Optimal Routing Table Design for IP Address Lookups Under Memory Constraints

Gene Cheung and Steve McCanne
Department of EECS
University of California, Berkeley
Berkeley, CA 94720

Abstract— **The design of lookup tables for fast IP address lookup algorithms using a general processor is formalized as an optimization problem. A cost model that models the access times and sizes of the hierarchical memory structure of the processor is formulated. Two algorithms, using dynamic programming and Lagrange multipliers, solve the optimization problem optimally and approximately respectively. Experimental results show our algorithms have visible improvements over existing ones in the literature.**

I. INTRODUCTION

In the Internet Protocol (IP) architecture, *hosts* communicate with each other by exchanging packetized messages or *packets* through a fabric of interconnected forwarding elements called *routers*. Each router consists of input interfaces, output interfaces, a forwarding engine, and a routing table. When an IP packet arrives on an input interface, the forwarding engine performs a *route lookup* — it locates an entry from the routing table based on the contents of the packet's IP header (typically the destination address) and uses this entry to determine the output interface through which to forward the packet toward its ultimate destination.

If forwarding performance within a router were infinitely fast, then the overall performance of the network would be determined solely by the physical constraints — i.e. bit rate, delay, and error rate — of the underlying communication links. However, in practice, building high-speed routers is a hard problem and as physical-layer link speeds continue to increase, the forwarding path of the router can easily become the bottleneck in network performance. This is especially true for routers situated deep within the backbone of the network, where very high traffic volumes require very high-speed processing. Thus, to optimize network performance, we must not only optimize the speed of the physical-layer transmission media but also the forwarding performance of the routers.

While the forwarding path in a high-speed router consists of many important stages, each posing a potential bottleneck, the *route lookup* stage, in particular, poses a unique challenge because its performance is sensitive to the size of the routing table. And as IP networks like the Internet scale to very large and interesting sizes, the routing tables in the backbone routers can become large and thus pose a serious performance problem. That is, as the number of potential destinations known to a router grows, the complexity of managing this state and of forwarding packets against this state increases.

Fortunately, the Internet design anticipates this scaling problem and consequently imposes hierarchy onto the IP address space. Because of this, every router need not have a routing table entry for every possible network destination. Instead, address-space hierarchy can be exploited to achieve address aggregation whereby a large collection of hosts can be addressed as a single entity via a variable length network address [3]. For example, all of the hosts at U.C. Berkeley can be referred to simply as 128.32/16 because they all have the same 16-bit prefix of "128.32". As a result, routers have much smaller routing tables and the problem of searching this table for a given route on a per-packet basis is simplified.

While on the one hand, a smaller routing table theoretically simplifies the problem of searching for a given route, on the other, the variable width of network addresses adds complexity to the lookup algorithm. Though the lookup rule is relatively straightforward — the route that matches the longest prefix of the destination address is chosen — very fast algorithms that implement this rule are not at all obvious nor are they straightforward to implement. In light of this, a growing body of work has addressed the problem of fast IP route lookup [1], [2], [4], [5], [6], [7]. [4] is an early well-known implementation of address lookup, which uses PATRICIA Tree [11]. [6] discussed a hardware implementation of address lookup. [7] proposed a novel adaptation of binary search for address lookup. The other papers discussed implementations of address lookup that use lookup tables — this is the approach we pursue in this paper.

Conceptually, a route lookup can be cast as a table lookup, where a single table is indexed by the destination address to yield the routing entry. Since a lookup operation takes constant time, this simple algorithm has complexity $O(1)$. However, the sheer size of the IP address space precludes a single table approach since such large memories are impractical, expensive and slow. Alternatively, we can represent the routing table as a binary tree and decode the address one bit at a time. But this is inefficient as it requires iterative, sequential processing of the address bits.

Recent work combines the tree representation with table-driven lookups to enhance performance [1], [2]. The idea is to compute a representation of the tree as a set of tables where forwarding is accomplished by traversing the routing table tree via multiple table lookups. The challenge then is to devise an algorithm when given a set of routes, produces a "good" set of tables. If the tables are not chosen intelligently, performance can suffer – when tables are too large, they cannot fit into small, fast, memories, and when too small, the algorithm must perform extra lookups.

To choose the tables layouts intelligently, we have developed a model for table-driven route lookup and cast the table design problem as an optimization within this model. Suppose we have three types of hierarchical memories, with memory size and access speed (S_1, T_1), (S_2, T_2) and (S_3, T_3) respectively. Suppose each prefix j has an associated probability p_j. Assuming independent IP packet arrival, the average prefix retrieval time is:

$$C = \sum_j p_j(a_j T_1 + b_j T_2 + c_j T_3) \quad (1)$$

0-7803-5417-6/99/$10.00 ©1999 IEEE.

Prefix	Interface number	Probability
111	5	.2
110	4	.3
1	3	.2
01	2	.2
00	1	.2

Fig. 1. Example of a set of Prefixes

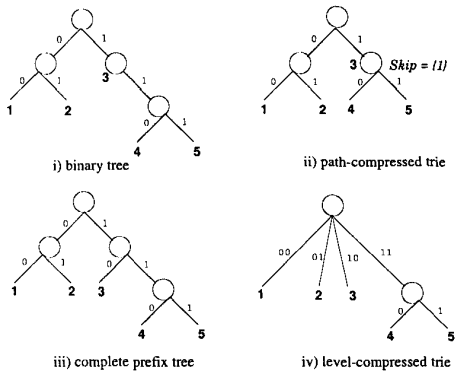


Fig. 2. Example of Prefix Data Structures

where a_j is the number of type 1 memory access needed to retrieve prefix j, b_j is the number of type 2 memory access needed, c_j is the number of type 3 memory access needed. a_j is basically the number of lookup tables residing in type 1 memory used in the retrieval process of prefix j. (1) is correct only if the tables assigned to each type of memory do not exceed the capacity of that type of memory. The problem is how to structure a set of tables that minimizes (1) while satisfying the memory size constraints.

Although previous works on fast IP route lookup propose elegant designs for their table lookup schemes, no approach methodically attempts to design an optimal strategy along these lines [1], [2]. A notable exception is more recent work by Srinivasan and Varghese [5], but this work optimizes the worst case rather than the average case.

In this paper, we present a general framework for generating optimal tables for table-driven IP routing. We first present a brief introduction to the data structures used in IP address lookups. We then present a pseudo-polynomial time algorithm that gives the optimal table design given our model. One can prove that finding the optimal set of tables is an NP-complete problem [8], and thus we devised an approximate algorithm that runs in polynomial time and provides performance within a constant bound of the optimal. Next, we present performance results of our algorithm. Finally, we discuss directions for future work and conclude.

II. Overview of Data Structure

Many efforts have been made in an attempt to speed up IP route lookups, and the key distinguishing factor across these various approaches is the data structure employed to represent the routing table. In this section, we give a brief introduction to these structures, and describe in depth the structure we use — generalized level-compressed trie. Based on our generalized data structure, we then formalize our optimization problem to provide the context for the algorithms presented in later sections.

A. Binary Tree, Trie, PATRICIA Tree, and Complete Prefix Tree

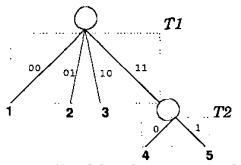
A *Binary tree* is one of the simplest representation of a set of prefixes. Each prefix is represented by a node in the tree, usually a leaf, and the path from the root to that node reveals the associated bit pattern of the prefix. For example, Figure 1 shows a set of prefixes and their associated interface numbers and probabilities, and Figure 2(i) shows the binary tree representation of this set. Each node of the tree has at most two children, hence binary. A *trie* [12] is a more general data structure, where each node can have any number of children. A binary tree then, also called binary trie, is a special case of trie. The implementation of binary tree involves bit-by-bit manipulation, which is inefficient for long prefixes.

A *path-compressed trie* is a trie where certain sequences of input bits may be skipped over before arriving at the next node. Compression is evident when a long sequence of one-child nodes needs to be traversed before arriving at a sub-trie of prefixes. Figure 2(ii), for instance, shows a path-compressed trie version of the same set of prefixes. Path-compressed binary trie, also known as *PATRICIA Tree* [11], is used by many early implementations of IP route lookups [4] and provides reasonable performance. A problem with path compression is that if the skipped bits do not match the assumed path, then backtracking is required to retreat back to the pre-path-compressed node, thereby incurring computational overhead. To eliminate backtracking, a binary tree can be first converted to a *complete prefix tree* [1], which is a tree where each node has either two children or none. Intuitively, a complete prefix tree contains a set of prefixes that are *prefix-free*, i.e. no prefix is a prefix of another. When traversing a complete prefix tree to retrieve a prefix, only a leaf terminates a search, and so no backtracking is needed. A binary tree of prefixes can always be converted to a complete prefix tree using techniques in [1] or [5]. For example, Figure 2(iii) shows the complete prefix tree representation of our set of prefixes.

A *level-compressed trie* [2] is a trie in which each complete subtree of height h is collapsed into a subtree of height 1 with 2^h children. For example, starting at node i, if there are exactly 2^h descendent nodes at level h, then all the intermediate nodes from level 1 to level $h-1$ can be replaced by 2^h branches originated directly from node i. Figure 2(iv) shows a level-compressed trie version of the set of prefixes from the complete prefix tree in Figure 2(iii). A level-compressed trie can be implemented using lookup tables, where for each table, a length h bit sequence is extracted and used as index. h is called the *stride* of the trie.

B. Generalized Level-Compressed Trie

Intuitively, a section of the tree needs not be full to be level-compressed. For example, a set of prefixes $\{00, 01, 1\}$ can be expanded to $\{00, 01, 10, 11\}$ with prefixes 10 and 11 containing the same information as the original prefix 1. Then a trie of stride 2 can be used, implemented as a 2-bit indexed lookup table. This technique is called *prefix expansion* in [5]. It is also



a) generalized level-compressed trie

Prefix	a	b	c
111	1	1	0
110	1	1	0
10	1	0	0
01	1	0	0
00	1	0	0

b) memory assignment parameters

Fig. 3. Example of a Configuration

used as a part of the data structure introduced in [1], although [1] took a more implementational approach instead and did not formulate it as an optimization problem. This notion of generalized level compression provides an extra degree of freedom in creating lookup tables.

For our lookup table data structure, we make the following design decisions: Given a set of prefixes represented by a binary tree, *stage 1* transforms it to a complete prefix tree. Then from the complete prefix tree, *stage 2* transforms it to a generalized level-compressed trie structure, which we implement using lookup tables. The approach is advantageous for two reasons: first, the transformation to complete prefix tree avoids costly backtracking; second, although both level-compressed tries and generalized level-compressed tries can be efficiently implemented as a lookup table tree, generalized level-compressed tries are more general and therefore more flexible than level-compressed tries. The stage 1 transformation is straightforward using techniques in [1] or [5]. The stage 2 transformation, however, affects the lookup speed of prefixes directly and must be done carefully; this is the focus of our optimization problem.

The problem can now be formalized as follows: given a complete prefix binary tree, what is the form of generalized level-compressed trie, so that when cast onto hierarchical memories, minimizes the average lookup time per prefix, defined in (1)? We will tackle this problem in the next two sections.

As an example of generalized level-compressed tries, consider the set of prefixes in Figure 1 and the memory constraints: $(S_1, T_1) = (4, 1)$, $(S_2, T_2) = (4, 2)$ and $(S_3, T_3) = (100, 5)$. A *configuration*, defined as a particular instance of generalized level-compressed trie complete with table assignments to memory types, is shown in Figure 3. The configuration has two lookup tables: one of height 2 and size 4, another of height 1 and size 2. They are assigned to memory type 1 and type 2 respectively. The corresponding set of parameters, a's, b's, c's, are shown in Figure 3*b*. Notice the memory size constraints are satisfied. The cost of this configuration is 1.6. Using this configuration, we can rewrite (1) as the following:

$$\begin{aligned} C &= (p_1 + p_2 + p_3 + p_4 + p_5)T_1 + (p_4 + p_5)T_2 \\ &= w_1 T_1 + w_2 T_2 \end{aligned} \quad (2)$$

where $w_1 = p_1 + \cdots + p_5$ is defined as the *weight* of the first table, and $w_2 = p_4 + p_5$ is the weight of the second table. w_i is basically the probability mass of subtree rooted at i. In short, we can write the cost function as a linear sum of scaled weights of the configured lookup tables. The terminology will become useful in the algorithm development sections.

III. Dynamic Programming Algorithm

Having defined the optimization problem, we now present two algorithms that solve this problem. In this section, we develop a pseudo-polynomial algorithm that finds the optimal solution using dynamic programming. However, because our problem can be reduced to the 3D matching problem, it is NP-complete [8]. Thus in the next section, we present an approximate solution that runs in polynomial time.

Out optimal algorithm will bear much resemblance to the knapsack dynamic programming algorithm [10]. We consider the problem with two memory types; a generalization to three memory types is straightforward and thus omitted. In section III-A, we discuss the development of this algorithm, and in section III-B, detail its performance bound.

A. Development of Algorithm

Recall the optimization problem is: given a complete binary tree, what is the optimal generalized level-compression transformation performed on the tree that minimizes average prefix lookup time, expressed in (1)? The overview of our algorithm is the following. We first need to decide how many levels to compress from the root of the tree. Knowing the number of compression levels, we create a lookup table and decide if this table should go in type 1 (fast) or type 2 (slow) memory. If we put this table in type 1 memory, then there will be less type 1 memory left for the descendent sub-trees at the bottom of this table. In either case, however, we must decide how to divvy up the remaining type 1 memory among the descendent sub-trees. The optimal solution is the one that yields the minimum average lookup time among all these decisions.

Let $TP_1(i, s_1)$ be a recursive "tree-packing" cost function that returns the minimum cost of packing a tree rooted at node i into type 1 memory of size s_1, and type 2 memory of size infinite. Rooted at i, a table of height h will have weight w_i and size 2^h. We have two choices regarding this table. We can either put it in type 1 memory, resulting in cost $w_i T_1$ and $s_1 - 2^h$ remaining type 1 memory space for the rest of the tree. Alternatively, we can put it in type 2 memory, resulting in cost $w_i T_2$ and s_1 remaining type 1 memory space. $TP_1(i, s_1)$ is the minimum of these two choices, for all possible height h. We can formalize this logic recursively as follows:

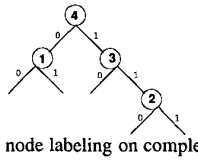
$$\begin{aligned} TP_1(i, s_1) = \min_{1 \le h \le H_i} \Big\{ &\min [w_i T_1 + TP_2(L_{h,i},\, s_1 - 2^h), \\ &w_i T_2 + TP_2(L_{h,i},\, s_1)] \Big\} \end{aligned} \quad (3)$$

where H_i is the height of tree rooted at node i, $L_{h,i}$ is the set of internal nodes at height h of tree rooted at node i, and $TP_2(L, s_1)$ is a sister "tree-packing" cost function that returns the minimum cost of packing a set of trees of root nodes $\{i \in L\}$ into type 1 memory of size s_1. To complete the analysis, we will also need the following base cases:

$$\begin{aligned} TP_1(.,-) &= \infty, \quad TP_2(.,-) = \infty \\ TP_2(\{\},.) &= 0 \end{aligned} \quad (4)$$

The first base case says if the memory constraint is violated, i.e. a negative value as the second argument, then this configuration is not valid and cost is infinite. The second base case says if the first argument is the empty set, then the cost is zero.

Note that in (3), TP_1 calls TP_2, a function with possibly multiple node argument, $L_{h,i}$; we need to provide a mean to solve TP_2. If the set of nodes L contains only one node, then by definition TP_2 is the same as TP_1. If the set has more than one node, we first split up the set L into two disjoint subsets, L_1 and



a) node labeling on complete prefix tree

TP	0	2	4
$TP_1(4,.)$			1.6
$TP_1(1,.)$	0.8	0.4	0.4
$TP_1(3,.)$	1.2	1.2	0.6
$TP_1(2,.)$	0.6	0.3	0.3
$TP_2(\{1,3\},.)$	2	1.6	1.4

b) dynamic programming table

Fig. 4. Example of Dynamic Programming

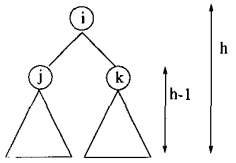


Fig. 5. Counting Rows of DP table for Full Tree Case

L_2. We then have to answer two questions: i) given s_1 available type 1 memory space, how do we best divvy up the memory space between the two subsets, ii) given the division of memory space, how do we perform generalized level-compression on the two subsets of nodes to yield the overall optimal solution. The optimality principle states that a globally optimal solution must also be a locally optimal solution. So if we give s memory space to set L_1, to be locally optimal, optimal generalized level-compression must be performed on that set, returning a minimum cost of $TP_2(L_1, s)$. That leaves $s_1 - s$ memory space for L_2, with minimum cost of $TP_2(L_2, s_1 - s)$. The globally optimal solution is the minimum of all locally optimal solutions, i.e. the minimum of all possible value of s. We can express this idea in the following equation:

$$\begin{aligned} TP_2(L, s_1) &= TP_1(i, s_1) \qquad \text{if } |L| = 1,\ L = \{i\} \\ TP_2(L, s_1) &= \min_{0 \le s \le s_1} \{TP_2(L_1, s) + TP_2(L_2, s_1 - s)\} \\ &\quad \text{s.t.} \quad L_1 \cap L_2 = \{\}, \quad L_1 \cup L_2 = L \end{aligned} \tag{5}$$

There are many ways to split up the set L into subsets L_1 and L_2. For efficiency reason soon to be discussed, we will divide in such a way that all nodes in L_1 has a common ancestor, all nodes in L_2 has a common ancestor, and that the two ancestors of the two sets are distinct (if possible) and on the same level of the tree.

A.1 Dynamic Programming Table

We argue that TP_1 (TP_2) can be computed using a dynamic programming algorithm because its structure leads to overlapping subproblems. Each time the function TP_1 (TP_2) is called with argument (a, b), the algorithm can check the (a, b) entry of a dynamic programming table to see if this subproblem has been solved before. If it has, the function returns the value from the table. If not, it solves it using (3) and (5), inserts the value in the table and returns the value. This way, a given subproblem is solved only once. The reason for the method of dividing set L in (5) is now clear: it maximizes the number of overlapping subproblems. As an example, our complete prefix tree, reproduced in Figure 4(a) with node labels, is used as input to this algorithm. Assuming $(S_1, T_1) = (4, 1)$ and $(S_2, T_2) = (\infty, 2)$, the resulting dynamic programming table (DP table), resulted from call to TP to root node 4 as argument is shown in Figure 4(b). Note the overlapping subproblem in this case: for node 4, $TP_1(4, 4)$ calls $TP_1(2, 0)$ when $h = 2$. However, $TP_1(2, 0)$ has been solved from previous call to $TP_1(3, 2)$ when $h = 1$ already.

B. Analysis of Algorithm

We now show the running time of the algorithm is $O(HnS_1{}^2)$, where H is the height of the complete binary tree, n is the number of nodes, and S_1 is the size of type 1 memory. The algorithm runs in pseudo-polynomial time, i.e. although the complexity of the algorithm looks polynomial, its running time is actually exponential in the size of the input.

Since this is a dynamic programming algorithm, the running time is the amount of time it takes to construct the DP table. Suppose we label the rows of the table using the first argument of the TP functions, as illustrated in Figure 4(b). There are two types of row labels: single-node labels and multiple-node labels. The number of single-node labeled rows is simply the number of nodes, n. The number of columns is at most S_1, and therefore the size of the half of the table with single-node labeled rows is $O(nS_1)$. To solve each entry in this half of the table, we use (3) and need at most H_i comparisons. Let the height of the complete prefix tree be H. (Since IPv4 address is 32 bits long, height of the tree is at most 32.) Therefore the running time for this half of the table is $O(HnS_1)$.

To account for the half of the table with multiple-node labeled rows, we need to bound the number of these rows. The tree that maximizes the number of multiple-node rows in a DP table for a fixed number of nodes is a full tree, since each call from (3) to its children nodes at height h will result in a multiple-node argument. We will count the number of multiple-node rows for full tree as follows. Let $f(h)$ be a function that returns the number of multiple-node labeled rows in DP table for a full tree of height h. Suppose node i is the root of that full tree, with children nodes j and k, as shown in Figure 5. Since node j and k have height 1 less than i, they will each contribute $f(h-1)$ to the multiple-node labeled rows of DP table. In addition, as the call to $TP_1(i, S_1)$ swings h from 1 to $H_i - 1$, each new call to TP_2 with root node argument $L_{h,i}$ creates a new row. Therefore the total number of multiple-node labeled rows is:

$$\begin{aligned} f(h) &= 2 * f(h-1) + h - 1 \\ f(1) &= 0 \end{aligned} \tag{6}$$

After rewriting (6), we get:

$$f(h) = 2^{h-1} \sum_{i=1}^{h-1} \frac{i}{2^i} \tag{7}$$

Notice the summation in (7) converges for any h and can be bounded above by 2. Therefore the number of multiple-node labeled rows is $O(2^h)$. Since the tree is full, $h = \lceil \log_2 n \rceil$. Therefore the number of multiple-node labeled rows for a full tree is $O(2^{\log_2 n}) = O(n)$. The size of this half of the table is again $O(nS_1)$, and each entry takes at most S_1 comparisons using (5). So the running time of a full tree, for the half of the table with multiple-node labeled rows, is $O(nS_1{}^2)$. Since this is the worst case, the general tree with n nodes will also have running time $O(nS_1{}^2)$ for this half of the table. Combining with the part of the table with single node labeled rows, the running time is $O(HnS_1{}^2)$.

IV. Lagrange Approximation Algorithm

For non-trivial type 1 memory sizes, the DP algorithm given above may be impractical. Thus, to improve on the executing time, we developed an approximate algorithm with bounded error that runs in polynomial time. In section IV-A, we present a high-level description of the algorithm, and in section IV-B, discuss the performance bound of the algorithm. Proofs of correctness of the Lagrange approximate algorithm is provided in the Appendix.

A. Development of Algorithm

Our algorithm is based on an application of Lagrange multipliers to discrete optimization problems with constraints, similarly done in [9] for bit allocation problems. Instead of solving the original constrained table lookup problem, we solve the dual of the problem, which is unconstrained. The unconstrained problem is in general much easier to solve, but it needs to be solved multiple times, each time with a different multiplier value. The multiplier value is adjusted for each iteration until the constraint of the original problem is met.

A.1 Lagrange Multipliers for the Table Lookup Problem

We formulate our original problem as a constrained optimization problem which we can recast as follows:

$$\min_{b \in B} H(b) \quad \text{s.t.} \quad R(b) \leq S_1 \tag{8}$$

where B is the set of possible configurations, $H(b)$ is the average lookup time for a particular configuration b, $R(b)$ is the total size of type 1 table sizes, and S_1 is the size of type 1 memory. The corresponding dual problem is:

$$\min_{b \in B} H(b) + \lambda R(b) \tag{9}$$

Because the dual is unconstrained, solving the dual is significantly easier than solving the original problem. However, there is an additional step; we must adjust the multiplier value λ so that the constraint parameter, $R(b)$, satisfies (8).

In (9), the multiplier-rate constraint product, $\lambda R(b)$, represents the penalty for placing a lookup table in type 1 memory, and is linearly proportional to the size of the table. For each λ, we solve the dual problem, denoted as LP, as follows. For each table rooted at node i and height h, we have two choices: i) place it in type 1 memory with cost plus penalty $w_i T_1 + 2^h \lambda$; or, ii) place it in type 2 memory with cost $w_i T_2$. The minimum of these two costs plus the recursive cost of the children nodes will be the cost of the function at node i for a particular height. The cost of the function at node i will then be the minimum cost for all possible height, which we can express as follows:

$$LP_1(i, \lambda) = \min_{1 \leq h \leq H_i} \left\{ LP_2(L_{h,i}, \lambda) + \min[w_i T_1 + \lambda 2^h, w_i T_2] \right\} \tag{10}$$

where LP_1 and LP_2 are sister functions defined similarly to TP_1 and TP_2. Because we are optimizing with respect to λ which does not change for each iteration, for LP_2 with multiple-node argument, it is simply the sum of calls to LP_1 of individual nodes:

$$LP_2(L, \lambda) = \sum_{j \in L} LP_1(j, \lambda) \tag{11}$$

Similar to TP_2, we can alternatively solve it using recursion:

$$\begin{aligned} LP_2(L, \lambda) &= LP_1(i, \lambda) \qquad \text{if } |L| = 1,\ L = i \\ LP_2(L, \lambda) &= LP_2(L_1, \lambda) + LP_2(L_2, \lambda) \\ &\quad \text{s.t.} \quad L_1 \cap L_2 = \{\}, \quad L_1 \cup L_2 = L \end{aligned} \tag{12}$$

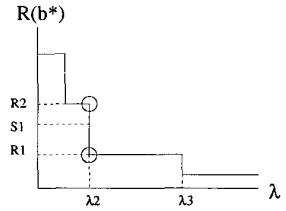


Fig. 6. Sum of Sizes of type 1 memory assigned Tables vs. Multiplier

After solving the dual problem using (10) and (12), we have an "optimal" configuration of tables from the prefix tree, denoted by b^*, that minimizes the dual problem for a particular multiplier value. The sum of sizes of tables assigned to type 1 memory will be denoted $R(b^*)$. If $R(b^*) = S_1$, then the optimal solution to the dual is the same as the optimal solution to the original problem. If $R(b^*) < S_1$, then the optimal solution to the dual becomes an approximate solution with a bounded error — in general, the closer $R(b^*)$ is to S_1, the smaller the error is.

The crux of the algorithm is to find λ such that the memory size constraint is met, i.e. $R(b^*) \leq S_1$. In general, for a multiplier problem, the constraint variable, $R(b^*)$ in our case, is inversely proportional to the multiplier λ, as shown in Figure 6. Therefore, a simple strategy is to search for the appropriate multiplier value using binary search on the real line to drive $R(b^*)$ towards S_1. As shown in Figure 6, the rate constraint-multiplier function is a decreasing step function – a consequence of the fact that the optimization problem we have is discrete, not continuous. For our implementation, we terminate the search for a new multiplier when we reach a horizontal portion of the curve, i.e. the new multiplier fails to yield a new rate $R(b^*)$.

B. Analysis of Algorithm

The actual running time of the algorithm depends on the data set. However, we can estimate the average running time of the algorithm by performing the following analysis.

We first estimate the execution time to solve (9 for each multiplier value λ. This is again a dynamic programming problem, where the DP table has only one column. Each entry i of the DP table contains the value $LP_1(i, \lambda)$ ($LP_2(L, \lambda)$). For single node arguments, computing an entry i in the table needs at most H_i operations using (10). Let the height of the complete prefix tree be H. (Again, for IPv4's 32-bit address, $H \leq 32$.) Since there are n single nodes, it takes $O(Hn)$ for each multiplier value for this half of the table. For multiplier node arguments, we know the number of entries is $O(n)$ from analysis from previous section. In this case, computing each entry takes constant number of operations using (12). Therefore it takes $O(n)$ for this half of the table. Combining the two halves, it takes $O(Hn)$. Let A be the number of iterations of binary search for multiplier values needed to be performed. Then the complexity is $O(HnA)$. In experimental results, A is found to be around 10. Considering n is in the neighborhood of tens of thousands, and comparing this complexity to the optimal algorithm, we see a substantial improvement in execution time.

V. IMPLEMENTATION

After we have obtained a configuration – a set of lookup tables and the mapping of tables to different memories – from either the optimal dynamic programming algorithm or the Lagrange approximate algorithm, we need to implement the lookup algorithm on the native platform. On architectures where cache movements can be explicitly made by native code, we can map tables directly to cache spaces via configuration's table assignments. On architectures where such cache control is not available, then our cost model becomes an approximation, since cache movements are less predictable. However, note that if our optimization is done correctly, then the tables with the largest weights will be assigned to faster memory. Correspondingly, tables with the largest weights will be most frequently accessed, and therefore will most likely be in faster memory. Therefore our cost model is still a good approximation. Nevertheless, we make the following minor adjustments to our algorithms to better approximate cache movements in such architectures.

A. Light/Dark Entry of Lookup Tables

In general, a lookup table of size 2^h may have many entries that have zero or close to zero probability of being accessed. The prefixes that correspond to those entries are prefixes of erroneously routed packets – packets with destination addresses that the router does not know where to forward with the current routing table. These packets are subsequently dropped. We denote these entries in the lookup tables as *dark entries*. On the one hand, these entries must exist so that the router can recognize them when they appear. On the other, their occurrences are so infrequent that it would be unwise to include them in our optimality calculation. In this section, we discuss a simple modification to our developed algorithms to disregard these dark entries, and consider only their counterpart, *light entries*.

A.1 Reformulation of Algorithms

Suppose we create a table of height h from node i, with 2^h total entries. Because only the light entries are accessed frequently, the size of the table that gets moved around in hierarchical memories will only be the number of the light entries, while the dark entries reside in the slowest memory. Let $d(i,h)$ denote the dimension, or number of the light entries of a height h table rooted at node i. In general, this number is upper bounded by 2^h and lower bounded by 0.

For our optimal dynamic programming algorithm, when we are considering whether to place a table in type 1 or 2 memory, the size of the table in question will be $d(i,h)$ instead of 2^h. The modified equation will be the following:

$$TP_1(i,s_1) = \min_{1\le h\le H_i} \Big\{ \min\big[\, w_i T_1 + TP_2(L_{h,i},\, s_1 - d(i,h)),\; w_i T_2 + TP_2(L_{h,i},\, s_1)\,\big] \Big\} \quad (13)$$

Similarly, for our Lagrange approximation algorithm, the penalty for placing a table in type 1 memory is no longer $\lambda 2^h$, but instead $\lambda d(i,h)$. The following modified equation is the result.

$$LP_1(i,\lambda) = \min_{1\le h\le H_i} \Big\{ LP_2(L_{h,i},\,\lambda) + \min\big[\, w_i T_1 + \lambda d(i,h),\; w_i T_2 \,\big] \Big\} \quad (14)$$

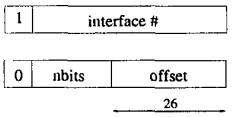


Fig. 7. Array Element Layout

The questions that remains are: i) how to find $d(i,h)$ efficiently; and, ii) does the calculation of $d(i,h)$ increase the order of complexity of the algorithms. We will answer these questions next.

A.2 Tabulation of Light Entries

We will again find the number of light entries of tree rooted at i and height h, $d(i,h)$, using a dynamic programming algorithm. Let the left branch of node i represent the path to all prefixes with 0 as left-most bit, and right branch represent the path to prefixes with 1 as left-most bit. For the base case when $h = 1$, the table has 2 total entries. We first check if the left branch has probability greater than zero. If so, then this entry is light. Similarly, we check the right branch. $d(i,1)$ is the sum of these two checks. If $h > 1$, then we solve this recursively. If there is a left child node, j_L, then we recursively call $d(j_L, h-1)$ to find the number of light entries for the left side. If there is no left child node, i.e. it is a leaf, then we check if the leaf has probability greater than zero. If so, we scale it by 2^{h-1} to account for the prefix expansion of height h. A similar procedure is done for the right side. The total number of light entries is the sum of the left and the right side. Let $\mathbf{1}(c)$ be the indicator function that returns 1 if clause c is true, and 0 otherwise. We can then use the following equation to express this analysis:

$$d(i,h) = \begin{cases} \mathbf{1}(p_L > 0) + \mathbf{1}(p_R > 0) & \text{if } h = 1 \\ d(j_L, h-1) + d(j_R, h-1) & \text{if } h > 1, \exists j_L, j_R \\ d(j_L, h-1) + \mathbf{1}(p_R > 0) * 2^{h-1} & \text{if } h > 1, \exists j_L, \nexists j_R \\ \mathbf{1}(p_L > 0) * 2^{h-1} + d(j_R, h-1) & \text{if } h > 1, \nexists j_L, \exists j_R \\ \mathbf{1}(p_L > 0) * 2^{h-1} + \mathbf{1}(p_R > 0) * 2^{h-1} & \text{if } h > 1, \nexists j_L, j_R \end{cases} \quad (15)$$

where j_L, j_R are the left and right child node of node i, and p_L, p_R are the left and right branch probability of node i.

We now show that the tabulation of $d(i,h)$ does not increase the order of complexity of our algorithms. Each time the value $d(i,h)$ is needed, the function first checks if the value is in the (i,h) entry of the dynamic programming table. If so, it simply returns the value. If not, it tabulates it using (15), stores it in the DP table, and returns it. We assume every entry of the DP table maybe used, so the complexity is again the time required to construct the entire DP table. The size of the DP table is the number of nodes by the maximum height H, which is again 32 for IPv4 32-bit address. The time required to tabulate each entry of the table, using (15), is a single addition. Therefore the complexity is $O(Hn)$. This clearly does not increase the order of complexity of our algorithms.

B. Encoding of Table Entries

We now describe the mapping scheme of lookup tables to memory, and the encoding scheme we use for each entry of the lookup tables. We first define an array p, large enough to contain all the elements in all the tables. For all the tables that are assigned to first level memory, we map the tables onto the array in breadth-first order, starting at the root of the tree. This will ensure that all first level assigned tables are in contiguous memory, and that each first level assigned mother-child table pairs are

```
#define match(A)
  nbits = startNbits;
  pa = &p[0];
  while (true)    {
        code = *(pa + (A >> (32 - nbits)));
        if (code < 0)
             break;
        A <<= nbits;
        nbits = code >> 26;
        pa = &p[0] + (code & ((1<<26) - 1));
  }
  index = code & ~(1<<31);
```

Fig. 8. main loop in C code

closer together than other first level assigned tables. We then do the same procedure for the second level assigned tables starting at the root of the tree, and then the third level assigned tables.

Each element of the tables contains the following information. As shown in Figure 7, if the associated index points to a leaf of the tree, then its most significant bit (MSB) is 1, and the next 31 bits will contain the interface number. If the associated index points to another table, i.e. there is at least another table before we reach a leaf, then MSB is 0. The next 5 bits encodes the number of bits needed to index the next table, or simply, the height of the next table. The last 26 bits contains the offset in memory location of the next table relative to the first element of the array p. Given this encoding, we can perform a route lookup with a simple coded loop that parses an address against our table data structures. Prototype code to do so is shown in Figure 8.

VI. Results

In order to have meaningful comparisons for the obtained results, we compare our algorithm to the one in [5], as this is the other contemporary work that carries out deliberate optimization of lookup table designs. We begin with a brief overview of their algorithm.

A. Controlled Prefix Expansion

In [5], the authors present a series of techniques for a range of data structures; one of the data structures is *optimal trie using varying strides* – this is equivalent to our generalized level-compressed trie. The particular technique used for this structure is *controlled prefix expansion*. Its objective is to find a set of tables with minimum total size, such that no single prefix will require more than k lookups. k is selected by the designer to yield a worst case lookup time of kT_2, where T_2 is the type 2 memory access time, with the assumption that all the tables will fit in the type 2 memory space. The following "minimize size" function, called initially on the root node i, performs this optimization:

$$MS(i,k) = \begin{cases} \min\limits_{1 \le h \le H_i} \left[2^h + \sum\limits_{j \in L_{h,i}} MS(j, k-1) \right] & \text{if } k > 1 \\ 2^{H_i} & \text{otherwise} \end{cases} \quad (16)$$

Each function call to a node i requires at most H_i operations. Again, let the height of the complete prefix tree be H, which is again 32 for IPv4 address. Since the function is called on every node, the order of complexity is $O(n * H) = O(Hn)$. If the total size of table does not fit in the secondary cache, however, k needs to be increased and the optimization be performed again. Let B be the number of times the optimization needs to be performed until the total size of tables fit in type 2 memory. The complexity is then $O(HnB)$.

algorithm	memory	i.i.d. prefix	scaled prefix
S & V, k=7	84.4 kB	1.020 mil.	–
S & V, k=6	86.3 kB	1.156 mil.	–
S & V, k=5	93.6 kB	1.259 mil.	–
S & V, k=4	115.0 kB	1.399 mil.	–
S & V, k=3	202.5 kB	1.626 mil.	1.695 mil.
S & V, k=2	748.5 kB	–	–
Lagrange	2.202/1.221 MB	1.770 mil.	2.198 mil.

Fig. 9. Performance Comparison for PAIX: speed in lookups per second

B. Comparison

We first note that the order of complexity of the Srinivasan & Varghese algorithm, $O(HnB)$, is comparable to the complexity of our Lagrange approximate algorithm, $O(HnA)$. Experiment shows that B for minimum size algorithm is about 3, while A for Lagrange algorithm is about 10. To compare performance between the S&V algorithm and the Lagrange algorithm, we use the two algorithms to generate sets of lookup tables separately. Our testing environment is a Pentium II 266MHz processor, with L1 data cache 16kB and L2 cache 512kB. We assume L1 cache access speed is 1 clock cycle, L2 cache is 3 clock cycles, and third level memory is 10 clock cycles. We first used the "PAIX" routing table as input, down-loaded from Merit Inc [13] on 6/19/98. It is a small size routing table with 2638 prefixes. Because we were unable to collect real statistics from a trace at a router, we assume two possible probability distributions of prefixes: i) each prefix is independent and identically distributed; and, ii) each prefix is exponentially scaled according to its length – for example, prefix of length 8 is twice as likely as prefix of length 9.

For each above mentioned probability distribution, we generate 10 million IP addresses and store them to a file on the local disk. We decode each of the 10 million addresses in the following manner: first extract the 32 bit quantity (IP address) from the file, then perform the sequential table lookups described by the S&V algorithm (Lagrange algorithm). We decode the 10 million IP address file 20 times, record the execution time of the entire decoding process, and find the average lookup time per address. Note that the average lookup time obtained in this manner includes the local disk access time, execution time of the table lookup operations, and the hierarchical memory access time.

In Figure 9, we see the results of our Lagrange algorithm and the minimum size algorithm of [5], for various values of k's. We first note that $k = 3$ case is the chosen implementation for minimum size algorithm, since $k = 2$ case is too large to fit in the secondary cache. From the table, for the i.i.d. prefix case, we see a 8.9% increase in speed of our algorithm over minimum size algorithm of $k = 3$. For the scaled prefix case, we see a 29.7% increase in speed. We also compared performance for a much larger "AADS" routing table, with 23,068 prefixes. Again, without available statistics, we assume the prefixes are distributed in the above mentioned two cases. The comparative results are shown in Figure 10. Notice that in this case, $k = 4$ is the chosen implementation since this is the smallest value of k such that the set of tables fit in L2 cache. For the i.i.d. prefix

algorithm	memory	i.i.d. prefix	scaled prefix
S & V, k=7	386.6 kB	1.035 mil.	–
S & V, k=6	389.8 kB	1.172 mil.	–
S & V, k=5	403.2 kB	1.115 mil.	–
S & V, k=4	448.0 kB	1.252 mil.	1.681 mil.
S & V, k=3	631.4 kB	–	–
Lagrange	2.126/1.950 MB	1.570 mil.	1.869 mil.

Fig. 10. Performance Comparison for AADS: speed in lookups per second

case, we see a 25.4% increase in speed of our algorithm over minimum size algorithm of $k = 4$. For the scaled prefix case, we see a 11.2% improvements.

VII. Conclusion

In this paper, we take a theoretical approach to solve the routing table design problem to minimize the average lookup time per prefix. We develop an optimal and an approximate algorithm that solve the optimization problem based on our cost model. The complexity of our approximate algorithm is comparable to the ones in the literature. Experimental results show our approximate algorithm has visible improvements over an existing algorithm in the literature, for both the i.i.d. prefix distribution case and the scaled prefix distribution case.

In this work, we consider only lookup tables as a mean to decode prefixes of IP addresses. There are other methods we plan to investigate in the future, such as logical operations (`if` statements). A lookup algorithm using a hybrid of logical operations and lookup tables may be more optimal.

Appendix

We now prove our approximation algorithm terminates in a bounded error solution. Our proofs are similar to ones stated in [9], and are included in here for completeness. We will begin with the duality proof of Lagrange multipliers.

A. Theorem 1

Let our design problem be:

$$\min_{b \in B} H(b) \quad \text{s.t.} \quad R(b) \leq S_1 \tag{17}$$

where b is a configuration in all possible configurations in set B, $H(b)$ is the associated cost of using configuration b, $R(b)$ is the total size of lookup tables assigned to type 1 memory, and S is the available size of type 1 memory. Define the dual problem to be:

$$\min_{b \in B} H(b) + \lambda R(b) \tag{18}$$

For every multiplier $\lambda \geq 0$, the corresponding optimal solution $b^*(\lambda)$ to (18), is also optimal solution to (17) with $S_1 = R(b^*(\lambda))$.

A.1 Proof 1

Suppose $b^*(\lambda) = b^*$ is the optimal solution to (18) for a given λ. That means:

$$H(b^*) + \lambda R(b^*) \leq H(b) + \lambda R(b) \quad \forall b \in B \tag{19}$$

In particular, the above equation is also true for a subset of B, namely $B_1 \subseteq B$ such that $B_1 = \{b \in B | R(b) \leq R(b^*)\}$. Therefore:

$$\begin{aligned} \lambda [R(b^*) - R(b)] &\leq H(b) - H(b^*) \\ 0 &\leq H(b) - H(b^*) \quad \forall b \in B_1 \end{aligned} \tag{20}$$

Therefore b^* is also the optimal solution to (17) with $S_1 = R(b^*)$. □

We now prove the constraint parameter, $R(b^*)$, is inverse proportional to the multiplier λ.

B. Lemma 2

Suppose $b_1^* = b^*(\lambda_1)$ is optimal solution to (18) for multiplier value λ_1, and $b_2^* = b^*(\lambda_2)$ is optimal solution to (18) for multiplier value λ_2. Suppose further that $\lambda_1 \geq \lambda_2$. Then $R(b_1^*) \leq R(b_2^*)$.

B.1 Proof 2

By optimality of b_1^* and b_2^* for their respective multipliers:

$$H(b_1^*) + \lambda_1 R(b_1^*) \leq H(b_2^*) + \lambda_1 R(b_2^*) \tag{21}$$

$$H(b_2^*) + \lambda_2 R(b_2^*) \leq H(b_1^*) + \lambda_2 R(b_1^*) \tag{22}$$

If we add the two equations and collect terms, we get:

$$(\lambda_1 - \lambda_2) R(b_1^*) \leq (\lambda_1 - \lambda_2) R(b_2^*) \tag{23}$$

Since $\lambda_1 \geq \lambda_2$ by assumption, therefore $R(b_1^*) \leq R(b_2^*)$. □

Finally, we prove that our approximate solution from solving the dual has a bounded error.

C. Lemma 3

Suppose $b_1^* = b^*(\lambda_1)$ is optimal solution to (18) such that $R(b_1^*) < S_1$. Suppose $b_2^* = b^*(\lambda_2)$ is optimal solution to (18) such that $R(b_2^*) > S_1$. Suppose further that b^* is the true optimal solution to (17). Then the approximation error of using b_1^* as solution to (17) is:

$$|H(b_1^*) - H(b^*)| \leq |H(b_1^*) - H(b_2^*)| \tag{24}$$

C.1 Proof 3

Since b_2^* is optimal to (18) for λ_2:

$$H(b_2^*) + \lambda_2 R(b_2^*) \leq H(b^*) + \lambda_2 R(b^*) \leq H(b^*) + \lambda_2 S_1 \tag{25}$$

The last step is true since $R(b^*) \leq S_1$. Rewriting the above:

$$H(b_2^*) - H(b^*) \leq \lambda_2 [S_1 - R(b_2^*)] \leq 0 \tag{26}$$

The last step follows since by assumption $S_1 \leq R(b_2^*)$. Lemma 3 follows from the last equation. □

References

[1] M.Degermark, A.brodnik, S.Carlsson, S.Pink, "Small Forwarding Tables for Fast Routing Lookups," SIGCOMM 97, pp.3-13, 1997.

[2] S.Nilsson and G. Karlsson, "Fast Address Lookup for Internet Routers," to appear in *International Conference on Broadband Communication"*, April 1998.

[3] Y.Rechter and T.Li, *An Architecture for IP Address Allocation with CIDR RFC 1518.* http://sunsite.auc.dk/RFC/rfc/rfc1518.html, 1993.

[4] Keith Sklower, "A Tree-based Routing Table for Berkeley UNIX". Technical Report, University of California, Berkeley.

[5] V.Srinivasan and G.Varghese, "Faster IP Lookups using Controlled Prefix Expansion," to appear in *ACM Sigmetrics 98*, 1998.

[6] P.Gupta, S.Lin, N.McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Infocom 98*, vol.3, pp.1240-7, 1998.

[7] B.Lampson, V.Srinivasan, G.Varghese, "IP Lookups using Multiway and Multicolumn Search," *Infocom 98*, vol.3, pp.1248-56, 1998.

[8] G.Cheung, S.McCanne, C.Papadimitriou, "Software Synthesis of Variable-length Code Decoder using a Mixture of Programmed Logic and Table Lookups," submitted to *DCC'99*, November 1998.

[9] Y.Shoham and A.Gersho, "Efficient Bit Allocation for an Arbitrary Set of Quantizers," *IEEE Trans. ASSP*, vol.36, pp.1445-1453, September 1988.

[10] Garey and Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, pp.247, 1979.

[11] D.Morrison, "PATRICIA- Practical Algorithm To Retrieve Information Coded in Alphanumeric," *Journal of the ACM*, vol.15, No.4, pp.514-534, October 1968.

[12] E.Fredkin, "Trie memory," *Communications of the ACM*, 3:490-500, 1960.

[13] *http://www.merit.edu/ipma*