiguous. It will then be easier to create patches for the holes but it will be nearly impossible to use the memory areas for storing data structures etc.

We need a memory management algorithm which is something in between these two extremes. The key to achieve this is the following observation: *In the block tree lookup, the leftmost block in the block tree is always accessed first followed by accessing one or two additional blocks beyond the first block.* It follows that a block tree can be stored in two parts where information for locating the second part and computing the size of the respective parts is available after accessing the first block.

A *stockling* is a *managed memory area* of s blocks (i.e. b bits blocks) that can be moved and stored in two parts to prevent fragmentation. It is associated with information about its size s, whether or not the area is divided in two parts and the location and size of the respective parts. Moreover, each stockling must be associated with the *address to the pointer* to the data structure stored in the stockling so it can be updated when the stockling is moved, a technique also used in [11]. Finally, it is associated with a (possibly empty) procedure for encoding the location and size of the second part and the size of the first part in the first block.

Let n_s be the number of stocklings of size s. These stocklings are stored in, or actually constitutes a, *stockpile* which is a contiguous sn_s blocks memory area. A stockpile can be moved one block to the left by moving one block from the left side of the stockpile to the right side of the stockpile (the information stored in the block in the leftmost block is moved to a free block at the right of the rightmost block). Moving a stockpile one block to the right is achieved by moving the rightmost block to the left side of the stockpile. The rightmost stockling in a stockpile is possibly stored in two

parts while all other stocklings are contiguous. If it is stored in two parts, the left part of the stockling is stored in the right end of the stockpile and the right end of the stockling at the left end of the stockpile. This technique was also used in [12].

Assume that we have c different sizes of stocklings $s_1, s_2, \ldots, s_c$ where $s_i > s_{i+1}$. We organize the memory so that the stockpiles are stored in sorted order by increasing size in the growth direction. Furthermore, assume without loss of generality that the growth direction is to the right. Allocating and deallocating a stockling of size s_i from stockpile i is achieved as follows:

Allocate s_i.

> Repeatedly move each of stockpiles $1, 2, \ldots, i-1$ one block to the right until all stockpiles to the right of stockpile i have moved s_i blocks. We now have a free area of s_i blocks at the right of stockpile i. If the rightmost stockling of stockpile i is stored in one piece, return the free area. Otherwise, move the left part of the rightmost stockling to the end of the free area (without changing the order between the blocks). Then return the contiguous s_i blocks area beginning where the rightmost stockling began before its leftmost part was moved.

Deallocate s_i.

> Locate the rightmost stockling that is stored in one piece (it is either the rightmost stockling itself or the stockling to the left of the rightmost stockling) and move it to the location of the stockling to be deallocated. Then reverse the allocation procedure.

In Fig. 3, we illustrate the stockpiling technique in the context of insertion and deletion of structures of size 2 and 3 in a managed memory area with stockling sizes $2, 3$ and 5. Each structure consist of a number of blocks and these are illustrated by squares with a shade of grey and a symbol. The shade is used to distinguish between blocks within a structure and the symbol is used to distinguish between blocks from different structures. We start with a 5-structure and then in (a) we insert a 2-structure after allocating a 2-stockling. Observe that the 5-structure is stored in two parts with the left part starting at the 6th block and the right part at the 3rd block. In (b) we allocate and insert 3 blocks and as a result, the 5-structure is restored into one piece. A straightforward deletion of the 2-structure is performed in (c) resulting in that both remaining structures are stored in two parts. Finally, in (d) a *new* 3-structure is inserted. This requires that we first move the 5-structure 3 blocks to the right. Then, the left part (only the white block in this case) of the *old* 3-structure is moved next to the 5-structure and finally the new 3-structure can be inserted.

The cost for allocating an s_i stockling and inserting a corresponding structure is computed as follows. First, we have to spend $(i-1) \cdot s_i$ memory accesses for moving the other stockpiles to create the free space at the end of the stockpile. We then have two cases: (i) Insert the data structure directly into the free area. The cost for this is *zero memory accesses*

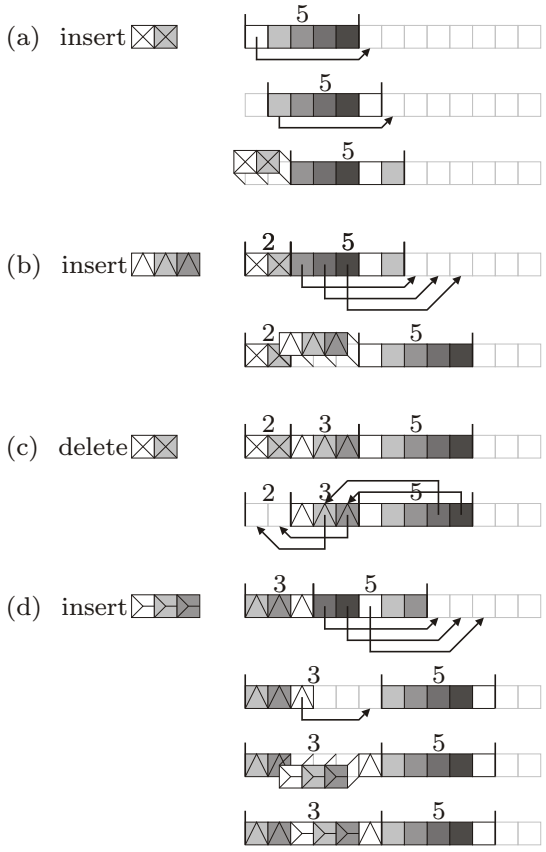

Fig. 3. Examples of stockpiling

since we have already accessed the free area when moving the stockpiles (insertion can be done simultaneously while moving the stockpiles). (ii) We need move the leftmost part of the rightmost stockling. However, it occupies an area which will be overwritten when inserting the data structure. Therefore, we get an additional s_i memory accesses for inserting the data structure. For deallocation, we get an additional cost of s_i memory accesses since we may need to overwrite the deleted stockling somewhere in the middle of the stockpile. We also need to account for the cost for updating pointers to the data structures that are moved. Since the stockpiles are organized by increasing size, at most one pointer needs to be updated for each stockpile moved plus two extra pointer updates in the current stockpile. It follows that

Lemma 1: The cost for inserting a s_i blocks data structure when using stockpile memory management is $is_i + (i-1) + 2 = is_i + i + 1$ memory accesses and the cost for deletion is $(i+1) \cdot s_i + (i-1) + 2 = (i+1) \cdot s_i + i + 1$ memory accesses.

Stockpiling can be used also if it is not possible to store data structures in two parts. In each stockpile, we have a dummy stockling and ensure that after each reorganization, it is always the dummy stocklings that are stored in two parts. The extra cost for this is $\sum s_i$ space and, in worst case, $\sum_{j=1}^{i} s_j$ memory accesses for swapping the dummy with another stockling that happened to be split up after a reorganization.

F. Amortized Space and Update Costs

In this section we analyze the worst case amortized space and update costs for the forwarding table data structure. By our design choices, we have already fixed the lookup costs to 4 memory accesses for the longest prefix match operation and one additional memory access for retrieving the next-hop information.

We begin by computing the space required for the forwarding table. The cost for the first level is fixed to 2^{16} times the size of a pointer which is 4 bytes. This gives a total of 2^{18} bytes. For each sub-universe of density ≤ 2295, a $(3, 16)$ block tree is used. If the density is 2296 or larger, a 2^8-ary trie node of size $4{\cdot}2^8 = 1024$ bytes is used and below the node there will be a number of $(2, 8)$ block trees. The cost for the trie node is amortized over at least 2296 intervals. Hence, its contribution to the amortized cost per interval is $1024/2296$ bytes. Our goal is to achieve a *compression ratio* of 10 bytes per prefix (not counting the first level). This corresponds to 5 bytes per interval in $(3, 16)$ block trees and $5 - 1024/2296 \approx 4.554$ bytes per interval in $(2, 8)$ block trees.

Now consider a $(3, 16)$ block tree representing 2295 intervals. Storing such a tree requires $\lceil 2295/9 \rceil = 255$ leaves, $\lceil 255/17 \rceil = 15$ level 2 nodes, and 1 level 3 node giving a total of $255 + 15 + 1 = 271$ blocks. The amortized cost per interval is $271 \cdot 32/2295 \approx 3.7786$ bytes per interval which is considerably better than the design goal of 5 bytes. In fact, the number of intervals can be reduced down to 1734 before it becomes too expensive to use 271 blocks ($271 \cdot 32/1734 = 5.0012$). Hence, if the density is $1735 \ldots 2295$, we can afford to allocate 271 blocks for the block tree. Similarly, storing a $(3, 16)$ tree with 1734 intervals requires $\lceil 1734/9 \rceil = 193$ leaves, $\lceil 193/17 \rceil = 12$ level 2 nodes and 1 level 3 node. We then get a total of $193 + 12 + 1 = 206$ blocks. The amortized cost for representing 1734 intervals in 206 blocks is $206 \cdot 32/1734 \approx 3.8016$ bytes per interval and the number of intervals can be reduced down do 1318 before it becomes too expensive to use 206 blocks. By continuing the computations along these lines we obtain a mapping between densities and a minimal set of allocation units or *stockling sizes* used in our reference implementation according to table I.

To achieve the desired compression ratio for densities of 37 intervals or 18 prefixes or less, the quantization effects resulting from under utilized blocks need to be reduced. In the process, we will slightly deviate from the presented reference stockling sizes but not in a way that affects the analysis of the update costs. We use one of the following approaches or a combination of the two.

Quarter block trees: $(3, 16)$-block trees containing $10 \ldots 37$ intervals are represented using one node and a number of leafs. The maximum number of intervals endpoints in the node is $\lceil 37/9 \rceil - 1 = 4$ occupying 8 bytes and as long as these lies in the same block we can search among them in one memory access. We then have two memory accesses left to spend for the lookup and therefore, it does not matter if leafs are stored across a block boundary. Hence, for $10 \ldots 37$ intervals, we can store the block trees in special memory area with 8 byte blocks instead of 32 byte blocks and this is sufficient to achieve the desired compression ratio. Similarly, for $1 \ldots 9$ intervals, we use a single leaf stored in a special memory area with 2 byte blocks. These smaller block sizes requires that we increase the reference fields in the pointers to 21 bits instead of 17 and this requires that 4 of the 13 bits from the default next-hop index field are stored in the block tree itself. We use code 11_{bin} to distinguish this from the three standard cases.

Number of basic intervals	Stockling size
$1735 \ldots 2295$	271
$1319 \ldots 1734$	206
$1005 \ldots 1318$	157
$768 \ldots 1004$	120
$596 \ldots 767$	93
$461 \ldots 595$	72
$365 \ldots 460$	57
$288 \ldots 364$	45
$224 \ldots 287$	35
$180 \ldots 223$	28
$148 \ldots 179$	23
$135 \ldots 147$	21
$103 \ldots 134$	16
$84 \ldots 102$	13
$71 \ldots 83$	11
$58 \ldots 70$	9
$52 \ldots 57$	8
$45 \ldots 51$	7
$36 \ldots 44$	6
$27 \ldots 35$	5
$18 \ldots 26$	4
$10 \ldots 17$	3
$1 \ldots 9$	1

TABLE I

RELATION BETWEEN DENSITIES AND STOCKLING SIZES.

Prefix lists: Each (16 bits) prefix and the corresponding next-hop index requires $16 - x$ bits for the prefix itself, $x + 1$ bits for the prefix length (represented in base 1), and 13 bits for the next-hop index. The total number of bits for storing up to 18 prefixes and next-hop indices including the number of prefixes is $18 \cdot (16 - x + x + 1 + 13) + 5 = 545$. We can skip block trees and store the prefixes directly as a list. By storing prefix lists of length 18 in an 8 byte aligned contiguous memory area and smaller prefix lists in a 4 byte aligned memory area, we can guarantee that the prefix list does not straddle more than two block boundaries and can thus be searched in at most three memory accesses as required. Our storage requirement is 10 bytes per prefix and with this representation we use only 4 byte per prefix for $1 \ldots 18$ prefixes.

For $(2, 8)$ block trees, we use similar techniques to map their densities to stocklings of size 23 and smaller and the same approach to reduce quantization effects. Recall that the level 3 node is not fully utilized if the density is 2295 and the reason for this is that we need to use four bytes to encode stockling information (if the stockling is stored in two parts). The same goes for level 2 nodes. Therefore we step down to 2 level block trees first when the density is below 135. This

is the reason for the jump from 21 blocks to 16 blocks. Out of the four bytes we use 17 bits to reference to the right part of the stockling, 5 bits for the size (as there are 23 different sizes), and 9 bits for the size of the left part. At the cost of some additional pointer updates during incremental updates, we can even skip the encoding of the size of the left part to save 9 bits. This is achieved as follows: Let i be the index of the leftmost block of an s-stockling relative the base of the memory. Make sure that $i \equiv 0 \pmod{s}$ when allocating an s-stockling. For stocklings stored in two parts, store i plus the size of the left part in the pointer. Computing the size of the left part during lookup is then achieved by taking the pointer value modulo s.

Conceptually, we allocate a big chunk of memory for the forwarding table from the beginning. Since we have a guaranteed compression ratio, we can guarantee that for a given maximum number of prefixes $N_{\max}$ we will not run out of memory. At one end of the chunk, we place the first level trie node and the next-hop table as in Fig. 2. The rest of the chunk is our *dynamic memory*. A set of stockpiles is implemented in each end of the dynamic memory. They are called sp_{lo} and sp_{hi} respectively and grows, in opposite directions, towards each other. By the worst case amortized space boundary, we can guarantee that they will not grow into each other as long as the number of prefixes is less than $N_{\max}$. The stockpile sizes in the respective stockpile set are given by

$$\begin{aligned} sp_{\text{lo}} &= 206, 157, 93, 72, 5, 4, 3, 1, \mathbf{32}, \text{ and} \\ sp_{\text{hi}} &= 271, 120, 57, 45, 35, 28, 23, \\ &\quad 21, 16, 13, 11, 9, 8, 7, 6. \end{aligned}$$

Observe that we added stocklings of size **32** which will contain 2^8-ary trie nodes. Moreover, we have configured the stockpile so that this is the innermost size class. This means that the trie nodes will be located beside the next-hop table, but more importantly, trie nodes will be stored in one part. Thus it is no problem to perform direct indexing.

By Lemma 1, we can compute the cost for allocation and deallocation from each stockpile. To compute the worst case update cost, we only need consider *transitions* that can occur such that deallocating one size and allocating the next smaller or larger size (assuming that we can perform updates without worrying about simultaneous lookups). By exhaustive search among the possible transitions, we have found that the worst case transition is when a we go from 271 blocks, causing a deallocation cost of 544 memory accesses to 206 blocks at an allocation cost of 208 memory access. The total worst case transition cost between block trees is then $208 + 544 = 752$ memory accesses. Will it be more expensive than this when we go from 2295 intervals, in a block tree, to 2296 intervals which requires a trie node and possibly some $(2, 8)$ block trees? Handling this in a straightforward fashion will result in an explosive transition at a considerable expense. To achieve a smooth transition, we allocate a trie node (32-stockling) and then construct the *initial set* of $(2, 8)$ block trees inside the 271-stockling. Moreover, we keep the 271-stockling during the course of the lifetime of the trie node (i.e. as long as the density is larger than 2295) and use it as the preferred choice for storing $(2, 8)$ block trees. In this way, we avoid explosive and implosive transitions and obtain smooth transitions also on the verge of using block trees. Hence, the worst case cost for incremental updates is 752 memory accesses.

III. Performance evaluation

To evaluate the practical lookup and update speeds, we have written a reference implementation in C that measures the number of CPU cycles needed for each operation. In our tests, we have used a standard Pentium 4 PC with a 3.0 Ghz CPU.

A. Lookup performance

Our test application measures the average lookup times by looking up 10,000,000 random IP addresses. Each address is looked up twice to get measurements of uncached and cached entries respectively.

The application used to evaluate lookup performance supports forwarding tables that are either randomly created or read from a file. In our measurements, we have however not used any random datasets. The routing information services (RIS) project [13] provides an archive of BGP routing table snapshots from different points in the Internet that are updated three times a day. We have arbitrarily chosen to use a dataset collected in London at midnight on January 31:st 2004. The shortest prefix present in the table is 8 bits long, and 54 out of the 131227 prefixes are 32 bits long. Multicast and class E addresses are not included in either the datasets or the random lookups.

To enable comparison with other proposed algorithms, we have also included three popular datasets made available by IPMA in the past. The details about each dataset are shown in table II together with measured results.

The measured lookup speeds as presented in table II show the average number of CPU cycles needed when doing 10,000,000 random lookups on each dataset. Although these numbers are machine-specific, it hints the forwarding capacity supported by a software implementation of our data structure on a standard PC.

From the measured performance, we observe that the compression ratio increases with the dataset size. This is not surprising as a larger dataset increases the possibility of aggregating prefixes. Moreover, the cost of the root array is amortized over a larger number of prefixes. A normal router implementation using our data structure would most likely reduce the number of cache misses and try to keep as much of the data structure in the cache, e.g., by having a dedicated forwarding CPU. This suggests that on a standard dual-processor Pentium machine, it is possible to perform software-based IPv4 forwarding at tens of Gigabits per second - at worst.

The bottleneck is not expected to be in the forwarding processing, but in hardware components as it normally is hard to shuffle packets around in a standard PC at the speeds supported by our data structure. Actual performance could be

TABLE II

DATASET INFORMATION AND MEASURED LOOKUP PERFORMANCE

		Dataset			
		London	**Mae-East**	**Pac-Bell**	**Mae-West**
Number of entries in dataset		131,227	38,470	20,637	15,050
Number of unique prefixes		131,227	32,061	17,431	13,294
Total Size (bytes)		948,196	414,432	344,156	319,004
Bytes/routing entry (total)		7.23	10.77	16.68	21.20
Bytes/routing entry (excl. root array)		5.23	3.96	3.97	3.78
Uncached lookups	**CPU cycles/lookup**	107	65	39	36
	lookups/second	27,799,411	45,516,566	76,514,811	82,204,341
	supported wire speed (Gbit/s)	10,2	16,75	28,1	30,25
Cached lookups	**CPU cycles/lookup**	44	28	18	16
	lookups/second	67,673,468	103,558,663	165,602,475	179,201,603
	supported wire speed (Gbit/s)	24,9	38,1	60,9	65,9

TABLE III

DATASET INFORMATION AND MEASURED WORST-CASE UPDATE PERFORMANCE

	Dataset	
	S_{lo}	S_{hi}
Size of data structure (Mb)	4.0	2.9
Number of blocks	125919	92580
Avg. allocation (CPU cycles)	4320	4598
Avg. deallocation (CPU cycles)	7083	6642
Worst-case updates per second	233271	236654

TABLE IV

MEASURED PERFORMANCE FOR RANDOM UPDATES (UPDATES/SECOND)

	Dataset			
	London	**Mae-East**	**Pac-Bell**	**Mae-West**
Excl. support struct.	347,450	501,228	518,412	801,773
Incl. support struct.	219,988	300,384	327,465	414,948

increased by implementing the data structure on a line card, thus eliminating the need to pass packets over the internal data bus.

B. Incremental update performance

When measuring the incremental update performance, we have done two experiments. In the first experiment, we use a worst-case scenario including pathological route flapping on a large fictive routing table. In the second experiment, we perform random updates on the same data sets used in the lookup performance measurements.

Worst-case incremental updates: In this experiment, we use a fictive routing table containing 700,000 entries. The motivation for using a fictive routing table is to be able to create the worst-case update scenario in every update. This worst-case scenario is equivalent to pathological route flapping in the border area between different stockling sizes. In this case, an update means that the previous stockling must be deallocated before allocating a new stockling as is described in section II-F.

Our test application measures the average cost of deallocating and allocating stocklings for two different data corresponding to the sp_{lo} and sp_{hi} stockpile sizes defined in section II-F. Details of the datasets and measured performance are presented in table III.

Average allocation and deallocation costs are represented in number of CPU cycles required. The worst-case update scenario is a deallocation of one stockling, followed by an allocation of another. The amount of updates that can be carried out per second in the worst-case scenario is calculated by dividing the CPU speed with the sum of the allocation and deallocation costs.

Random incremental updates: In this experiment, we use the same three data sets as for the lookup performance measurements. First, all entries in each data set are inserted into the data structure. We then randomly choose one entry from the data set. If the entry is already present in the forwarding data structure, it is removed - otherwise it is inserted. This procedure is repeated 1,000,000 times.

The average cost for the incremental updates for each of the data sets are presented in table IV. The results excluding the supporting data structures measures the performance on the forwarding data structure and gives an approximation of how many random incremental updates per second our machine can support for the given data set. We have also measured the total cost of the updates - including the supporting data structure.

On our test machine, we can perform over 230,000 incremental worst-case updates per second. As a reference, 20,000 updates/second is suggested as ”ideal” in [14] and the largest observed update rates in the Internet today is on the order of 400 updates/second [15]. From these numbers is is tempting to conclude that 230,000 worst-case updates per second is by far good enough. However, it is hard to actually achieve these speeds as there are other components than our data structure involved in every update. The routing daemon must process incoming route update messages and maintain a routing table from which the forwarding table is constructed. There are also supporting mechanisms that act as an interface between the routing daemon and the forwarding data structure, generating the update events in our data structure. In a system-wide perspective, we do consider it feasible to design these components efficiently enough to support at least 20,000 updates per second.

In a single-CPU system, updates and lookups can not occur concurrently. Assuming that a worst-case update is roughly

TABLE V
LOOKUP/UPDATE RATIOS FOR UNCACHED LOOKUPS

	Dataset			
	London	**Mae-East**	**Pac-Bell**	**Mae-West**
Ratio	80	91	147	102

100 times slower than a worst-case lookup, an update speed of 20,000 updates per second would reduce the amount of lookups per second by approximately 2,000,000 - equivalent to a supported wire speed of 0.6 Gbit/s using our machine configuration. If these update rates were present in the Internet today, we believe that the reduction in lookup performance would be of minor importance as the update rates would suggest a pathological network problem. When 20,000 updates per second might be perfectly normal, we can always use multiple-CPU-systems to eliminate the reduction in lookup performance.

To estimate an average update cost in contrast to the measured worst-case average, we argue as follows. The writing cost for going to n basic intervals to $n+1$ in a subuniverse is $\sim n/9$. We can sum this over the densities $1 \ldots 2295$ to obtain a cost of 292740. We will traverse 23 different sizes of stocklings and know that the worst-case cost of each such transition is ≤ 752, which gives a cost of $752 \cdot 23 = 17296$. The total cost of $292740 + 17296 = 310036$ is amortized over 2295 updates, which gives an estimated average of 135 memory accesses per update.

At a first glance, the update speed achieved might seem surprising. However, since the average cost for an update is less than 135 memory accesses and the worst-case cost for a lookup is 4 memory accesses, an average update could potentially be as fast as 34 times the cost for a lookup. From our experiments we can compute the lookup/update ratio by dividing the number of uncached lookups per second with the number of incremental updates per second (excluding the support structure). The results are shown in table V.

As could be expected, the lookup/update ratio is smaller for larger tables except for Pac-Bell. The reason for this is that the prefixes are less evenly distributed in Pac-Bell compared to the other tables. In the regions of the table that contains prefixes of length > 16 bits, many prefixes are present while other regions contains virtually no prefixes. As a result, a random lookup is more likely to be completed in a single memory access while updates of random prefixes are likely to occur where the density of prefixes is high. Assuming that the average lookup cost in the London data set is around 2 memory accesses, the theoretically expected lookup/update ratio is 67, which is not far from the results in our experiments.

IV. DISCUSSION

In this section, we discuss some of the properties of the proposed algorithm and possible usage scenarios not covered in detail in this paper due to space restrictions. We also discuss how the algorithm perform in relation to the Luleå algorithm [2] by Degermark et. al. and Tree bitmaps [11] introduced by Eatherton et. al., which we consider to be the two algorithms closest to our approach.

A. Why guaranteed compression?

By designing the forwarding table with worst case memory requirements for a specific $N_{\max}$ we can guarantee that no more than 10 bytes per prefix are required for any routing table containing $\leq 2^{18}$ entries - not included the fixed amount of memory needed for storing the first-level array and the next-hop table. To our knowledge, there is no other LPM algorithm available which can guarantee such efficient compression even for pathological sets of routing entries.

One might argue that it is mildly interesting to provide a guaranteed compression ratio. However, as is rightfully pointed out in [11], the memory management is a very important part of an IP lookup engine. Without a guaranteed compression ratio for a given N_{max}, it is hard to bound the total memory consumption for any given - possibly pathological - routing table. We guarantee that for *any* routing table containing $\leq 2^{18}$ entries, our forwarding table will *never* consume more than 2.7 Mb of memory or use more than 4 memory accesses per lookup. These guarantees significantly simplifies system design in both software and hardware implementations.

B. Implementation in hardware

It would be rather straightforward to implement our forwarding table in hardware. Both direct indexing and the block tree search operations can easily be implemented as single cycle operations. We need to partition the memory into four memory banks corresponding to the four memory accesses in a way that guarantees that we will not run out of memory in any of the banks as long as the number of prefixes is below the design limit.

The size of the first memory bank is 8192 blocks since it only contains the 16 bit trie node. The second memory bank contain all level 3 block tree nodes and the 8 bit trie nodes. For a given design limit of $N_{\max}$ prefixes, the maximum number of blocks required for these block tree nodes is $2N_{\max}/135$ and the maximum number of blocks for the trie nodes is $(2N_{\max}/2296) \cdot 32$. Hence, the maximum number of blocks required in the second memory bank is $2N_{\max} \max\left(135^{-1}, 32/2296\right)$. In the third memory bank, we store level 2 block tree nodes from $(3, 16)$ block trees and $(2, 8)$ block trees. The maximum number of such nodes relative $N_{\max}$ is obtained from $(2, 8)$ block trees. In the worst case, we need to have $2N_{\max}/13$ blocks in the third memory bank but this can be reduced to $2N_{\max}/52$ by using quarter block trees. In the fourth memory bank we need at most $N_{\max}/9$ blocks. For a target $N_{\max}$ of 2^{18} routing entries, we then get

memory bank	**blocks**	**bytes**
1	8192	262144
2	7308	233856
3	10082	322624
4	58254	1864128
total	**83836**	**2682752**

This shows that it is feasible to implement the algorithm on chip using 300 MHz SRAM memory and four pipeline stages to obtain a lookup performance of 300 million lookups per second corresponding to ~ 100 gigabit per second wire speed forwarding using a routing table with up to 2^{18} routing entries. With sufficient pin count we can achieve a raw incremental update performance of $300,000,000/752 = 400,000$ updates per second. Considering what is possible to achieve with the supporting software package, 100,000 worst-case incremental updates per second seems realistic in a real application.

C. Comparison to other algorithms

Ever since Degermark et. al. and Waldvogel et. al. independently of each other presented the first efficient solutions to the IPv4 longest matching prefix problem in 1997 [2] [16], a considerable amount of different algorithms and data structures for the problem have been suggested. It is common to characterize the performance of such algorithms by the complexity of lookups, updates, and the memory consumption. [17] presents a survey of such algorithms where the performance charactistics are expressed in terms of *asymptotic behavior* using Big-O notation. While asymptotic behavior does not make sense when having a bounded universe, we adapt to the custom and present the corresponding characteristics of our algorithm in the same way. In principle, our algorithm have a *guaranteed performance* of $O\left(\frac{w}{k}\right)$ time for lookups, $O\left(\frac{b}{w}\,2^{\frac{w}{k}}\right)$ time for updates, and $O\left(N\right)$ space.

Many of the previous techniques have reasonable lookup performance but poor compression ratio and/or update performance. In particular, the guaranteed compression ratio is typically bad [14]. Some of the previous techniques are related to ours in the sense that similar constructs are used as building blocks in their data structures. In the approach of searching among hash tables [6], the multicolumn scheme [3] binary search is used in various contexts. This is also the case in [2] where a special type of branchless binary search exploiting pointer arithmetic is used to search sparse chunks. LC-tries [4] and Expanded tries [8] both use variable-stride multibit tries, as we do.

To our knowledge, however, no previous algorithm combines search techniques as efficient as block trees with multibit tries as carefully as we do to achieve the combination of guaranteed compression ratio, lookup speed and update performance. In fact, the first algorithm and data structure with similar characteristics to ours that we know of is the Tree Bitmap algorithm by Eatherton et. al. [11]. Both the Luleå algorithm [2] and the Tree Bitmap algorithm feature guaranteed lookup performance but also guaranteed compression ratio (even though it is not explicitly stated), and a relation between guaranteed lookup performance and compression is implicit in their results. In addition, the Tree Bitmap algorithm also supports efficient updates with guaranteed performance.

Both the Luleå Algorithm and Tree bitmaps combine index arrays and multibit tries to reduce the memory consumption. As they have similar performance characteristics to ours, we focus our comparison with other algorithms to these two.

Compression ratio: When compressing publicly available routing tables, both the Luleå algorithm and tree bitmaps typically achieve a better average compression ratio than our algorithm, but are more vulnerable to pathological sets of routing entries as they do not guarantee the same compression ratio. A forwarding table with comparable performance to ours can be implemented by initial array of size 2^{16} on top of a 6-6-4 tree bitmap data structures (which is really the best we can do with a block size of 256). Such a table may require more than 18 bytes per prefix (except for the initial array) and hence, considering guaranteed compression ratio, we outperform the tree bitmap algorithm with almost a factor of 2. A similar comparison with the Luleå algorithm shows that we outperform it by more than a factor of 2.

Incremental updates: If comparing the cost of incremental updates with tree bitmaps (the Luleå algorithm does not support incremental updates), Eatherton et. al. computes an upper bound on number of memory accesses for allocating 18 blocks to 1852 memory accesses. With a worst-case scenario of 752 memory accesses, we can perform incremental updates at least 2.46 times faster. It should be noted that this comparison is not entirely fair as we use 4 memory accesses for worst-case lookups while Eatherton's tree bitmaps use 6. For tree bitmaps to have a worst-case of 4 memory accesses, a stride of size 6 would be needed. An update would then require at least $2 \cdot (2^6)^2 = 8192$ memory accesses to rearrange the blocks, followed by a pointer update resulting in a few extra hundred memory accesses. This gives that with the same amount of worst-case memory accesses, we outperform tree bitmaps in update speed by at least an order of magnitude.

Large routing tables: It is also interesting to consider the difference in performance on large routing tables. Eatherton et. al. use a randomly generated routing table with 1,000,000 entries in their evaluation, resulting in a 23 Mbyte data structure. Although we neither know how their random table is generated nor have run any experiments including such large routing tables, it is possible to analytically predict the results. 2^{20} random prefixes of length ≥ 16 spread out over 2^{16} buckets in the root array gives an expected number of $2^4 = 16$ entries in each subuniverse. Using prefix lists with 4 bytes/prefix, our data structure would on average have a total size of $2^{18} + 2^{20} \cdot 4 = 4,456,448$ bytes. In worst-case, it would be $2^{18} + 10 \cdot n = 10,262,144$ bytes large.

D. Generalization of the algorithm

The techniques described in this paper could also be used to obtain an efficient IPv6 forwarding table data structure. [18] However, we have to spend some extra memory accesses to achieve a reasonable compression rate. By having more memory accesses to spend, the block tree structures tends to become larger thus increasing the costs for incremental updates. Some preliminary investigations indicate that a possible remedy to this problem is to combine a fixed stride trie and block trees with the tree bitmap structures.

We have used the algorithm and the data structure to implement longest prefix matching but it could also be used for

other more general matching problems such as *most narrow interval* and *first matching interval*. This suggests that many of the techniques described here could be used to implement efficient filtering and packet classification engines. It might be cost-efficient enough to compete with ternary CAMs when considering lookup performance, energy consumption and, in particular, update performance.

Stockpiling can be used for memory managements also in other algorithms and data structures. As an example, we could the reduce the cost of incremental updates in tree bitmaps from 1852 to ≤ 263 memory accesses. However, tree bitmaps would still use more memory accesses per lookup.

V. Conclusion

We have introduced a high-performance data structure for IPv4 forwarding that provides a guaranteed compression ratio while supporting high-speed incremental updates keeping the number of memory accesses needed for a lookup down. In our configuration, any set of routes with $\leq 2^{18}$ entries can be stored within 2.7 Mb of memory. In worst case, lookups need 4 memory accesses and incremental updates 752 memory accesses. On a 3.00 Ghz Pentium machine, this corresponds to supported wire speeds exceeding 10 Gbit/s and more than 230,000 updates/second. The data structure could be implemented in hardware and extended to support alternative usage scenarios supporting larger routing tables, IPv6 forwarding and interval matching. Future work includes applying similar techniques as used in this paper to new usage scenarios.

References

[1] Gene Cheung and Steven McCanne, "Optimal routing table design for IP address lookups under memory constraints," in *INFOCOM (3)*, 1999, pp. 1437–1444.

[2] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink, "Small forwarding tables for fast routing lookups," in *SIGCOMM*, 1997, pp. 3–14.

[3] Butler Lampson, Venkatachary Srinivasan, and George Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 324–334, 1999.

[4] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE Journal on Selected Areas in Communications*, June 1999.

[5] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Transactions on Computer Systems*, vol. 17, no. 1, pp. 1–40, 1999.

[6] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, "Scalable high-speed prefix matching," *ACM Transactions on Computer Systems*, vol. 19, no. 4, Nov. 2001.

[7] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian, "Delayed internet routing convergence," in *Proceedings of SIGCOMM*, Stockholm, Sweden, September 2000, pp. 175–187.

[8] V. Srinivasan and George Varghese, "Faster IP lookups using controlled prefix expansion," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. 1998, pp. 1–10, ACM Press.

[9] V. Srinivasan, *Fast and Efficient Internet Lookups*, Ph.D. thesis, Washington University, 1999.

[10] Mikael Sundström, "Block Trees - an Implicit Data Structure Supporting Efficient Searching among Interval Endpoints," Tech. Rep., Luleå University of Technology, 2005.

[11] Will Eatherton, George Varghese, and Zubin Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates," *SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 97–122, 2004.

[12] Svante Carlsson and Mikael Sundström, "Linear-time in-place selection in less than 3n comparisons," in *Algorithms and Computation, 6th International Symposium, ISAAC'95, Cairns, Australia, December 4-6, 1995, Proceedings*, vol. 1004 of *Lecture Notes in Computer Science*, pp. 244–253. Springer, 1995.

[13] RIPE, "Routing information service (RIS)," nov 2003, http://www.ripe.net/ripencc/pub-services/np/ris/index.html.

[14] Pankaj Gupta, *Algorithms for Routing Lookups and Packet Classification*, Ph.D. thesis, Stanford University, 2000.

[15] Lan Wang, Xiaoliang Zhao, Dan Pei, and Randy Bush et. al., "Observation and analysis of BGP behavior under stress," in *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment*. 2002, pp. 183–195, ACM Press.

[16] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, "Scalable high speed IP routing table lookups," in *ACM SIGCOMM '97*, Sept. 1997, pp. 25–36.

[17] M. Ruiz-Sanchez, Ernst Biersack, and Walid Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8–23, 2001.

[18] Mikael Sundström and Lars Åke Larzon, "Design and Tuning of Classifiers for IPv6-length Addresses with Guaranteed Lookup Performance, Compression Ratio and Support for Incremental Updates," Submitted for review.