

# Improving and Analyzing LC-Trie Performance for IP-Address Lookup

Jing Fu, Olof Hagsand and Gunnar Karlsson  
 KTH, Royal Institute of Technology, Stockholm, Sweden  
 Email: {jing, olofh, gk}@kth.se

**Abstract**—IP-address lookup is a key processing function of Internet routers. The lookup is challenging because it needs to perform a longest prefix match. In this paper, we present our modifications to an efficient lookup algorithm, the LC-trie, based on a technique called prefix transformation. Thereafter, the LC-trie's performance is analyzed for both the original and our modified algorithm using real and synthetically generated traces. The performance study includes trie search depth, prefix vector access behavior, cache behavior and packet lookup time. The study is based both on experiments and a model for packet lookup time. The results show that the modified algorithm requires only 30% of the lookup time of the original algorithm. In particular, the modified algorithm is capable of performing 60 million packet lookups per second on a Pentium 4, 2.8 GHz, computer for a real traffic trace. Further, the results show that the performance is about five times better on the real trace compared to a synthetically generated network trace. This illustrates that the choice of traces may have a large influence on the results when evaluating lookup algorithms.

**Index Terms**—IP-address lookup, trie, prefix transformation, performance evaluation

## I. INTRODUCTION

As packet transmission rates are continuously growing, routing of IP packets requires faster IP lookups. For each packet, a router needs to extract the destination IP address, make a lookup in a routing table, and transmit it on an outgoing interface. Each entry in the routing table consists of a prefix length pair. Instead of an exact match, an IP-address lookup algorithm performs a longest-prefix match.

LC-trie is an efficient route lookup algorithm proposed by S. Nilsson and G. Karlsson that uses a modified trie data structure [1]. The algorithm is implemented in Linux Kernel 2.6.13. It uses level compression and path compression to reduce the number of trie nodes and the depth of the trie. As a result, a typical IP-address lookup in the LC-trie requires fewer number of node traverses than in an ordinary trie.

However, a considerable part of the lookup time in the LC-trie is spent in prefix and base vector references, which results in a decreased lookup performance. In this work, the LC-trie algorithm is modified in order to

improve its performance. The idea is based on prefix transformation which turns the prefixes in a routing table into a disjoint, complete and minimal set. Thereafter, the algorithm is modified to skip the prefix and base vector references. The modified algorithm results in lower memory usage, fewer memory accesses, and instruction executions, which improve the performance.

Our performance evaluation of the original and modified algorithms covers the search depth of the LC-trie lookups, memory access and cache behavior for the trie nodes, as well as the base and prefix vectors. In particular, we derive a model to estimate the packet lookup time. Using this model, we measure the performance achieved by the original and modified algorithms for various traces. We also validate the model by performing experiments to measure the packet lookup time.

In addition to using synthetically generated random network trace, we use a real trace from the Finnish University and research network (FUNET) [2]. The FUNET trace could be considered as representative of Internet backbone traffic since it contains both university and student dormitory traffic. By comparing the results from the synthetic network trace and the FUNET trace, we show that the performance is about five times better on real data.

The rest of the paper is organized as follows. Section II gives a brief introduction to the LC-trie algorithm. Section III presents the modified algorithm. Section IV presents and characterizes the data, including the routing table and associated traffic traces. Section V describes the performance measurement experiments and introduces the model for estimation of packet lookup time. Section VI presents and analyzes the results from the model and from the experiments. Section VII describes the related work and compares with other efficient approaches for IP-address lookup. Finally, section VIII concludes the paper.

## II. THE LC-TRIE LOOKUP ALGORITHM

A trie is a general data structure for storing strings [3]. Each string is represented by a leaf node in the trie while the value of the string corresponds to the path from the root to the leaf node. Prefixes in a routing table can be considered as binary strings, their values being stored in leaf nodes. In a longest prefix match, a given address is looked up by traversing the trie from the root node. Every bit in the address represents a path selection: If the

---

This paper is based on "Performance Evaluation and Cache Behavior of LC-Trie for IP-Address Lookup," by J. Fu, O. Hagsand, and G. Karlsson, which appeared in the Proceedings of the 2006 IEEE Workshop on High Performance Switching and Routing (HPSR), Poznan, Poland, June 2006.

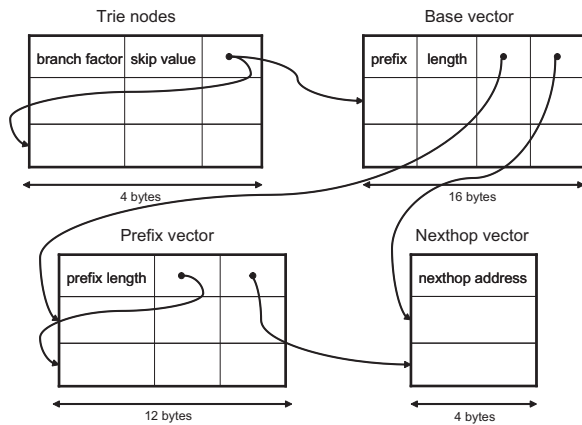


Figure 1. The LC-trie data structure.

bit is zero, the left child is selected, otherwise, the right one is selected. However, the trie data structure is not very efficient. The number of nodes for a whole routing table may be large and the trie may be deep. For IPv4 addresses, the search depth varies from 8 to 32 depending on the prefix length. The search depth is larger in the case of IPv6 which has longer prefix length.

To deal with this problem, the LC-trie uses level compression and path compression to reduce the number of nodes and the average depth in the trie. The path compression removes internal nodes with only one child. The level compression replaces the  $i$  complete levels of a binary trie with a single node of degree  $2^i$ . With these methods, the LC-trie reduces the number of nodes and the depth of the trie.

In order to further optimize the LC-trie performance, the LC-trie may use a weaker criterion for level compression. It uses a fill factor that only requires that a fraction of the prefixes are present to allow level compression. In addition, it allows fixed branching at the root independent of the fill factor. Experiments show that fixed branching to  $2^{16}$  children gives the best performance, since most of the prefix lengths are longer than 16.

The routing table consists of four parts as shown in Fig. 1. The first part is the LC-trie data structure as previously described. Each trie node contains a branching factor, a skip value and a pointer. The branching factor represents the number of descendants of the node, and the skip value indicates how many bits that have been skipped by path compression. The pointer points into the leftmost child if it is an internal node, or into the base vector entry if it is a leaf node.

The second part is a base vector containing complete strings. In addition to the prefix string and its length, each base vector entry contains two pointers: one pointer into the next-hop vector and one pointer into the prefix vector. The third part is a prefix vector that stores proper prefixes of other strings. Each prefix vector entry contains the length of the prefix, and two pointers: one pointer into the next-hop vector and one pointer into the prefix vector. The prefix pointer is needed, since it might happen that a path in a trie contains more than one prefix. Finally, there

is a next-hop vector storing next-hop addresses.

When performing an IP-address lookup, the main part of the search is spent traversing the trie. In a standard setup with fixed branching to  $2^{16}$  children and a fill factor of 0.5, the search depth varies from one to five for most of the routing tables, which corresponds to an equal number of memory accesses. The next step is to access the base vector by following the pointer from the leaf node; this accounts for an additional memory reference. The base vector has to be looked up because the IP-address might not match the prefix due to the skipped bits in the trie lookup. If the IP-address does not match the string in the base vector, lookups in the prefix vector are required. In most of the cases, the prefix vector is accessed no more than once per lookup. Finally the lookup algorithm needs to access the next-hop vector once to determine the next-hop address.

### III. THE MODIFIED ALGORITHM

The modified algorithm is based on a technique called prefix transformation. The prefixes are transformed into a disjoint and complete set, in order to remove the base and prefix vector references in lookups.

#### A. Prefix Transformation

Prefix transformation is to turn a set of prefixes in a routing table into a equivalent set; prefix sets are equivalent if they provide exactly the same lookup results for all possible IP-addresses. There are a variety of ways to transform prefixes. One prefix transformation technique is called prefix compression, which aims at reducing the number of prefixes. A reduced prefix set has lower memory usage, which in turn provides better cache behavior to improve the performance. Another transformation technique is called prefix expansion, which expands the prefixes in order to provide more efficient lookups. In several approaches, prefix expansion is used to reduce distinct prefix lengths [4] [5]. A smaller number of prefix lengths may be exploited by the lookup algorithm and results in improved performance.

The transformation performed in this study is based both on prefix expansion and prefix compression. The expansion turns the prefixes into a disjoint and complete set. A prefix set is disjoint if the address ranges covered by all prefixes are all disjoint. A complete prefix set has the whole address range covered by the prefixes. Therefore, with a disjoint and complete prefix set, there is an exact match for all possible IP-addresses.

In addition to maintain a disjoint and complete prefix set, the goal is also to reduce the number of prefixes to a minimum. A smaller number of prefixes has a lower memory usage and results in improved cache behavior and lookup performance. A prefix set is in its *normal form* if it is disjoint, complete, and there is no other equivalent, disjoint and complete prefix set with a lower number of prefixes.

Prefix transformation can be performed in a stand-alone application or be a part of the lookup algorithm. The

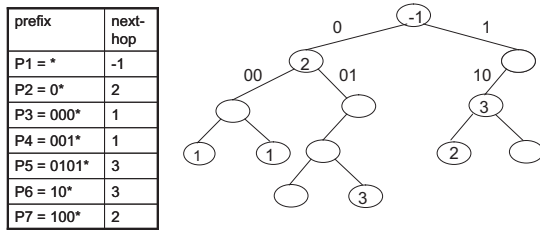


Figure 2. A routing table and its corresponding prefix trie.

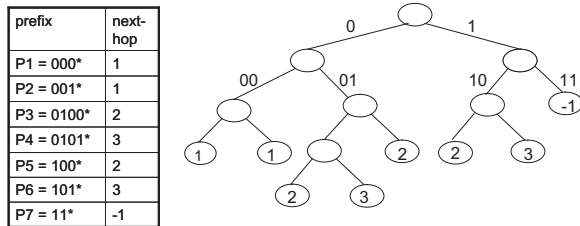


Figure 3. A leaf-pushed routing table and its corresponding prefix trie.

application may take a routing table with prefixes as input, and produces a new table containing transformed prefixes. Subsequently, the produced table is used by the lookup algorithm instead of the original table. Prefix transformation can also be merged into the lookup algorithm, the algorithm may take any routing table as input and transforms the prefixes before building the data structure for routing.

The algorithm for transforming a prefix set into its normal form is summarized into a number of steps as follows:

1. If there is no default route, add a default drop prefix for packets that should be discarded. A default drop prefix is of zero length and matches all IP-addresses. This makes the prefix set complete since an IP-address will match the drop prefix if no other prefix matches.

2. Construct a prefix trie according to the routing table. A routing table and its corresponding prefix trie are shown in Fig. 2. The prefix and its length can be obtained implicitly from the trie structure and the digit inside a trie node denotes its nexthop.

3. Expand the trie by performing leaf pushing which is to replace all aggregated prefixes with leaves. The basic technique is to push a prefix in a node into its two child nodes. For example, prefix  $P6 = 10^*$  in Fig. 2 represents all addresses that start with 10. Among these addresses, some will start with 100 and the rest will start with 101. Thus, the prefix  $10^*$  with length two is equivalent to the union of  $100^*$  and  $101^*$  with length three. In particular, the expanded prefix will inherit the nexthop of the original prefix. However, if there are two prefixes  $10^*$  and  $100^*$  with different nexthops, when expanding  $10^*$  to  $100^*$  and  $101^*$ , the nexthop for  $100^*$  should be kept since it is more specific. The push operation is performed recursively starting from the root node. Fig. 3 shows the leaf-pushed prefix set with all prefixes in leaf nodes.

4. Collapse subtrees: If all leaf nodes of a subtree have the same nexthop, the nexthop of the subtree root node is

```

node = trie[0];
pos = 0;
branch = node.branch;
adr = node.adr;
while (branch != 0) {
    node = T[adr + EXTRACT(pos, branch, s)];
    pos = pos + branch;
    branch = node.branch;
    adr = node.adr;
}
return nexthop[adr];

```

Figure 4. Pseudo code for lookup operation in the modified lookup algorithm.

set to this nexthop. In addition, all subtree nodes except the subtree root node are removed. For example, the prefixes  $P1$  and  $P2$  shown in Fig. 3 should be collapsed to a new prefix  $00^*$  with nexthop 1.

After the above steps, a routing table is transformed into its normal form, which contains a prefix set that is disjoint, complete and minimal.

### B. LC-Trie Modification

Several modifications to the LC-trie algorithm are performed when using a routing table in its normal form. First, the prefix vector is removed since there are no proper prefixes of other prefixes. Second, the base vector is removed since all IP-addresses match exactly one prefix. When traversal of a trie reaches a leaf node, it is an exact match by certainty and does not need a comparison of the prefix in the base vector. Third, a prefix trie in its normal form does not contain internal node with only one child, consequently, path compression will have no effect. Therefore, skip value information is removed and does not need to be accessed during a traversal of a trie node. Fourth, since the base vector is removed, a trie node's pointer into the base vector entry is replaced by a pointer directly to the nexthop entry.

The search algorithm can be implemented very efficiently as shown in Fig. 4. Let  $s$  be the searched string and let  $\text{EXTRACT}(p, b, s)$  be a function that returns the number given by the  $b$  bits starting from position  $p$  in the string  $s$ . An array representing of the trie  $T$  is used, with the root node stored in  $T[0]$ .

The lookup starts at the root node, the child node matching the searched string is determined iteratively until a leaf node is reached. By following the pointer from the leaf node, the nexthop entry is obtained and returned.

## IV. DATA

In order to evaluate the LC-trie algorithm, the following input data is required: (1) A routing table consisting of prefixes, and (2) a trace of destination IP-addresses. In the following subsections, the routing table and the associated traces used in the experiments are described and characterized.

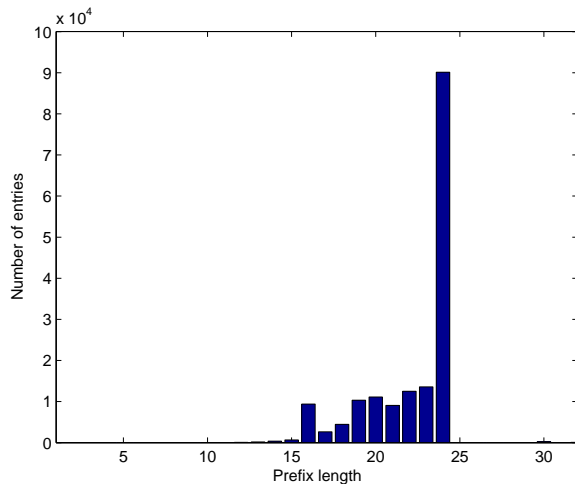


Figure 5. Prefix length distribution for the FUNET routing table.

**A. Routing Table**

A snapshot of a routing table from the Helsinki core router of the FUNET is used. The routing table is also available at CSC’s FUNET looking glass page [6]. The routing table was downloaded in May 2005 and contains 160129 distinct routing entries. Duplicate routing entries that store multi-path entries to the same prefix are removed for simplicity. In Fig. 5, the distribution of the prefix lengths is shown. As can be seen from the figure, the prefix lengths varies from 8 to 32 where the most common prefix length is 24. Further, more than 99.8% of the prefixes have a length between 16 and 24.

The transformation of the FUNET routing table into its normal form resulted in 91726 distinct prefixes. The number is smaller than the original table size as a result of the collapsing of prefixes with the same nexthop. Before the collapsing, the expansion of prefixes resulted in 294252 prefixes. The collapsing is very successful since the number of nexthops is only ten, and most of the prefixes are destined to a small number of nexthops. As the number of nexthops grows, or prefixes are more uniformly distributed to all nexthops, the gain of the collapsing will decrease.

The LC-trie data structures are built using both the original routing table and the table in its normal form. A fixed branching to 2<sup>16</sup> children at the root and a fill factor of 0.5 is used. The sizes of the data structures for these two routing tables are shown in four separated parts in Table I. As can be seen in the table, the total size of the routing table in its normal form is significantly smaller than the original table size.

**B. FUNET Trace**

The first trace used is a real packet trace containing more than 20 million packets with destination addresses. It is captured on one of the interfaces of FUNET’s Helsinki core router. The router is located close to Helsinki University of Technology and carries the FUNET

TABLE I.  
THE FUNET ROUTING TABLE IN ITS ORIGINAL AND NORMAL FORM.

Table	Routing entries	Trie nodes (MB)	Base vector (MB)	Prefix vector (kB)	Nexthop vector (B)	Total size (MB)
Original table	160129	1.27	2.29	160	40	3.45
Normal form	91726	0.89	0	0	40	0.89

international traffic. The interface where the trace was recorded is connected to the FICIX2 exchange point, which is a part of Finnish Communication and Internet Exchange (FICIX). It peers with other operators in Finland. In addition to universities in Finland, some of the student dormitories are connected through the FUNET. The trace itself is confidential due to the sensitive nature of packet’s source and destination IP address. To our knowledge, there is no trace public available on the Internet containing real IP addresses.

**C. Random Network Trace**

The second trace has been synthetically generated by choosing prefixes randomly from a uniform distribution over the 160129 routing entries in the original routing table. The length of the generated trace is equal to the length of the FUNET trace. In such a trace, every entry in the routing table has an equal probability to be chosen. However, as the table may contain prefixes that are more specific than others, this introduces a slight non-uniformity but we believe this to be of minor importance. The trace could be generated by choosing IP addresses randomly over the IP address space as an alternative. Such a trace would give higher probability to shorter prefixes, and hence a higher locality in the lookup.

**D. Trace Characterization**

Fig. 6 shows the fraction of destination IP addresses captured by the most popular prefixes in the original table. As can be seen from the figure, the FUNET trace is heavily skewed towards a few popular prefixes: 40% of the destination addresses are captured by the 30 most popular prefixes and 80% of destination addresses are captured by the 1000 most popular prefixes. In addition, all destination addresses are captured by 100000 prefixes even though there are 160129 prefixes in total. For the random network trace, the 1000 most popular prefixes cover only 0.6% of the destination addresses. Thus, the traces are very different. In the FUNET trace, the destination addresses are concentrated to a small number of networks, while the destination addresses in the random network trace are evenly distributed.

Fig. 7 shows the prefix length distribution for the FUNET lookups using the original FUNET table. The number of lookups is scaled to 160129, which corresponds to the number of distinct routing entries. For the random network trace, the prefix length distribution is similar to that shown in Fig. 5. As can be seen from

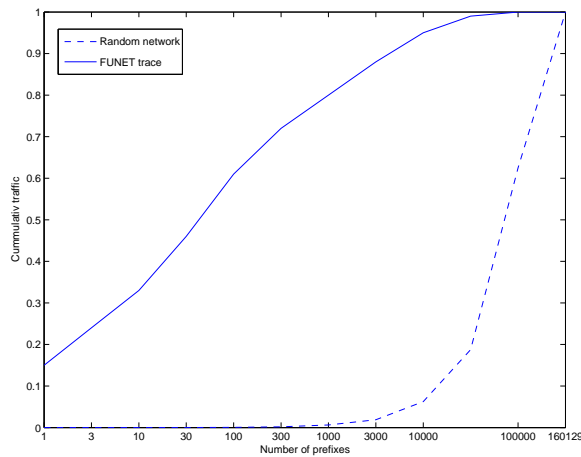


Figure 6. Traffic distribution by prefix.

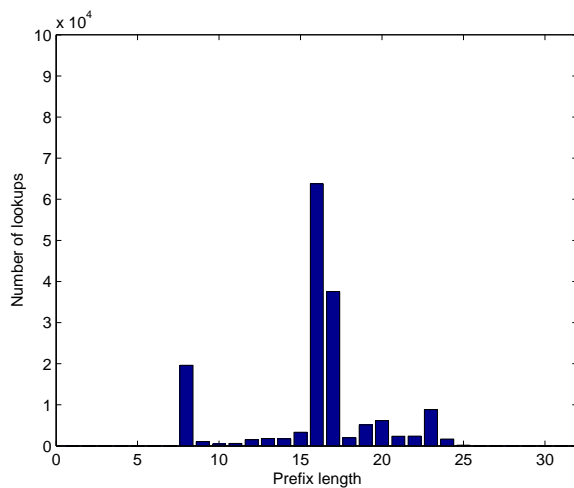


Figure 7. Prefix length distribution for the FUNET lookups.

these figures, the prefix length for most of the FUNET lookups is 16, while the prefix length for most of the random network lookups is 24. The differences on prefix length for the two traces may result in differences in trie search depth and further affects the lookup performance.

## V. PERFORMANCE MEASUREMENT EXPERIMENTS

A series of experiments to study the behavior and performance of the original LC-trie and the modified lookup algorithms has been performed. First, the average search depth in a trie is studied.

### A. Search Depth

The search depth in a trie has an impact on the number of memory accesses and executed instructions. In turn, it has an affect on the packet lookup time. To perform the experiments, the lookup algorithms were modified to record each of the trie node references in the lookups. In addition, the original and modified algorithm with the FUNET routing tables, and the two traces are used in the

experiments. Using the output from the experiments, the average search depths for various setups are calculated.

### B. Prefix Vector Reference

The prefix vector reference in lookups is studied as well. Fewer number of prefix vector references reduces the number of memory accesses and executed instructions, which decreases the packet lookup time. In the experiments, the LC-trie source code is modified to record each of the prefix vector references in the lookups.

### C. Cache Behavior

The cache behavior when performing the lookups is also studied. This includes the cache behavior for the trie nodes, the prefix vector, and the base vector. In current cache and memory technology, a cache access requires about 5 ns, while a main-memory access requires about 60 ns. So, depending on whether a data item can be found in the cache or not, the access time for the data item may have large differences. This has a large impact on the packet lookup time since a significant part of the packet lookup time is spent on data accesses.

We have designed a cache model and implemented two cache replacement policies. The cache model supports a fully associative cache with line size of 16 bytes and two cache replacement policies. In addition, the cache model supports multiple cache levels, since most computers today have two cache levels.

The supported cache replacement policies are random replacement policy and least recently used (LRU) policy. The random replacement policy simply removes a randomized cache item when the cache is full. The LRU policy has to record the latest time an item was accessed, and removes the least recently used item when the cache is full.

In the experiments, the trie node, base vector, and prefix vector references are used as input data. By varying the cache size and cache replacement policy, the cache behavior output for these data structures are obtained.

### D. Packet Lookup Time

The packet lookup time for various setups is also analyzed and studied. As packet lookup time is dependent on the computer architectures, we derived a general model to estimate the packet lookup time. In addition, our results are validated by performing experiments on a computer. Thereafter, the results achieved from the model and from the experiments are compared.

The LC-trie implementation in C is sequential and the average packet lookup time  $\bar{T}_s$  is the sum of the average instruction execution time  $\bar{T}_i$  and the average data access time  $\bar{T}_d$ .

1) *Average Instruction Execution Time:*  $\bar{T}_i$  can be calculated by taking the product of the average number of instructions in a lookup  $\bar{N}_i$ , and the time required to execute a single instruction  $\bar{T}_n$ .

The number of instructions executed for each lookup is dependent on the trie search depth and the prefix vector references. The formula for calculating average number of instructions  $\bar{N}_i$  is as follows:

$$\bar{N}_i = \bar{D}_s N_t + \bar{D}_p N_p + N_c, \quad (1)$$

where  $\bar{D}_s$  is the average trie search depth and  $\bar{D}_p$  is the average number of prefix vector references.  $N_t$  and  $N_p$  denote the number of instructions executed for each trie node and prefix vector reference respectively.  $N_c$  is the count for other instructions executed in the lookup function.

The original and modified LC-trie lookup source codes were compiled using the gcc compiler with the highest optimization level -O4. Thereafter, the text output of the compiled assembler instructions were used to calculate  $N_t$ ,  $N_p$ , and  $N_c$ .

2) *Average Data Access Time:* For the original LC-trie algorithm,  $\bar{T}_d$  can be calculated by taking the sum of the access time spent in the trie nodes  $\bar{T}_{d,t}$ , prefix vector  $\bar{T}_{d,p}$ , and base vector  $\bar{T}_{d,b}$ . For the modified algorithm,  $\bar{T}_d$  equals  $\bar{T}_{d,t}$  since there is no prefix or base vector reference.

$\bar{T}_{d,t}$  can also be calculated by taking the product of average trie search depth  $\bar{D}_s$ , and the sum of the access time spent the L1 cache, L2 cache, and main memory for a single trie node. The formula is as follows:

$$\bar{T}_{d,t} = \bar{D}_s(T_1 + \bar{P}_{1,t}T_2 + \bar{P}_{2,t}T_m), \quad (2)$$

$\bar{P}_{1,t}$  and  $\bar{P}_{2,t}$  are the L1 and L2 cache miss probabilities for the trie nodes while  $T_1$ ,  $T_2$ , and  $T_m$  are the access latencies for the L1 cache, L2 cache, and main memory. For all memory accesses, the L1 cache is accessed first. The average number of L2 cache accesses depends on the L1 cache miss probability, and the average number of main memory accesses depends on L2 cache miss probability.

In a similar way, the formula for  $\bar{T}_{d,p}$  is as follows:

$$\bar{T}_{d,p} = \bar{D}_p(T_1 + \bar{P}_{1,p}T_2 + \bar{P}_{2,p}T_m), \quad (3)$$

where  $\bar{P}_{1,p}$  and  $\bar{P}_{2,p}$  are the L1 and L2 cache miss probabilities for the prefix vector references.

The formula for  $\bar{T}_{d,b}$  is simpler, since the base vector is referenced exactly once in all lookups.

$$\bar{T}_{d,b} = T_1 + \bar{P}_{1,b}T_2 + \bar{P}_{2,b}T_m \quad (4)$$

Finally, by adding  $\bar{T}_d$  and  $\bar{T}_i$ , the average packet lookup time  $\bar{T}_s$  is obtained.

3) *Validation:* The results from the model are validated by performing experiments on two computers: a Pentium 4 computer with a clock frequency of 2.8 GHz, and an AMD Athlon 64 3800+ computer with a clock frequency of 2.4 GHz. We assume that the Pentium 4 computer performs 1.2 instructions per clock cycle, and the Athlon computer performs 1.6 instructions per clock cycle. These numbers are not exact, loosely based on the measurement results from the SPEC [7], and may vary depending on the

TABLE II.  
AVERAGE TRIE SEARCH DEPTHS.

$\bar{D}_s$	Original LC-trie	Modified LC-trie
Random network	2.57	2.47
FUNET trace	1.20	1.06

TABLE III.  
PREFIX VECTOR REFERENCE, AVERAGE AND PERCENTAGE OF LOOKUPS WITH DIFFERENT NUMBER OF REFERENCES.

	$\bar{D}_p$	0	1	2
Random network	0.045	95.5%	4.4%	0.05%
FUNET trace	0.285	71.8%	27.9%	0.3%

applications. Consequently, the times required to perform a single instruction are 0.30 ns and 0.26 ns respectively. Both computers have two cache levels, a L1 data cache and a L2 cache. The Pentium 4 has a small L1 cache of 8 kB, but with a small access latency at 1 ns. The Athlon, on the other hand, has a larger L1 cache at 64 kB with a larger access latency at 2 ns. Both computers have a L2 cache of 512 kB with 6 ns of access latency. In addition, both computers have main memory at adequate size, with access latency of 60 ns.

## VI. RESULTS AND ANALYSIS

In this section, the experimental results are presented and analyzed.

### A. Search Depth

As can be seen from Table II, there is a large difference in search depth between the random network trace and the FUNET trace. The average search depth  $\bar{D}_s$  for the random network trace is about twice as high compared to the FUNET trace. It could be seen in the table that the modified algorithm using a routing table in its normal form decreases the search depth slightly. One reason is the less number of routing entries in the normal form. For data from a Bernoulli-type process with character probabilities that are not all equal, the expected average search depth is  $\Theta(\log \log n)$ , where  $n$  is the number of prefixes [8]. Another reason is that more efficient level compression could be performed, since the trie is complete.

### B. Prefix Vector Reference

Table III shows the prefix vector accessing behavior for the LC-trie algorithm. As can be seen from the table, the prefix vector is not accessed often. For the random network and the FUNET trace, the prefix vector is not accessed in 95.5% and 71.8% of the lookups respectively. Further, the prefix vector is rarely accessed twice per lookup.

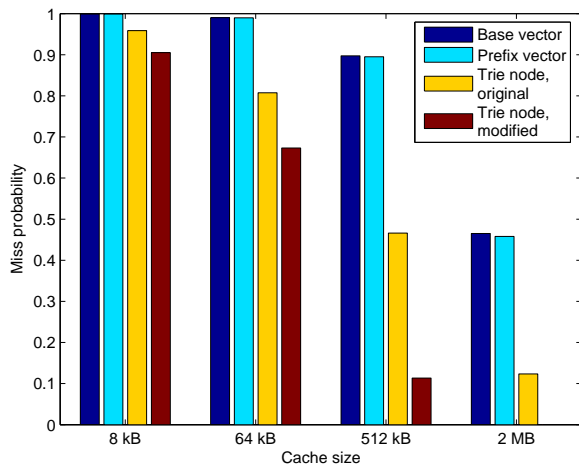


Figure 8. Cache behavior for the random network trace, random replacement policy.

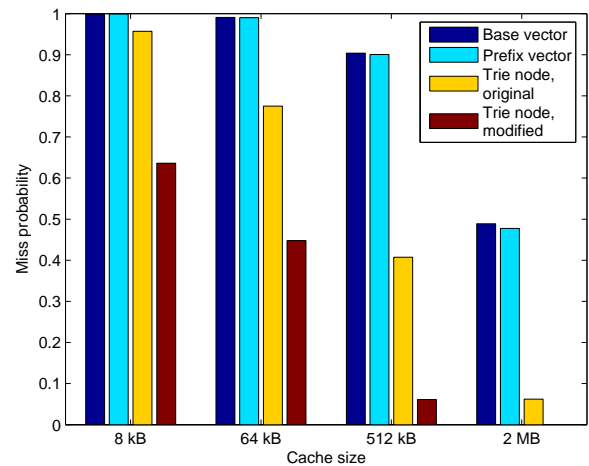


Figure 10. Cache behavior for the random network trace, LRU replacement policy.

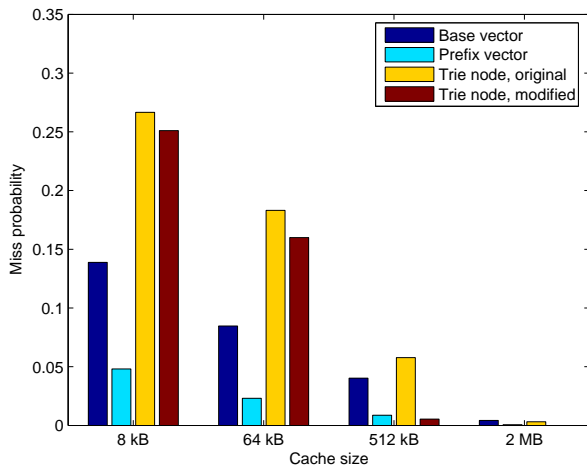


Figure 9. Cache behavior for the FUNET trace, random replacement policy.

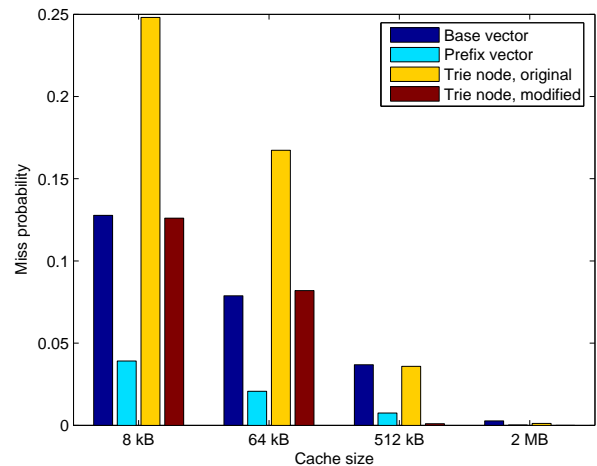


Figure 11. Cache behavior for the FUNET trace, LRU replacement policy.

C. Cache Behavior

The cache behavior for the two traces and two replacement policies is shown in Fig. 8 to 11. The cache size has been varied from 8 kB to 2 MB. For the original algorithm, the cache-miss probabilities for the trie nodes, prefix vector, and base vector are shown on the y-axis. For the modified algorithm, only the cache-miss probability for the trie nodes is shown.

As can be seen from the figures, the trie node miss probability is generally lower when using the modified algorithm. In particular in the case of using larger cache sizes. The original LC-trie data structure is about 3.45 MB while the modified data structure only requires 0.89 MB. Therefore, using the modified algorithm, the miss probability is extremely low at the cache size of 512 kB and there is no cache miss at the cache size of 2 MB.

It could be observed as well that the FUNET trace has much lower cache miss probability compared to the random network trace. In the setups with a 512 kB

cache using the original algorithm, the average trie node cache miss probability for the FUNET trace is 3% for LRU replacement policy and 6% for random replacement policy. In the random network trace, the trie node cache miss probability is 30% for LRU replacement policy and 50% for random replacement policy. There are even larger differences in cache miss probabilities for the base vector and prefix vector: 4% and 1% respectively for the FUNET trace, while in the random network trace, the miss probability is approximately 90% for both vectors.

It can also be seen that the LRU replacement policy has a slightly lower cache miss probability compared to the random replacement policy, especially for the FUNET trace and for the modified algorithm. This behavior is mostly likely caused by the traffic locality in the FUNET trace. Finally, as expected, larger cache sizes lead to lower cache miss probability in all setups.

TABLE IV.  
AVERAGE NUMBER OF INSTRUCTIONS AND INSTRUCTION  
EXECUTION TIME.

	$\overline{D}_s N_t$	$\overline{D}_p N_p$	$N_c$	$\overline{N}_i$	$\overline{T}_i$ (ns) P4/Athlon
Random network, original	64	1	49	114	34 / 30
FUNET trace, original	30	7	49	86	26 / 22
Random network, modified	57	0	25	82	25 / 21
FUNET trace, modified	24	0	25	49	15 / 13

D. Packet Lookup Time

In this section, the average instruction execution time  $\overline{T}_i$  and the average data access time  $\overline{T}_d$  obtained from the model are presented. Thereafter, the average packet lookup time  $\overline{T}_s$  achieved from both the model and the experiments are analyzed.

1) *Average Instruction Execution Time:* From the assembly output,  $N_t$ ,  $N_p$  and  $N_c$  for both the original LC-trie algorithm and the modified algorithm are obtained.  $N_t$  for the original and modified algorithm are 25 and 23 instructions respectively.  $N_t$  for the modified algorithm is two instruction less since counting of skip value is not necessary when traversing through the trie nodes.  $N_p$  for the original algorithm is 25 instructions, and is zero for the modified algorithm since there is no prefix vector reference.  $N_c$  for the original and modified algorithms are 49 and 25 instructions respectively. The modified algorithm has a lower  $N_c$  since it has no base vector reference.

Combining these values with the results in search depth and prefix vector reference, it is possible to calculate the average number of instructions spent in trie nodes, prefix vector, and other operations. The total number of instructions were also calculated, together with the total instruction execution times for both Pentium 4 and Athlon computers, with  $\overline{T}_n$  equals to 0.30 ns and 0.26 ns respectively. As can be seen in Table IV, with the modified algorithm, the number of instructions and instruction execution time decrease for both traces. Besides, it could be observed that the FUNET lookup results in a lower number of instructions, caused by its lower search depth.

2) *Average Data Access Time:* Fig. 12 shows the average data access time  $\overline{T}_d$  for various setups. All configurations have L1 and L2 caches, and use the random replacement policy. The L1 cache is either 8 kB or 64 kB, with  $T_1$  set to 1 ns and 2 ns respectively to mimic the Pentium 4 and Athlon computers. The size of the L2 cache is either 512 kB or 2 MB with  $T_2$  set to 6 ns.  $T_m$  is set to 60 ns according to the hardware specification of both computers.

As can be seen from the figure, the modified algorithm provides a lower  $\overline{T}_d$ , which is caused by lower cache-

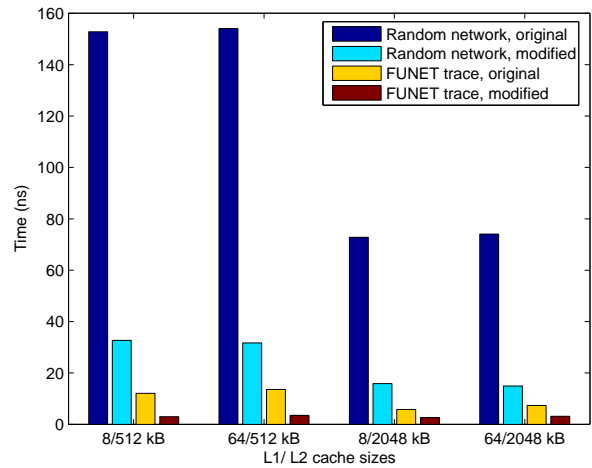


Figure 12. Average data access time,  $\overline{T}_d$ .

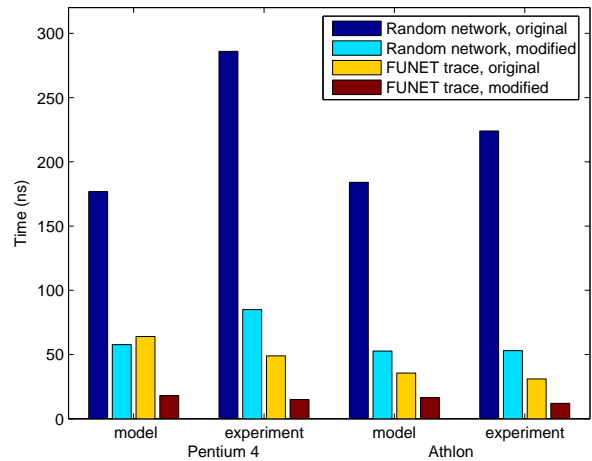


Figure 13. Average packet lookup time,  $\overline{T}_s$ .

miss probabilities and no prefix and base vector reference. Also,  $\overline{T}_d$  is much lower for the FUNET trace compared to the random network trace, which is caused by the lower cache-miss probabilities and the shorter search depths for the FUNET trace. Finally, it could be observed that using a L2 cache of 2 MB may significantly reduce the data access time for the random network trace, however, the impact on the FUNET trace is smaller.

3) *Validation:* The average packet lookup time  $\overline{T}_s$  according to the model and experiments are shown in Fig. 13. The experimental results are from the shortest running time among 10 experiments each performing more than 100 million lookups. When measuring the packet lookup time, the system clock was read before and after the lookup code. As can be seen in the table, the modified algorithms require lower lookup times in all setups.

Another observation is that the lookup time for the random network trace is about five times higher compared to the time for the FUNET trace. For the random network

trace, a larger part of lookup time is spent in data access. Hence, the best way to improve the performance is to reduce the data access time by using a larger cache. For the FUNET trace, the data access time constitutes a smaller part of the total lookup time. Therefore, the impact of a larger cache size on packet lookup time is lower.

It can be seen from the figure that  $\bar{T}_s$  obtained from the model and experiments are somehow close. When using the random network trace,  $\bar{T}_s$  obtained from the experiments are generally higher. The differences are acceptable since the Pentium 4 computer is not modeled in details. When using the random network trace, there is a large number main memory accesses since the cache-miss probabilities are high, the main memory might be contended and takes longer than the specified access time  $T_m$ , which explains the higher lookup time using the random network trace. In addition, many factors, such as predictive branching, cache replacement policy in the hardware and instruction cache miss may cause this discrepancy.

## VII. RELATED WORK

The current approaches for performing IP-address lookup use both hardware and software solutions. The hardware solutions make use of content addressable memory (CAM) and ternary content addressable memory (TCAM) to perform IP-address lookup [10] [11] [12]. To achieve higher performance and reduce the time for memory accesses, many hardware solutions use a pipelined architecture [13] [14] [15].

There are many approaches for fast IP-address lookup that are based on software solutions [5] [4] [16] [17] [18]. The majority of these algorithms use a modified version of a trie data structure. Srinivasan and Varghese present a modified trie data structure for performing IP-address lookup [5]. They try to expand the prefixes so that the number of distinct prefix lengths decreases. In particular, given an expansion level, dynamic programming could be used to determine where to perform expansion. Thereafter, level compression is used based on these expansion levels. While in our approach, level compression is performed based on a fill factor. Also, our approach adds a default prefix before performing expansion, therefore, there will be no internal nodes with only one child in the trie data structure, which may improve the efficiency of level compression. Degermark, et al., quantify the prefix lengths to levels of 16, 24 and 32 bits using prefix expansion [4]. Thereafter, level compression is used and an algorithm using a small data structure that fits into the cache is specially designed. Even though it shows good performance results, algorithms using fixed levels for level compression are difficult to handle changes in routing table structure and to scale to IPv6 addresses. Also, as the cache sizes are growing in PCs, the impact of having a small data structure on the performance will decrease. Finally, the software approaches described above have not performed collapsing of the prefixes based on nexthop

information, which could reduce the FUNET routing table significantly from 294252 to 91726 entries.

There are also several efforts for performing accurate performance measurement for IP-address lookup algorithms. Narlikar and Zane present an analytical model to predict the performance of software-based IP-address lookup algorithms [19]. Kawabe et al., present a method for predicting IP-address lookup algorithm performance based on statistical analysis of the Internet traffic [20]. Ruiz-Sanchez et al., surveys different route lookup algorithms [9]. In particular, they examined packet lookup times for various algorithms by running experiments using a random network trace.

## VIII. CONCLUSIONS

In this work, we have modified the LC-trie lookup algorithm in order to improve its performance. The idea is based on expansion and collapsing of the routing prefixes which turns them into a disjoint, complete and minimal set. Thereafter, the base and prefix vectors are removed from the data structure. This results in a smaller data structure and less number of executed instructions, which in turn improves the performance.

We have also performed a detailed analysis of the original and the modified algorithm. The results are obtained both through our model for packet lookup time and experiments. The performance measurements include trie search depth, prefix vector access behavior, cache behavior and packet lookup time. The results show that the modified algorithm requires only 30% of the lookup time of the original LC-trie algorithm. In particular, the results show that the modified algorithm requires about 15 ns to perform a lookup for the FUNET traffic on a Pentium 4, 2.8 GHz computer. This is approximately 60 million lookups per second for the FUNET trace and corresponds to an average throughput of 120 Gb/s for average sized packets. Also, the modified algorithm decreases the LC-trie memory usage from 3.45 MB to only 0.89 MB. The high throughput of 120 Gb/s on the FUNET trace indicates that a standard PC is capable of performing route lookup for commercial high-speed links. Of course, other limitations, such as the capability of the PC's forwarding functionality, may set the limitations. Also, the efficiency of the algorithm shows that software lookup algorithms can be competitive alternatives to expensive TCAMs. As the LC-trie is implemented in Linux Kernel 2.6.13, our model for packet lookup time can be used to predict the LC-trie performance.

We used the FUNET trace, which is a real and representative trace for aggregated Internet traffic. It contains both university and student dormitory traffic: This is a typical example of aggregated traffic with both office and home traffic. Large performance differences could be observed between the random network trace and the FUNET trace. We conclude that, it is especially important to use a proper trace and understand the characteristics of the trace when performing evaluation of route lookup algorithms.

## ACKNOWLEDGMENT

The authors would like to thank Markus Peuhkuri at Networking Laboratory in the Helsinki University of Technology for providing the FUNET routing table and packet traces.

The implementation of LC-trie for Linux is made by Robert Olsson and Jens Låås from SLU, Swedish University of Agricultural Sciences, Sweden and Hans Liss, Uppsala University, Sweden.

## REFERENCES

- [1] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083-1092, June 1999.
- [2] Finnish University and Research Network (FUNET), Available: <http://www.csc.fi/english/funet>.
- [3] E. Fredkin. "Trie memory," *Communications of the ACM*, pp. 490-499. 1960.
- [4] M. Degermark et al., "Small Forwarding Tables for Fast Routing Lookups". in Proc. *ACM SIGCOMM Conference'97*, pages 3-14, Oct. 1997.
- [5] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems*, vol. 17, no. 1, pp. 1-40, Oct. 1999.
- [6] CSC, Finish IT Center for Science, FUNET Looking Glass, Available: <http://www.csc.fi/funet/status/tools/lg.pl>.
- [7] SPEC, Standard Performance Evaluation Cooperation, Available: <http://www.spec.org/>.
- [8] A. Andersson and S. Nilsson. "Faster searching in tries and quadrees- an analysis of level compression," in Proc. of *Second Annual European Symposium on algorithms*, pp. 82-93, 1994.
- [9] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of ip address lookup algorithms," *IEEE Network Magazine*, vol. 15, no. 2, pp. 8-23, Mar. 2001.
- [10] A. McAuley and P. Francis, "Fast Routing table Lookups using CAMs," in Proc. of *IEEE Infocom'93*, vol. 3, pp. 1382-1391, San Francisco, 1993.
- [11] F. Zane, N. Giriya and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," in Proc. of *IEEE Infocom'03*, pp. 42-52, San Francisco, May 2003.
- [12] E. Spitznagel, D. Taylor and J. Turner, "Packet Classification Using Extended TCAMs", in Proc. of *ICNP'03*, pp. 120-131, Nov. 2003.
- [13] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," in Proc. of *IEEE Infocom'98*, pp. 1240-1247, San Francisco, Apr. 1998.
- [14] A. Moestedt and P. Sjödin, "IP Address Lookup in Hardware for High-Speed Routing," in Proc. of *Hot Interconnects VI*, Stanford, 1998.
- [15] J. Hasan and T. N. Vijaykumar, "Dynamic pipelining: making IP-lookup truly scalable," *ACM SIGCOMM Computer Communication Review*, vv. 35, no. 4, pp. 205-216, 2005.
- [16] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Transactions on Networking (TON)*, vol. 7, no. 3, pp.324-334, 1999.
- [17] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, v.44 n.3, p.289-303, Feb. 2004.
- [18] A.Feldman and S.Muthukrishnan, "Tradeoffs for Packet Classification," in Proc. of *INFOCOM*, vol. 3, 1193-1202, Mar. 2000.
- [19] G. Narlikar and F. Zane, "Performance modeling for fast IP lookups", in Proc. of *ACM SIGMETRICS 2001*, pp.1-12, 2001.
- [20] R. Kawabe, S. Ata and M.Murata, "Performance Prediction Method for IP lookup algorithms," in Proc. of *IEEE Workshop on High Performance Switching and Routing, 2002* pp.111-115, Kobe, Japan, May 2002.

**Jing Fu** is currently a Ph.D. student in the School of Electrical Engineering of KTH, the Royal Institute of Technology in Stockholm, Sweden. He received his M.Sc. degree in computer science from KTH, in 2003. His research interests include router architectures, IP-address lookup algorithms, and network architectures.

**Olof Hagsand** is an associate professor in internet-technology, at the School of Computer Science and Communication of KTH, the Royal Institute of Technology, Stockholm, where he directs the research and education activities of the Network Operations Center (KTHNOC). He received his PhD in Telecommunication Networks and Systems from KTH in 1995, and has been active both in the research and industry in the field of networking, distributed applications and router architectures. As a researcher, he has worked at SICS for ten years, and later at KTH since 2003. He has also an industrial experience in networking companies including Dynarc, Xelerated, and Ericsson.

**Gunnar Karlsson** is professor in the School of Electrical Engineering of KTH, the Royal Institute of Technology since 1998; he is the director of the Laboratory for Communication Networks. He has previously worked for IBM Zurich Research Laboratory and the Swedish Institute of Computer Science (SICS). His Ph.D. is from Columbia University (1989), New York, and the M.Sc. from Chalmers University of Technology in Gothenburg, Sweden (1983). He has been visiting professor at EPFL in Lausanne, Switzerland, and the Helsinki University of Technology in Finland, and ETH Zurich in Switzerland. His current research relates to quality of service, wireless LAN developments and wireless content distribution.