2003

Level Compressed DAGs for Lookup Tables

Ioannis Ioannidis

Ananth Y. Grama
Purdue University, ayg@cs.purdue.edu

Report Number:
03-025

LEVEL COMPRESSED DAGS
FOR LOOKUP TABLES

Ioannis Ioannidis
Ananth Grama

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #03-025
July 2003

Level Compressed DAGs for Lookup Tables

Ioannis Ioannidis and Ananth Grama,
Department of Computer Sciences,
Purdue University,
W. Lafayette, IN 47907.
{ioannis, ayg}@cs.purdue.edu

Abstract— **Trie-based data structures for implementing IP lookups have attracted considerable research attention. Techniques such as path compression, level compression, generalized level compression, and controlled prefix expansion are commonly used to implement lookup tables. In this paper, we present a fundamentally new technique that relies on directed acyclic graphs (DAGs), which, when coupled with generalized level compression, yield significantly better performance than existing techniques. Current implementations of trie-based lookup tables utilize a route validation table in addition to a trie to enable fixed-length sub-prefix resolution to support path compression. This path validation enables us to merge different, partially filled subtrees to form full subtrees. The resulting DAGs introduce spurious routes that are eliminated in the validation phase. When combined with level compression (and generalized level compression), this structure yields considerably shorter paths than existing approaches. In this paper, we describe transformation of tries to DAGs, algorithms for packing complementary subtrees, and profile performance of these algorithms and resulting improvements in lookup time. Specifically, we demonstrate, on actual lookup tables, performance gains of up to 34% compared to LC-tries with minimal memory overhead (a little over 1%). Considering the fact that an LC trie is already a highly optimized structure, these gains are remarkable.**

Keywords: Routing, lookup tables, tries, level compression, combinatorics

I. Introduction and Motivation

The problem of developing efficient data structures for IP lookups is an important and well studied one. Given an address, the lookup table returns a unique output port corresponding to the longest matching prefix of the address. Specifically, given a string s and a set of prefixes S, the target is the longest prefix s' in S that is also a prefix of s. The most frequently used data structure to represent a prefix set is a trie, because of its simplicity, efficiency, and dynamic nature. A variation, that has, in recent years, gained in popularity is the combination of tries with hash tables. The objective of these approaches is to create hash tables for the parts of the trie that are most frequently accessed. The obvious obstacle to turning the entire trie into a hash table is that such a table would not fit in the router's memory. The challenge is to identify parts of the trie that can be compressed into hash tables without exceeding available memory, while yielding most benefit in terms of memory accesses.

A scheme combining the benefits of hashing without increasing associated memory requirement, called *level compression*, is described in [1]. This scheme is based on the observation that parts of the trie that are full, can be replaced by hash tables containing leaves of the subtrie. This does not increase the memory needed to represent the trie or cause any routing information to be lost. This simple, yet powerful idea reduces the expected number of memory accesses for a lookup to $O(\log^* n)$, where n is the size of the original trie, under reasonable assumptions for the probability distribution of the input. In [2], a generalization of level compression, known as *controlled prefix expansion*, was presented. The objective there is to find a level compression plan that achieves a given lookup time with minimal memory requirements. In [3], an approximation algorithm for optimizing performance under a memory constraint was described. This was the first technique that provably approximates the optimal average lookup time within a constant factor, for constrained space.

Although the above techniques deal with various parameters for configuring lookup tables in main memory, there are other factors such as presence of excess memory or suitable cache placement schemes that impact performance. Generalized level compression introduces nodes into the original trie that do not carry additional information. These nodes merely align subtrie leaves into a hash table for better access time. The associated improvement in lookup time comes at the expense of excess memory required for the additional nodes. In this paper, we de-

This work is supported in part by National Science Foundation grants ACI-9875899, ACI-9872101. Computing equipment used for this work was supported by National Science Foundation and by the Intel Corp.

scribe a novel strategy for minimizing this excess storage while maintaining the performance gains of generalized level compression. This strategy relies on packing complementary nodes from different parts of the trie into a single hash table. In doing so, our scheme transforms a trie into a DAG. We show that the spurious routes added by this transformation can be easily eliminated using route validation tables that are already used in current trie-based lookup table implementations. We present time and memory optimal algorithms that transform a given LC trie into an LC DAG, which, for practical cases is optimal – i.e., one which has optimal packing of hash tables. We show that this packing recovers most of the excess memory of generalized level compression while yielding considerable performance improvements.

The rest of this paper is organized as follows: we initiate the discussion with an informal overview of the proposed scheme and put it in the context of related results in Section II. We provide algorithms for packing complementary subtries and analyze their performance in Section III. We present detailed experimental validation of our results in Section IV and discuss the implications of our results in Section V. Finally, we draw conclusions and outline ongoing efforts in Section VI.

II. Overview of the Scheme and Related Research

We initiate our discussion with a simple description of how lookup tables are typically implemented. Figure 1 illustrates a lookup table with two entries – 0 0: p1 and 1: p2. The associated trie can be path compressed as illustrated in the figure. For reasons of efficiency, it is desirable to resolve a fixed number of bits at each level in the compressed trie. Consequently, the trie is transformed to one that resolves 0: p1 and 1: p2. However, this introduces a false route – namely that a 0 1 prefix is also resolved to p1. To eliminate this prefix, a validation table is maintained in addition to the trie. Leaf nodes in the trie point to entries in the validation table. A port is returned only if the prefix matches the prefix in the corresponding entry of the validation table. For example, in this case, a prefix 0 points to entry 0 0: p1 in validation table. Therefore, a prefix 0 1 does not return p1 since it does not match the entry in the table. Matching entries in the validation table is performed using a single XOR operation, and is therefore very fast.

The presence of this validation step provides us with interesting possibilities for further optimizing the trie structure. Specifically, we can introduce additional false routes, knowing that these can be eliminated with no added overhead. We use this to transform the trie into a

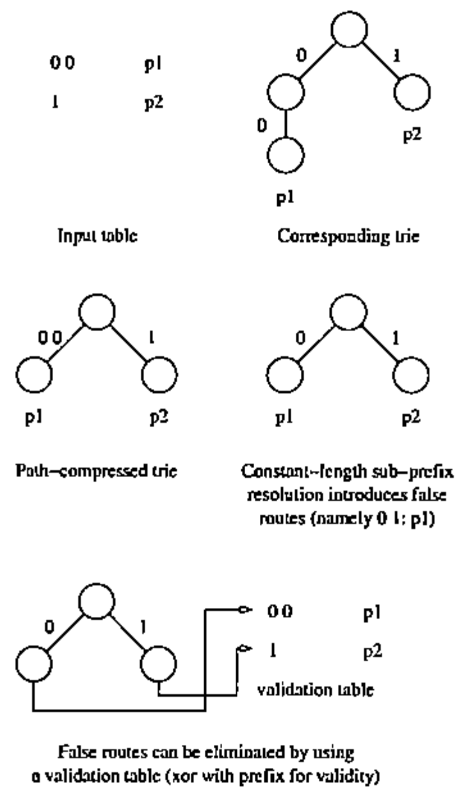
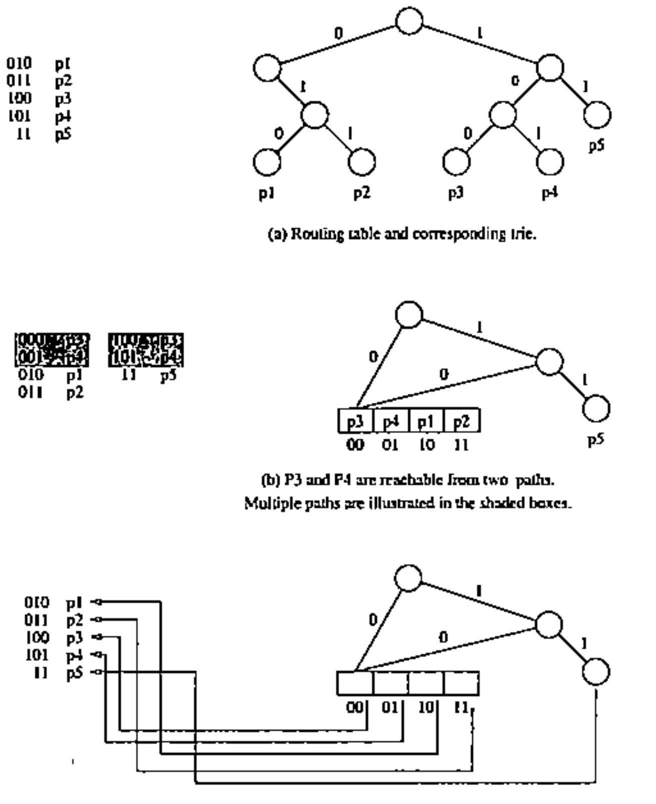

0 0 p1
1 p2
Input table
0
1
0
p2
p1
Corresponding trie
0 0
1
p1
p2
Path–compressed trie
0
1
p1
p2
Constant–length sub–prefix resolution introduces false routes (namely 0 1: p1)
0
1
0 0 p1
1 p2
validation table
False routes can be eliminated by using a validation table (xor with prefix for validity)

Fig. 1. A simple table with two prefixes, the corresponding trie, path compression, and using constant length sub-prefix resolution. False routes introduced by constant length sub-prefix resolution can be eliminated by using pointers to a validation table. Ports are returned only when a prefix matches corresponding entry in validation table.

DAG to improve performance and memory characteristics of LC and generalized LC tries.

We illustrate the DAG transformation with a simple example in Figure 2. The routing table in this case has five routes illustrated in Figure 2(a). In Figure 2(b), we introduce two additional routes corresponding to prefixes 0 0 0: p3 and 0 0 1: p4. The addition of these prefixes enables us to transform the trie in Figure 2(a) into a corresponding DAG, which still uses the same amount of memory as the original trie, however, the average depth of leaf nodes is lower. This technique can be used with both level compression and generalized level compression. In affecting this transformation, certain nodes can be accessed via multiple paths from the root, but only one path corresponds to a valid traversal. This path is reflected in the corresponding entry in the validation table and the other paths are discarded.

A traversal of the trie can reveal which parts of the hash tables are available for reuse in $O(n)$ time. A second traversal partitions the trie into hash tables according to their size. Matching reusable gaps with hash tables of equal size takes $O(n)$ time and is guaranteed to produce a valid DAG – a structure in which a node cannot



(a) Routing table and corresponding trie.

(b) P3 and P4 are reachable from two paths. Multiple paths are illustrated in the shaded boxes.

(c) Pointers to route validation table eliminate spurious paths

Fig. 2. Transformation of a trie into a DAG with appropriate route validation.

be accessed via more than one valid path. Furthermore, when for every table size, there are more hash tables than reusable gaps of the same size, the resulting structure is memory optimal.

A. *Related Research*

The literature on efficient implementation of IP routing is impressively varied. The classical implementation of IP routing for the BSD kernel is described in [4]. True to the spirit of UNIX, simplicity is not sacrificed for performance. In [5], [6], [7], hardware and cache-based solutions are proposed. Hardware solutions tend to become expensive and outdated, while cache-based solutions do not avoid the central issue of prefix matching. Some protocol-based solutions have emerged ([7], [8], [9], [10], [11], [12]), but all of these require modifications to the current Internet Protocol and raise the complexity of routing without completely avoiding the prefix matching problem.

Recent research has focused on algorithmic solutions ([2], [13], [14], [15], [16]). The advantage of these is their transparency to protocol and to advances in hardware platforms. In [1], the original level compression scheme was described. An early effort to formulate a generalization of level compression with memory constraints was presented by Cheung and McCanne ([17]). They formalize memory constraints in the form of an arbitrary memory hierarchy. Cheung et al. ([18]) also show that the problem is NP-complete even for one-level memory and present a simple, dynamic-programming, pseudo-polynomial time algorithm. An approximation using Lagrange multipliers is also described, although no constant bound on the error is derived for this approximation scheme. In [3], an approximation algorithm with constant approximation ratio was presented for the single-level version of the same problem. We use this scheme to derive solutions for the generalized level compression problems, to which we apply our method. There have been attempts to use memory placement in conjunction with architectural features such as cache memory and pipelining ([19]). Memory placement techniques that optimize cache performance without disrupting level compression can be found in [20], where the methods described preserve the tree structure.

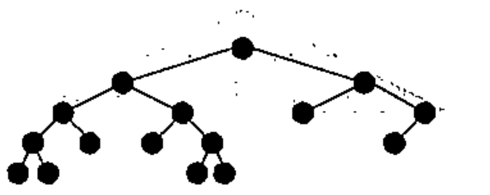
A detailed description of the validation mechanism in the context of LC tries can be found in [21].

III. DAG Transformation and Associated Algorithms

Level compression creates a hash table out of a full subtrie. Generalized level compression, on the other hand, creates hash tables from partially filled tries (Figure 3) to trade excess memory for performance. The nodes of such a table that do not correspond to original trie nodes fall into two categories. The first contains nodes that are extensions of a shorter prefix. The second contains nodes that do not extend any valid prefix. Consider the example illustrated in Figure 4 – a trie stores the prefixes 0 and 1 1 1. Obviously, only addresses starting with one of the two prefixes should be looked up in this routing table. If we turn the entire trie into a hash table, we do not store the original set of prefixes, but rather all prefixes of length three. However, of the eight strings stored in the hash table, only five can be accessed by a trace of correct addresses. These include, the original prefix 1 1 1 and the prefixes that start with 0. If prefixes 1 0 0, 1 0 1 and 1 1 0 are observed, an error should be signaled. Suppose that we now decide to store three nodes from another part of a larger trie at these three positions corresponding to the invalid prefixes. As long as no spurious IP addresses arrive, the modified trie can be used for routing without problems. Furthermore, applying level compression to the first three levels improves the running time, without increasing the net memory requirements.

The assumption that there will be no invalid lookups is not realistic, though. A trace may contain an erroneous address, starting with 1 0 0, for example. In this case, the node stored at that position in the hash table will be accessed, an (invalid) out-port will be retrieved and the packet with the invalid address will be routed. This is undesirable since this packet may potentially bounce through the network for considerable amounts of time. However, there is a standard feature of most lookup table implementations that prevents this from happening. Since path compression is used in most trie implementations and the skipped bits are not checked for space and efficiency considerations, errors like the one described are already possible. The solution is to store, at the leaves, not the out-port, but a pointer to a position in an array of all the prefixes. This position stores the prefix that corresponds to the leaf accessed and the outport. The retrieved prefix is compared to the address and if it is indeed a prefix of the address, the corresponding outport is returned. This validation mechanism, therefore, enables us to pack tries into parts of hash tables that cannot be validly accessed.

In the rest of the paper we assume that the input to our algorithm is a *blueprint* for constructing a level compressed trie for a given lookup table. A node in such a trie consists of a pair of pointers to the children and a number denoting the number of levels that will be turned into a hash table in the level compressed trie, starting from the current node. A blueprint is an abstraction of the output of a level compression algorithm that captures the form of the level compressed trie, while maintaining the structure of the original trie.


(a) Traditional level–compression can level–compress only the top two levels.

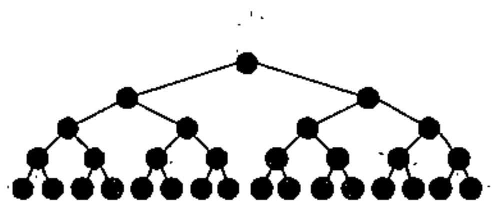
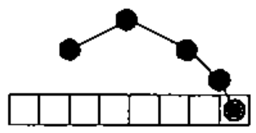
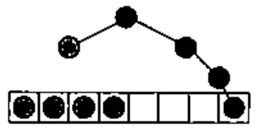

(b) Generalized level–compression can fill the missing nodes and level–compress the entire trie.

Fig. 3. Comparison of traditional and generalized level compression.


(a) The grey node occupies one entry.


(b) The grey node occupies four entries.

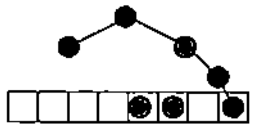

(c) The left subtrie of the grey node creates a gap of two entries.

Fig. 4. Mapping a subtrie to a hash table.

A. Identifying Gaps in Hash Tables.

Let a hash table correspond to a subtrie with root v and depth l. We denote such a subtrie as (v, l). We refer to a path of length less than l in the original trie and starting at v as a prefix. Also, we calculate all depths relative to node v. We therefore, effectively, consider v to be the root of the trie. This does not cause any loss in generality for our purpose, since level compression is a recursive process. Finally, we will refer to a section of a hash table that is available for reuse as a *gap*.

The rules for deciding whether a node belongs to a gap are simple. If a node corresponds to a prefix of length l, then this node cannot be part of a gap (Figure 4(a)). Therefore, when a traversal of (v, l) reaches depth l, no further gaps can be found. Neither can a node corresponding to an extension of a prefix of length less than l be part of a gap (Figure 4(b)). In this case, traversing (v, l) will reach a node u at depth less than l that has no children. If a node corresponds to a prefix that is not an extension of any of the prefixes in (v, l), then this node belongs to a gap. This happens when there is a node at depth less than l that has only one child (Figure 4(c)). The missing subtrie is a gap that can be reused and only a missing subtrie can produce a gap. The size of the gap is 2^{l-m}, where m is the depth of u.

To identify all gaps in accordance with these rules, a single traversal of the blueprint is required. Suppose that a node v, which has a level compression depth of $l \neq 0$ is reached during the traversal. Then, (v, l) is scanned with a modified traversal routine as follows: if a node u is in (v, l) at distance m from v, the number of its children is examined. If u has no children, no gap exists in that part of the hash table. If u has one or two children and $m = l$, the subtries rooted at the children are recursively traversed in the normal way, until another level compressed subtrie is reached. No gap exists in this case. If there are two children, but $m < l$, once again, no gap exists. The traversal continues recursively for the children in this modified fashion. If u has one child and $m < l$, then a gap of size 2^{l-m} exists in the section of the hash table corresponding to the subtrie of depth $l - m$ that would be rooted at the missing child of u. To keep track of the section that is marked, only the string matching the path from v to u needs to be passed on with each recursive call to the modified traversal. The modified traversal continues recursively for the child of u.

As noted above, a gap is specified by the hash table it is contained in, an offset from the start of the table and its size, which can be only a power of two. Since two gaps of the same size are equivalent, we can store gaps in d buckets, where d is the depth of the largest hash table. A gap of size 2^i is stored in bucket i, $0 \leq i < d$. The size of the gap doesn't need to be stored, since it can be inferred from the bucket number. Storing a gap takes $O(1)$ time. Lookup tables generally fit in main memory, therefore we can assume that all offsets and addresses are of size $O(1)$ and the space required for a gap is also $O(1)$.

During the gap identification process, each trie node is accessed once, with $O(1)$ time spent on the node. The total time for this step is $O(n)$, as is the required space.

B. Assigning the Gaps

Once all gaps have been identified, we need to decide how the trie should be packed in memory to take advantage of them. This requires a second traversal of the blueprint. When a node v with level compression depth l, $0 \leq l < d$, is reached, bucket l is accessed. If the bucket is not empty, a gap is retrieved. The subtrie of depth l, rooted at v is stored in the section pointed to by the gap. If the bucket is empty, the subtrie is stored outside a hash table. The traversal continues recursively from the nodes at depth l from v. The total time for this computation is $O(n)$.

The process described above can be sub-optimal because of two reasons. First, suppose there is a hash table of depth i, bucket i is empty, but there are two gaps in bucket $i - 1$, positioned next to each other in a hash table, which are not used. Such a situation results in suboptimal memory usage if the two gaps of depth $i - 1$ are not used by the end of the packing process. Second, suppose there is a gap in bucket i, but bucket $i - 1$ is empty. If there are two hash tables of depth $i - 1$, they will not use the gap of depth i, although it is a valid assignment. If bucket i is not empty at the end of the process, the assignment is sub-optimal.

The above cases suggest that a more elaborate assignment process should be employed. However, if all the buckets are emptied during the assignment process, the resulting memory placement will be optimal. It is our observation from extensive experimental results that these degenerate cases never arise in actual routing tables. We provide an explanation for this in subsequent sections.

C. Validity of an Assignment

For an assignment to be valid it suffices to give a unique memory location to each node and a unique valid access path to each memory location, as long as it preserves the hash tables indicated by the blueprint. Our assignment method relocates entire hash tables. A node can be assigned to more than one memory location only if it is relocated inside the hash table it originally belonged to. Since a gap of size i cannot be part of a hash table of the same size, the uniqueness of memory locations is maintained. Finally, a node can be reused only if it doesn't store a valid prefix or an extension of a valid prefix. This implies that there is at most one access path to every memory location.

D. DAG Transformation and Level Compression

The DAG representation of a trie we have described above can be used in conjunction with level compression as well as generalized level compression. The underlying principle is to pack hash tables that are populated with complimentary patterns. For example, consider two hash tables of size four, the first using entries 0, 1 and 2 and the second using entry 3 (Figure 5). The definition of a used and unused entry in a hash table is the same here as in Section III-A. The two hash tables can be stored in a single, full hash table of size four. In general, one can combine a group of subtries, not necessarily of the same depth, that do not conflict with each other. Level compression can then turn the group into a hash table. Such a scheme reduces lookup time without requiring excess space.

Despite its intuitive nature, this scheme does not perform well in practice when used with traditional level compression. There are two factors contributing to this. The first is the runtime of the grouping algorithm. If d is the depth of the trie and n the number of nodes, there are $O(n \cdot d)$ candidate subtries. To represent the pattern of the children of each subtrie we need $O(n)$ bits. If we use the full set of subtries, any 'reasonable' algorithm for this problem will take at least $O(n^2 \cdot \log n)$ time, because it would have to sort, or at least perform an equivalent operation on $O(n \cdot d)$ keys, each $O(n)$ bits long. Of course, it is difficult to define exactly what a 'reasonable' algorithm for this problem is, but it seems unlikely that the quadratic dependence of the running time from n can be reduced. Since routing tables are updated frequently, running the grouping algorithm would be expensive.

The second problem with such a scheme is even more difficult to overcome. Even if there is an algorithm that can find combinations efficiently, or we decide to work with a suitable subset of subtries, there are not enough complementary patterns in typical routing tables. In experiments where we allowed subtries up to five levels deep, the total gain in running time barely exceeded 1% of the gain produced by simple level compression. Furthermore, we estimated an upper bound on the gain of a more powerful grouping algorithm. According to this variation, a subtrie of arbitrary depth is chosen as a basis for a group. This basis is filled with subtries of depth at most five. The upper bound was estimated to be around 5% of the gain produced by simple level compression, with an expensive packing algorithm.

The failure to produce significant benefit can be attributed to the fact that shallow and sparse subtries are by far the more probable to appear, with the existence probability falling exponentially with height and number of leaves. In Figure 6, we illustrate the distribution of patterns for subtries of depth five of a Mae-West routing table. Each pattern is 32 bits long. The existence of the path 0 0 0 0 0 in the subtrie sets the most significant bit in the pattern while the existence of path 1 1 1 1 1 sets the least significant bit. Intermediate leaves and bits have an analogous correspondence. All subtries that exhibit the same pattern p of their leaves are represented on the horizontal axis of the function $y = p$ in the figure. Complementary patterns appear on axes symmetric to $y = 2^{31} - \frac{1}{2}$ (the central horizontal axis). An examination of the diagram reveals that a large number of patterns cluster tightly around a few non-symmetric axes. In fact, the upper half of the diagram is very similar to the lower half. The ideal case would be for the upper half to be the mirror image of the lower half, because all subtries would have complements. Very few patterns appear outside these clusters and, as a consequence, very few combinations can result in full hash tables.

While packing complementary hash tables does not work well with level compression, it is found to work very well with generalized level compression. Recall that generalized level compression compresses partially filled tries into hash tables based on their benefit. In doing so, it increases the memory requirement in order to reduce lookup time. Packing hash tables in this context can reduce the memory overhead of generalized level compression very significantly while preserving the performance benefits.

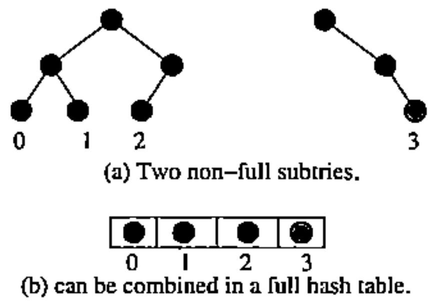

0
1
2
3
(a) Two non–full subtries.
0
1
2
3
(b) can be combined in a full hash table.

Fig. 5. Combining complementary subtries.

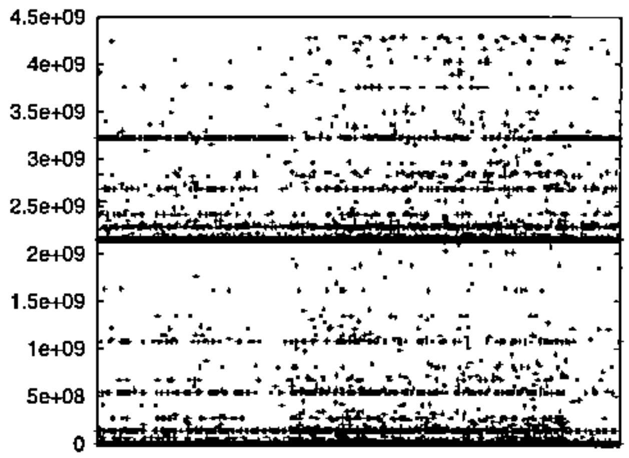
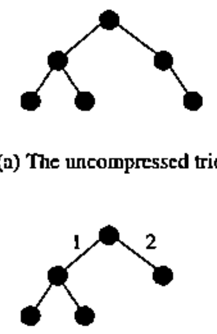


Fig. 6. Distribution of leaf patterns in Mae-West.

E. Path compression vs. Level compression

With traditional level compression, it is assumed that the input trie has been path compressed. This is because no nodes are filled and path compression affects only subtries that are not full. On the other hand, path compressing before level compressing can create some full subtries. For generalized level compression, however, the order needs to be reversed. Subtries that are not full, but which we may wish to fill, will disappear, if path compression is applied first. An example of this case is demonstrated in Figure 7.

Although input tries for generalized level compression are not path compressed, this does not mean that path compression should be ignored. In Figure 7, the path that would be path compressed, if path compression was applied first, should not contribute to the gain produced by level compressing the subtrie containing it. This is because the gain can be derived even without level compression. However, for the algorithm computing the DAG representation of the trie, this is irrelevant. This interaction of path compression and level compression is encapsulated in the algorithm we use to compute the blueprint of the level compressed trie. After the available gaps in the hash tables are identified, we need to find hash tables and single nodes that can be stored in these gaps. We can path compress the parts of the blueprint that are not hash tables. This eliminates single nodes that do not need to be stored. In this manner, only nodes that would appear in a level compressed and path compressed trie are considered for packing.



(a) The uncompressed trie.

(b) Path compressed trie. Level compression is not possible anymore due to varying skip numbers.

Fig. 7. Path compression interfering with generalized level compression. Edges to the children of a node are labeled with corresponding skip numbers, when different.

IV. Experimental Results

We run a series of experiments to measure the performance of our DAG transformation scheme on actual routing data using a Mae-West and a PacBell routing tables. The size of the resulting, uncompressed trie is a little over 5×10^5 nodes and 33581 nodes, respectively. Both tables were obtained from the IPMA project (`http://www.merit.edu/ipma`). We use a uniform distribution for lookups – i.e., every leaf is looked up with equal probability. Since the focus of our results is on storage requirements, this synthetic distribution does not affect our conclusions. We run generalized level compression for different amounts of excess space. We, then, calculate the resulting performance enhancement, and finally, how much of this excess space can be reused by packing. We also measure the runtime of our algorithm. All experiments are run on a 2GHz P4 with 512MB RAM.

The input blueprint in each case is provided by an implementation of the algorithm described in [3]. Nevertheless, there is no restriction on the level compression algorithm used, as long as its output is transformed into an appropriate blueprint.

A. Performance Enhancement

We use generalized level compression for different amounts of excess space, with the maximum being 20% of the original tries. Measurements indicate that the trade-off between benefit and computation time for excess space of more than 20% is unfavorable ([3]). Table I contains values of the performance enhancement as a percentage of that of simple level compression for varying amounts of excess memory. Excess memory is expressed as a percentage of the size of the original trie. The first column specifies the excess memory requirements of the generalized LC trie representation. The second and third columns contain the measured performance enhancements. We illustrate these results in Figure 8.

	Performance Enhancement	
Extra Memory	Mae-West	PacBell
0.2%	2.21%	14.66%
1%	11.8%	20.91%
2%	13.9%	24.88%
10%	25.52%	35%
14%	29.01%	38.21%
20%	34.27%	41.73%

TABLE I
EXPERIMENTAL MEASUREMENT OF IMPROVEMENT IN NUMBER OF LOOKUPS (AS A PERCENTAGE OF THAT OF LEVEL COMPRESSION) WITH INCREASING AMOUNTS OF EXTRA SPACE.

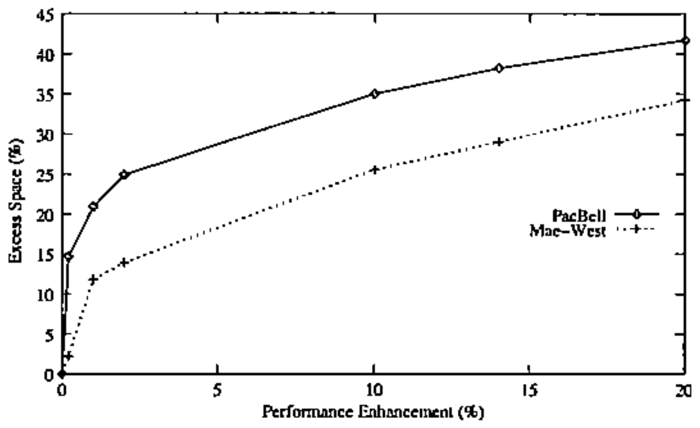


Fig. 8. Performance of generalized level compression in terms of memory accesses saved over a level compressed trie. Savings are represented as a percentage of those of level compression.

B. Storage Reduction by Subtrie Packing

Based on the blueprint for each case, we measure the storage requirements after transforming each trie into a DAG. Table II presents the excess storage requirements as a percentage of the size of the original trie. The first column specifies these requirements before transforming the trie into a DAG. The second column contains the storage requirements for the Mae-West table and the third for the PacBell one. The results from Table I and Table II indicate that by combining subtrie packing with generalized level compression we can improve greatly on simple level compression with very little cost in memory requirements. For the Mae-West table, which is more representative of a backbone routing table, we can improve level compression performance by 35% with only 1.21% more space. For the PacBell table, an improvement of 42% needs less than 7% in excess space.

Trie	DAG	
	Mae-West	PacBell
0.2%	0.0342%	0.1%
1%	0.22%	0.2%
2%	0.31%	0.56%
10%	1.18%	1.9%
14%	1.21%	4.09%
20%	1.21%	6.91%

TABLE II
EXPERIMENTAL MEASUREMENTS OF EXCESS SPACE REQUIREMENTS (AS A PERCENTAGE OF THE ORIGINAL TRIE SIZE) BEFORE AND AFTER TRANSFORMATION INTO A DAG.

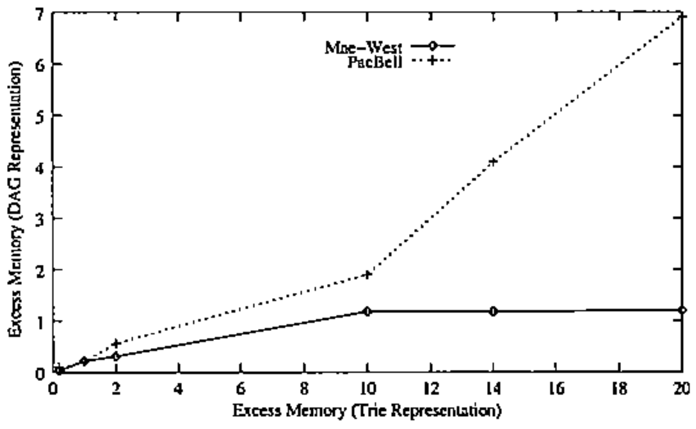


Fig. 9. Reduction of memory requirements for DAG representation.

C. Runtime of DAG Transformation

As we have discussed, the computation of the memory layout is a very fast process as it consists of only two trie traversals. Table III lists the computation times for different amounts of extra space for the generalized level compression algorithm. We list only the runtimes for the Mae-West table. Runtimes for the PacBell table are too small for consistent measurements because of its small size.

Extra Space	Runtime (in secs)
0.2%	0.23
1%	0.23
2%	0.27
10%	0.28
14%	0.28
20%	0.30

TABLE III
EXPERIMENTAL MEASUREMENTS OF RUNTIME WITH INCREASING AMOUNTS OF EXTRA SPACE.

V. Discussion of Results

For our experiments we have used two routing tables of significantly different sizes. The Mae-West routing table is more representative of a BGP routing table. Since the results for the PacBell table follow closely those for the Mae-West table, we first discuss the latter. We discuss specific differences with the PacBell results in the next section.

A. Results for the Mae-West Dataset

From our measurements it is clear that controlling the memory layout of a trie is an extremely beneficial and efficient process. The non-linearity of the growth of storage requirements for the DAG is a point of interest (Figure 9). When generalized level compression is allowed 10% extra space ($5 \cdot 10^4$ trie nodes), the DAG representation requires only 1.18% extra space (5908 nodes). However, when the extra space used by generalized level compression increases to 20%, the DAG representation requires just 1.21% extra storage (6043 nodes). Surprisingly, this is less than the figure for 14% extra space (6050 nodes). On the other hand, an extra space of 0.2% for generalized level compression translates to 0.0342% for the DAG, but extra space of 1% translates to 0.22% – a superlinear growth. It appears that a core of absolutely necessary extra space is reached for 10% extra space and it grows very slowly after that. To understand these results, we need to look at the distribution of gaps.

Table IV shows the distribution of gaps according to their size for generalized level compression with 10% extra space. Table V shows the same for 20% extra space. Almost all the reuse results from single node gaps. The number of gaps falls rapidly with their size, in accordance with our observations in Section III-D. Furthermore, the only population that increases when the extra space doubles is that of single node gaps. This has to do with the way generalized level compression works. Most of the extra space is used to fill subtries very close to being full.

These subtries are not very deep – usually only one or two non-full levels of a subtrie are filled. Rarely are massive amounts of extra space assigned to one subtrie. This allows for many, small gaps and very few sections of hash tables such as those illustrated in Figure 4(b).

Gap Size (in number of nodes)	Population
1	35528
2	2286
4	670
8	132
16	16
32 or larger	0

TABLE IV
GAP DISTRIBUTION FOR 10% EXTRA SPACE FOR GENERALIZED LEVEL COMPRESSION.

Gap Size (in number of nodes)	Population
1	85063
2	2395
4	688
8	137
16	16
32 or larger	0

TABLE V
GAP DISTRIBUTION FOR 20% EXTRA SPACE FOR GENERALIZED LEVEL COMPRESSION.

B. Results for the PacBell Dataset

There are two major respects in which results for the PacBell table differ from those for the Mae-West table. The first has to do with performance measurements. The generalized level compression algorithm performs much better for the PacBell table. However, this is due to an initial boost. For extra space of 2%, there is a difference of 10% between the enhancement for the two tables. This difference remains almost the same for the rest of the measurements, something that is evident from the identical form of the curves in Figure 8.

The second, and more important difference has to do with the storage reduction results. When generalized level compression works with extra space of up to 10% of the original trie, the DAG representations for the two tables are close to each other. However, after this point, the storage requirements for PacBell increase sharply (Figure 9). This seems to contradict the almost constant requirements for the Mae-West table. The explanation for this difference is that the PacBell table is much smaller than the Mae-West table and the opportunities for small gaps are fewer. As we note in the previous section, the larger part of the savings in storage come from small gaps, which are created when subtries of small depth are level compressed. With the smaller PacBell table, there are not as many small subtries that can be level compressed. This increase in storage requirements for the DAG representation should eventually be observed for the Mae-West table as the excess space available to generalized level compression grows.

Finally, we would like to note that in all our experiments the gap usage was optimal. In Section III-B, we showed that whenever the number of available gaps is no larger than the number of candidate subtries, our assignment method is optimal. From the distribution of the gap population, it should be clear why all assignments were optimal. There are very few medium sized gaps and no large gaps, which are the hardest to assign, but many small gaps, for which there are enough candidate one or two node subtries.

VI. CONCLUDING REMARKS AND ONGOING RESEARCH

In this paper we have presented a new, simple, powerful method for reducing storage requirements for IP lookup tables by controlling their memory layout. The resulting structure does not compromise the functionality or performance of the lookup procedure and is compatible with any generalized level compression algorithm. Using this representation we are able to produce a 34% improvement over simple level compression in terms of average lookup time, with only 1.21% increase in net storage requirements. Furthermore, the computation time makes our method practical even when the routing tables are updated frequently.

Ongoing research in our group focuses in more aggressive storage mechanisms for tries. In Section III-D we described an intriguing scheme for improving the performance of simple level compression by combining several non-full subtries that can be stored in a full hash table. The scheme is ultimately defeated by the nature of the input. However, it shows that it is not inconceivable that the lookup procedure can be further enhanced by carefully laying out the routing table.

REFERENCES

[1] S. Nilsson, G. Karlsson, "Fast Address Look-Up for Internet Routers". Proceedings of IEEE Communications Magazine (January).

[2] V.Srinivasan, G.Varghese, "Faster IP Lookups using Controlled Prefix Expansion", Proceedings of the ACM SIGMETRICS '98/Performance '98.

[3] I. Ioannidis, A. Grama, M. Atallah, "Adaptive Data Structures for IP Lookups", INFOCOM '03.

[4] K. Sklower, "A Tree-Based Routing Table for Berkeley Unix", Proceedings of the 1991 Winter Usenix Conference.

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

[6] A. McAuley, P. Tsuchiya, D. Wilson, "Fast Multilevel Hierarchical Routing Table Using Content-Addressable Memory", U.S. Patent Serial Number 034444.

[7] P. Newman, G. Minshall, L. Huston, "IP Switching and Gigabit Routers", IEEE Communications Magazine (January)

[8] G. Chandranmenon, G. Varghese, "Trading Packet Headers for Packet Processing", IEEE/ACM Transactions on Networking (April).

[9] C. Labovitz, G. Malan, F. Jahanian, "Internet Routing Instability", Proceedings of SIGCOMM '97 (October)

[10] G. Parulkar, D. Schmidt, J. Turner, "IP/ATM: A Strategy for Integrating IP with ATM", Proceedings of SIGCOMM '95 (October).

[11] Y. Rekhter, B. Davie, D. Katz, E. Rosen, G. Swallow, D. Farinacci, "Tag Switching Architecture Overview", Internet Draft.

[12] A. Bremler-Barr, Y. Afek, S. Har-Peled, "Routing with a Clue",

[13] P. Crescenzi, L. Dardini, R. Grossi, "IP Address Lookup Made Fast and Simple", Dipartmento Di Informatica, Università Di Pisa, Technical Report TR-99-01.

[14] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, "Scalable High-Speed IP Routing Lookups", Proceedings of SIGCOMM '97 (October).

[15] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, "Small Forwarding Tables for Fast Routing Lookups", Proceedings of SIGCOMM '97 (October)

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

[17] G. Cheung, S. McCanne, "Optimal Routing Table Design for IP Address Lookups Under Memory Constraints", Proceedings of INFOCOM '99, pp. 1437-1444

[18] G.Cheung, S.McCanne, C.Papadimitriou, "Software Synthesis of Variable-length Code Decoder using a Mixture of Programmed Logic and Table Lookups", DCC '99, pp. 121-130.

[19] S. Sikka, G. Varghese, "Memory-Efficient State Lookups with Fast Updates", Proceedings of SIGCOMM 2000, pp. 335-347.

[20] I. Ioannidis, A. Grama, "Adaptive Memory Placement Algorithms for Routing Tables", unpublished.

[21] S. Nilsson and G. Karlsson, "Fast Address Lookup for Internet Routers", in IFIP International Conference of Broadband Communications, 1998