PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

# High–performance routing–table lookup

Martina Zitterbart

*Phil. Trans. R. Soc. Lond. A* 2000 **358**, 2217-2231
doi: 10.1098/rsta.2000.0643

10.1098/rsta.2000.0643



# High-performance routing-table lookup

By Martina Zitterbart

*Institute of Operating Systems and Computer Networks, Technical University of Braunschweig, Bultenweg 74/75, 38196 Braunschweig, Germany* (zit@ibr.cs.tu-bs.de)

The dramatic performance increase observed in the Internet over the last couple of years requires considerably enhanced router technology in order to provide the view of an efficient global network to applications and users. Improvements are needed with respect to data rate but also considering router functionality, for example, to support Quality of Service for multimedia applications. This paper gives a brief overview of some aspects related to router design in general and to route lookup in more detail. Several actual approaches to increase route lookup performance are introduced.

**Keywords: IP routing; routing-table lookup; routing tries; binary search; caches; identifier lookup**

## 1. Introduction

The Internet forms the central part of the information society thanks to the appearance of the World Wide Web. With the advent of the Web, the Internet entered the private living room and forms a widely open space for electronic commerce and other applications. It is no longer a toy of research groups. The logical consequence of the wide acceptance of the Internet is a dramatic increase in number of Internet users and in traffic introduced into the global Internet. Consequently, the performance of the Internet itself needs to improve. During the last few years a dramatic increase in network speed has been observed from data rates in the range of megabits per second to data rates of multiple gigabits per second. The limits of optical transmission, however, have hardly been approached.

In addition to the pure transmission data rate, additional requirements, for example, with respect to Quality of Service (QoS), are raised for emerging multimedia applications. Timely delivery of audio and video data is important as well as the synchronization among different data streams. Furthermore, group communication forms an inherent requirement of applications, such as video conferencing, distributed team work and distance learning. Many different aspects of group communication exist (cf. Wittmann & Zitterbart 2000) including new requirements on router design and implementation, for example with respect to routing-table lookup.

Generally, the developments outlined above impose increased requirements for the network internal interworking units that are responsible for forwarding data units through the network. Consequently, Internet routers (IP routers) are of particular interest. They are considered throughout this paper. High-performance Internet routers are extremely important for the future development of global networking

(Metz 1998; Partridge 1998). This is a market opportunity that has been addressed by numerous startups that develop gigabit and terabit routers.

Forwarding data towards their destination(s) can be considered as the main task of today's routers. The corresponding functions form the so-called forwarding engine which is associated among others with lookup functions to the routing table. The routing table stores the information needed in order to correctly forward data towards the next hop on their journey to the final destination(s). For each data unit—or at least for each data stream—a lookup in the routing table is needed in order to determine the next hop. This lookup operation is a central component of data forwarding. Other functions with considerably low complexity, such as time-to-live (TTL) update and checksumming need to be performed as well. An additional requirement on routers that has established itself during the last few years is the need for firewalls.

Various lookup algorithms have been developed recently that have led to considerable performance improvements. Important factors with respect to performance are key length for memory access, amount of memory required, number of search steps, as well as cost to generate and update the data structure. The developments consider software implementations as well as hardware implementations. An overview of current algorithms forms the core of this paper along with some evaluation.

The paper is structured as follows. Section 2 briefly introduces general architectural issues related to Internet routers and identifies the main building blocks. Routing-table lookup is outlined in its basics. A survey on current algorithms for routing-table lookup is presented in § 3. Internet routers in the advent of QoS requirements are briefly discussed in § 4. Section 5 summarizes and concludes the paper.

## 2. Internet router

Before going into the details of routing-table lookup, the general structure and building blocks of Internet routers are discussed within this section. Intentionally, no implementation concepts are introduced here. Instead, it is intended to provide a very brief overview over the general functions that need to be implemented in such a router. The only task that is outlined more in detail is the routing-table lookup, since it is very performance critical. Consider the following very simple example. Using data units of length 1000 bits each on a network link with a data rate of 1 Gbit $s^{-1}$ results in the requirement of 1 million routing-table lookups for that interface. This clearly requires efficient and de-centralized solutions for high-end Internet routers in the backbone. With respect to performance requirements, different routers can be distinguished: access router, enterprise router and backbone router (Keshav & Sharma 1998). An *access router* provides access to Internet Service Providers (ISPs) for homes or small offices and, thus, serves a very small number of nodes. *Enterprise routers*, in contrast, interconnect multiple thousand computers in a campus or an enterprise. *Backbone routers* typically link ISPs and enterprise networks. Routing at high speeds between a considerably low number of ports is the main requirement for backbone routers. Low cost per port and the provisioning of a large number of ports is important for enterprise routers.

### (a) *Generic router architecture*

In a very simplistic view, a router consists of various network interfaces, a processing component and an interconnection unit among the interfaces. As an inter-

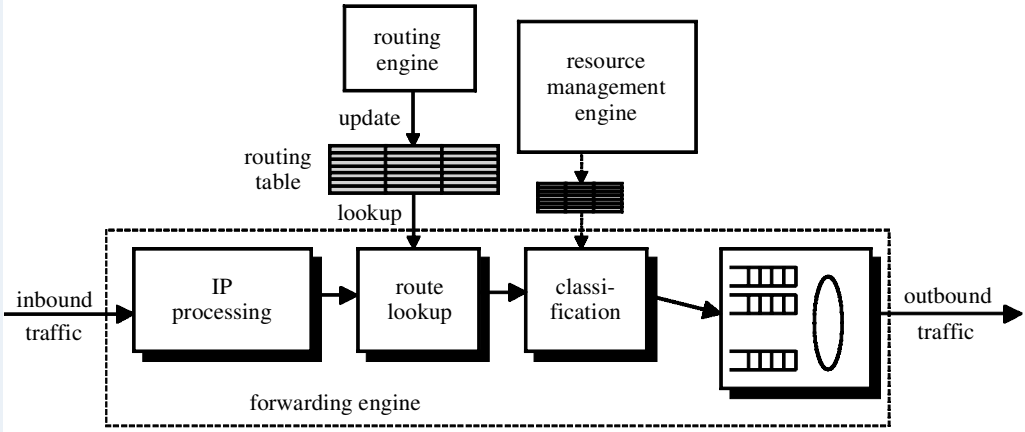


Figure 1. Major building blocks of router processing component.

connection unit, nowadays, switches are typically used. With respect to processing, two very different tasks need to be distinguished, namely, processing that is involved in the data path through the router and processing with respect to control tasks (e.g. routing protocol and network management). Clearly, the data path needs to be implemented highly efficiently, whereas the control path can be satisfied with a somewhat lower performance. In order to gain efficiency, some tasks involved in the data flow may be distributed onto the network interfaces allowing for parallel processing. The major building blocks of the processing component are depicted in figure 1. These are

(i) a forwarding engine;

(ii) a routing engine;

(iii) a routing table; and

(iv) a resource management engine.

Figure 1 shows the data flow being associated with the data path, i.e. incoming data units pass the *forwarding engine* in order to reach the outbound interface(s). The forwarding engine comprises tasks that need to be implemented highly efficiently and cost effectively. Therefore, it is advisable to implement some of the components distributed on the network interfaces, for example to allow for parallel IP processing. Since the subject of this paper is not router architectures but routing-table lookup, the reader is referred to Koufopavlou *et al.* (1994) for the discussion of architectural issues. A highly distributed router architecture with various hardware components is presented in Tantawy & Zitterbart (1992), and Partridge *et al.* (1998) presents an implementation of a gigabit router.

The forwarding engine primarily comprises IP processing, routing-table lookup, packet classification and scheduling, each being performance critical. IP processing includes functions such as address check, TTL check and checksumming. Classification is needed in order to classify the data unit corresponding to the requested QoS level. The data are then forwarded to the respective outbound queue, where they are scheduled accordingly. Routing-table lookup is needed in order to determine the

outbound interface(s) to which the data need to be forwarded. Therefore, the best matching entry in the routing table must be identified. This task is extremely performance critical since it may require multiple memory accesses and, thus, will be considered in further detail within this paper.

The routing engine and the resource management engine both belong to the control path and, thus, are not primarily performance critical with respect to the forwarding performance of routers. The *routing engine* comprises routing protocols which determine updates to the lookup table if required. The routing engine may simply be implemented on a general purpose processor as suggested in Koufopavlou *et al.* (1994). The *resource management engine* is needed for the provisioning of QoS. It simply decides how a data unit is treated in comparison with data units belonging to different data flows. The resource management entity, for example, decides into which outgoing queue the data unit is forwarded and which form of scheduling will be applied to it.

### (*b*) *Internet routing-table lookup*

The central task of a router is to determine the outbound network interface(s) for data flowing through it, i.e. to determine the next hop of a data unit. The required information is stored in the routing table. In order to be scalable for large networks, Internet routing tables do not store complete IP addresses; they keep address prefixes. For example, the following addresses can be represented by the prefix 1001: 1001$\underline{01}$1110$*$, 1001$\underline{00}$0101$*$ and 1001$\underline{11}$1110$*$, where $*$ represents the rest of the IP address. Using prefixes reduces the size of routing tables considerably. However, more complex lookup algorithms are needed since the *longest matching prefix* needs to be identified.

The basic rule behind the longest-prefix match is to select the longest prefix that matches the IP address under consideration. For example, consider a routing table with the following entries.

100
110
1101
110110
1101000

If a data unit is received with an IP address 1101101$*$, the longest prefix for this address needs to be identified. In this example, 110110 will be selected as longest matching prefix. It is most specific for the IP address under consideration. Other valid prefixes with shorter length are 110 and 1101.

At first, Internet addresses were clearly structured into three distinct classes: class A, class B and class C with 1, 2 and 3 bytes of network identification, respectively. As a result, prefixes of three different lengths (8, 16, 24) were of interest for so-called class-based routing. Nowadays, this clear distinction cannot be applied for Internet routing. As a result, prefixes of all lengths need to be considered and, thus, lookup is more complex since a variety of prefix lengths may co-exist within a routing table.

Due to the growth of the Internet in general, the size of the routing table has increased as well. Especially, backbone routers can easily end up with several thousand routing entries. This clearly requires clever search algorithms that bound the worst-case search time within the routing table. Furthermore, these algorithms need to take into consideration that multiple entries in the table may match and that the longest prefix among these entries is required. In Srinivasan & Varghese (1998), table sizes of backbone routers with 45 000 entries or more are reported with prefix lengths being distributed between 8 and 32. Enterprise routers typically have smaller tables with about 1000 entries. This is due to a high number of default routes being used. The Internet Performance Measurement and Analysis project† provides realistic routing tables that can be used to determine the performance of lookup algorithms.

Currently, IP addresses are 32 bit long; the new version of the IP protocol (IPv6) uses addresses with 128 bit and also applies longest matching prefix. As a result of the increased address length, routing-table sizes will also grow, i.e. efficient routing-table lookup becomes even more important.

Lookup operations are more frequently applied compared with update operations. In the case of stable routes, updates should occur rather infrequently. However, in Labovitz *et al.* (1997) it is stated that updates can occur up to 100 times per second. Clearly, such situations also require adequate update algorithms. However, the requirements for lookups are in the order of few hundred nanoseconds or below per lookup.

Critical parameters with respect to routing-table lookup are memory access speed and memory access width. Others of interest include the amount of memory needed to store the routing table as well as the cost of updates in the data structure. The following section outlines some algorithms for routing-table lookup, including traditional solutions as well as recent algorithms that have been developed during the last couple of years.

## 3. Routing-table lookup algorithms

Algorithms for routing-table lookup can be classified according to various criteria. In the following discussion they are basically subdivided into software-based and hardware-based solutions and into those that exploit caching.

### (*a*) *Software-based approaches*

#### (i) *Trie-based algorithms*

Search algorithms are often based on tree structures. With long key words, however, trees might become inefficient, since a comparison is involved at each node within the tree (cf. figure 2*a*). In such cases, often so-called *tries* are applied. In tries, search keys are stored in the leaves of the tree only and not within internal nodes of the tree (cf. figure 2*b*). This way, only one comparison per search is needed. The name 'trie' is derived from 're*trie*val', since the structure can very well be used for the retrieval of information. For the traversal of the trie, the bits of the search key are used until a leaf is reached. Then, the comparison with the word stored in the

† See http://www.merit.edu/ipma.

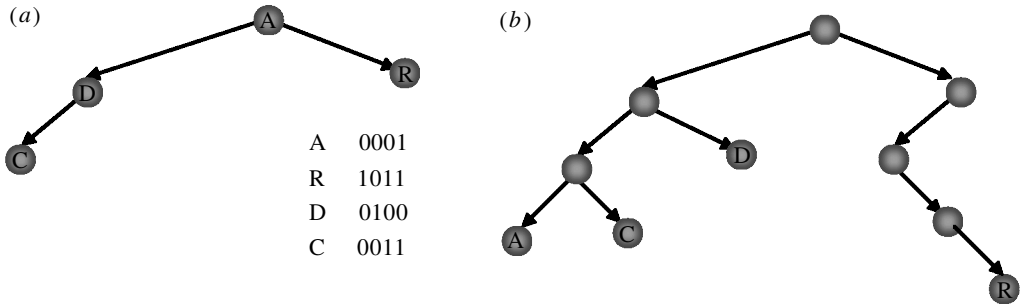


Figure 2. Examples of (*a*) a tree and (*b*) a trie.

leaf takes place. The first three bits of the key word 'C' are 001. Traversal through the trie is to the left in case of a 0 and to the right in case of a 1. Routing-table lookup is typically based on tries.

The *Patricia trie* algorithm (Knuth 1973) can be seen as a traditional way of implementing Internet routing-table lookup as it is, for example, used with Unix platforms (Wright & Stevens 1995). Patricia stands for 'Practical Algorithm to Retrieve Information Coded in Alphanumeric'. A trie generally can be seen as a data structure for representing strings. The value of a string corresponds to the path taken through the trie. The Patricia trie is a binary trie, i.e. only the values 0 and 1 are valid. Furthermore, as an optimization step *path compression* is applied. Path compression refers to the fact that internal nodes with only a single child are removed and the length of the skipped path is remembered in that node (the so-called skip value). As a result, the Patricia trie algorithm does not search for an exact match, but for a prefix. In figure 3 a simple example of path compression is depicted, where the paths to the nodes 1 and 4 are compressed. Tries may lead to high memory consumption since a set of children pointers and the skip value are stored at each internal node. Furthermore, the skipped bits in the path may require backtracking. Various approaches have been presented in order to improve the traditional Patricia trie, e.g. with respect to search time and memory consumption.

The path compression used for constructing a Patricia trie applies to sparsely populated areas of the trie. *Level compression*, in contrast, addresses highly populated areas of the trie. With level compression, each complete subtree of height $h$ collapses to a subtree of height 1 with up to $2^h$ children (Cheung & McCanne 1999). Thus, a level compressed trie is no longer a binary trie but a multibit trie. In the example shown in figure 3, a subtree of height $h = 3$ is collapsed.

The *LC trie* proposed in Nilsson & Karlsson (1998, 1999) enhances a Patricia trie with *level compression*. They then address the space problem of tries by defining a compact representation without requiring multiple children pointers at each node. The children of a node are simply stored in consecutive memory locations. As a result, only a single pointer to the leftmost child is needed in a node. Additionally, the nodes are stored in a contiguous array. Consequently, the trie only requires some 100 kbits of memory. The leaves of the trie point to a so-called base vector that contains the complete strings. It has to be checked whether the found match is a real match (because of path compression). Furthermore, in case of a hit, the next hop is identified by a corresponding pointer in the base vector. If the search was not successful, a pointer to the prefix table is followed. The prefix table contains

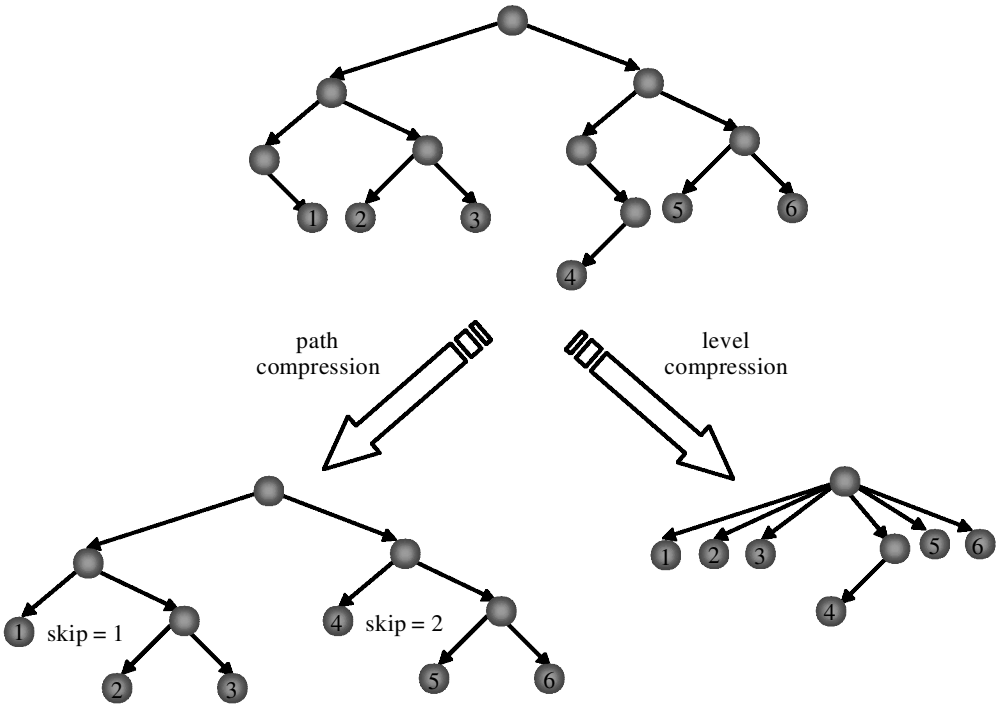


Figure 3. Path and level compression in tries.

information about proper prefixes of other strings. This is needed because internal nodes of the LC trie do not have pointers to the base vector and a path in the trie may contain more than one prefix. Experiments of the authors came up with a typical trie depth of 6. With respect to table lookups, about 2 million lookups per second could be achieved. Due to the construction of the data structure representing the trie, update operations can easily become very costly.

So-called *generalized level compressed tries* are introduced in Cheung & McCanne (1999). They are based on techniques presented in Degermark *et al.* (1997) and Nilsson & Karlsson (1998), trie completion and level compression, respectively. With the technique of *trie completion*, it is ensured that each node either has no or two children and backtracking is avoided. Then, level compression is applied in order to provide for an implementation using lookup tables each with length $h$. In this context, prefix expansion is applied in order to limit the number of different prefix lengths. For example the prefixes $\{00, 10, 1\}$ are expanded to $\{00, 01, 10, 11\}$, in order to determine the generalized level compressed trie. The step of level compression and prefix expansion is formulated as an optimization problem with the goal of minimizing the average lookup time. An optimal and an approximate algorithm have been developed, based on dynamic programming (Cormen *et al.* 1990) and on Lagrange approximation, respectively. With Lagrange approximation, roughly 1.5 million lookups per second can be achieved at the cost of high memory consumption.

In Srinivasan & Varghese (1998), an approach is presented that, in contrast to most other approaches, specifically addresses routing-table updates. Moreover, a 'framework' has been designed that can be tuned to different requirements, e.g. for backbone

routers or enterprise routers. Three techniques are used: controlled prefix expansion, selecting optimal expansion levels and local restructuring.

In addition to the trie completion mentioned above, *controlled prefix expansion* restricts the prefix lengths to a predefined set of length. Since the expansion may lead to prefix collisions, i.e. an expanded prefix can match an already existing prefix, so-called prefix capturing is used that removes the expanded prefix. Selection of prefix lengths is optimized with respect to minimize storage requirements. Dynamic programming is used for that optimization problem. In addition, local restructuring of the trie is used to further improve storage consumption.

(ii) *Approaches using binary search*

The approach presented in Waldvogel *et al.* (1997) is based on binary search over *hashing tables*. The basic idea is that hashing is applied for each prefix length. Binary search is used to reduce the number of searches from linear to logarithmic and, additionally, pre-computing is used to prevent backtracking in case of failures in the binary search. A hashing table is needed for each possible length of the prefix, i.e. 32 hashing tables are required. The search within the different prefix lengths, i.e. the associated hashing tables, is organized as a binary search and starts with the root of the binary tree being located at the median length of prefixes. As a result, this corresponds to *binary search on trie levels* (cf. figure 4). This way, the first step tries to match the first 16 bits of the IP address. In case of a failure, it is searched for a shorter match, in case of a hit for a longer match in the corresponding part of the tree. Markers in the binary tree are needed to direct binary search in order to find longer prefixes. For example, if the prefixes $P1 = 0$, $P2 = 00$ and $P3 = 111$ are available, the search would start at $P2 = 00$ and fail. A marker is needed that indicates that there might be a longer matching prefix. In the example, a marker entry $M = 11$ is added into the hash table. The algorithm is simple in the case of lookup, and well performing. About 1–2 million lookups per second can be achieved. However, updates can be costly since recomputation of markers may be required.

*Binary search on prefixes* (Lampson *et al.* 1998) is also based on binary search, but to the number of prefixes and not on trie levels as in Waldvogel *et al.* (1997). Since binary search can not be applied to variable length strings, the prefixes need to be expanded to a homogeneous length. In this sense, a prefix does not represent a single value but a range of values (lowest to largest expanded prefix). For example, with 6 bit addresses, the prefix $1*$ refers to the range of addresses from 100000 to 111111 and is represented by these two values. The expanded prefixes are, then, sorted. It should be noted that the ranges are nested (cf. figure 5). If binary search ends at a certain point in the table, the corresponding prefix is the first low value (L) in the preceding part of the table that is not followed by a high value (H) (cf. figure 5). The longest matching prefix problem is, thus, translated to finding the *narrowest enclosing range*. The prefixes for every region can be pre-computed in order to increase lookup performance at the cost of lowering update performance. Furthermore, the usage of an array as front end into the binary search can increase performance. The front end can store pre-computed prefixes, for example, for the first 16 bits along with pointers to binary search tables that comprise longer prefix values. According to measurement results, a worst case performance of around 2 million lookups per second can be achieved.

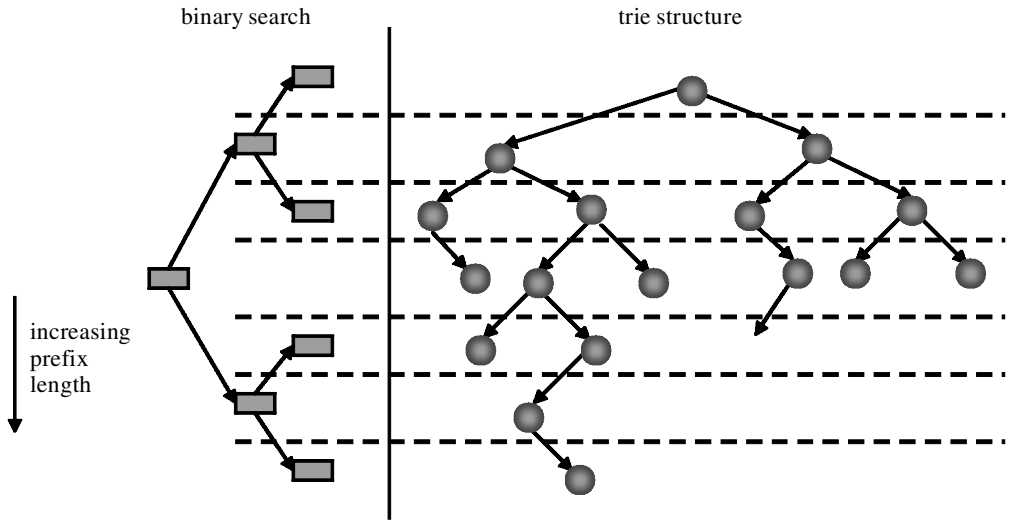


Figure 4. Binary search on trie levels (Waldvogel *et al.* 1997).

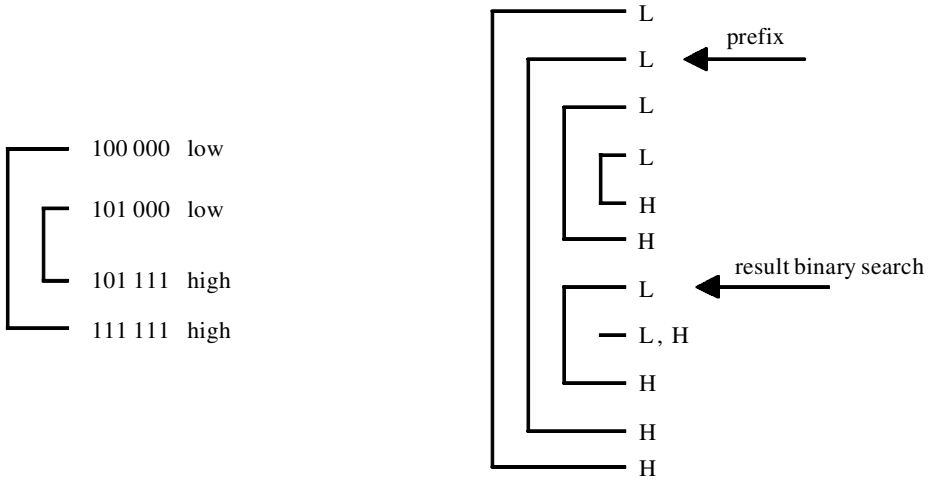


Figure 5. Encoding prefixes as ranges of low and high values (Lampson *et al.* 1998).

Additionally, multiway and multicolumn searches are introduced in Lampson *et al.* (1998). The basic idea of *multiway search* is to exploit the cache line of today's processors. Within a burst, a so-called search node with multiple entries is loaded into the cache. This reduces access time for subsequent accesses to the additionally loaded entries significantly. The number of entries in a search node is in accordance with the cache line. Restructuring of the data is necessary in order to provide locality of access. *Multicolumn search* addresses the problem of long identifiers compared with the word length of the processor (e.g. IPv6 addresses). The identifiers are subdivided into $N$ components and, for each component, binary search is applied until a hit is found. Then binary search to the next column is applied and so on. Around 1 million lookups per second were measured for IPv6 addresses.

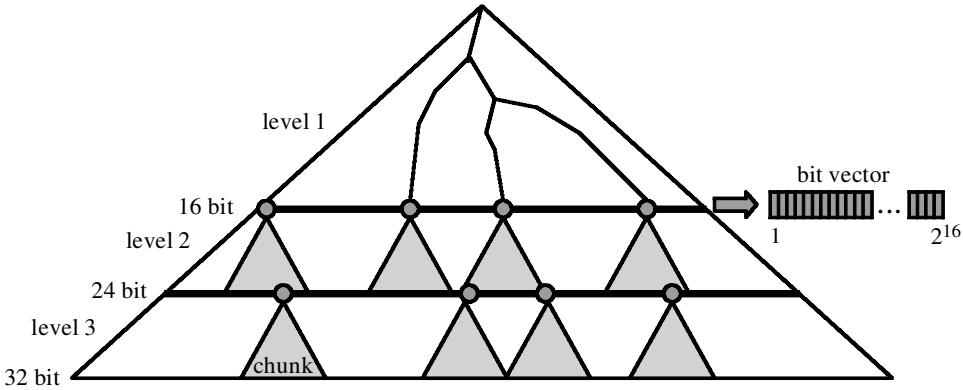


Figure 6. Trie with chunks (Degermark *et al.* 1997).

### (*b*) *Exploit caching*

Besides the design of clever algorithms for applying trie-based search, binary search or hashing to improve route lookup, the *compaction of the routing table* plays an important role. The goal is to keep the routing table in cache and, thus, to significantly lower access time.

A tricky approach with a rather complex data structure for compaction of the routing table and for minimizing memory access is described in Degermark *et al.* (1997). A key to this approach is the compact representation of tries of height $h$. The initial data structure is a trie to which trie completion is applied. Three levels are distinguished in the trie: level 1 being bound at depth 16, level 2 being bound at depth 24 and level 3 being bound at depth 32 (cf. figure 6). Searching in a level can result in a hit and, thus, an index into the next hop table, or in a miss and, thus, an index into an array of chunks of the next level. A chunk is a subtree of height 8. At level 16, a bit vector is introduced with one bit for each possible node at this level, i.e. with $2^{16}$ bits. The bit indicates

(1) whether the prefix tree continues below this level;

(2) whether a node at this level corresponds to a prefix; or

(3) whether the trie has a leaf at a depth less than 16.

In the first case, an index to an array of chunks of the next level is required. In cases 2 and 3, an index to the next hop table needs to be associated with the bit. The arrays of chunks are based on a compact representation. The chunks of level 2 and 3 are handled the same way as in level 1 if they are densely populated. Otherwise, a less complex solution is applied that basically scans the nodes linearly. With this algorithm five memory accesses are needed per 16 bits of the address. In software, a lookup performance of about 2 million lookups per second is reported, provided that the forwarding table is located in the secondary cache. The algorithm has not been further investigated with respect to IPv6. At first glance it appears that either the levels to be distinguished need to grow or the size of the bit vector and code words increases. Update operations are not described further within the available documents. They, however, may require the complete re-construction of all vectors, arrays and tables.

A different approach on exploiting caching for route lookup maps IP addresses to *virtual memory addresses* (Chiueh & Pradhan 1999). In order to reduce virtual address space consumption, only a fixed-length portion of the IP address is mapped to a virtual address. The remaining bits need to be checked separately without the advantage of caching. A so-called host address cache with the most recently used IP addresses resides in the level 1 cache. A portion of the level 1 cache is reserved for this. In addition, the destination network address table exists which is needed in case of cache misses. In the design of that table, size is traded for lookup performance. As already seen in other approaches, the table is subdivided into three levels corresponding to the first 16 bit and two subsequent strides of 8 bit. The performance can be as high as 90 million lookups per second, if the host address cache is used optimally. Without this assumption, around 30 million lookups per second are reported.

### (*c*) *Hardware-based approaches*

The advantage of hardware-based approaches is that they may achieve higher performance due to the application of dedicated circuits and specialized memory organization compared with pure software-based solutions. However, the development of these circuits pays off only if a large quantity is installed in routers and, thus, prices are low. Some issues on hardware-based routing tables are discussed in Pei & Zukowski (1992).

#### (i) *Table lookup with CAMs*

The algorithms discussed above basically use different types of RAM (random access memory) to store entries of the routing table. CAMs (content addressable memory) appear to be an attractive alternative for hardware-based implementations of Internet routing-table lookup. In the case of RAM, an address is given to the memory, and the content of the corresponding memory is returned. With CAMs, the content is provided and the address where the content is stored is returned. The search requires one cycle only, since parallelism is applied for the search within a CAM. This makes CAMs especially suited for search purposes such as table lookups.

The routing tables in McAuley & Francis (1993) and Tantawy & Zitterbart (1992) were designed around a (potentially large) number of CAM modules. Such a solution is attractive with respect to the search task. However, the dynamics of routing tables make their usage somewhat difficult, since the addition or update of routing-table entries with CAMs cannot be easily implemented with high performance. CAMs are specialized to high-performance searching only. Moreover, a different physical entry of a single IP address is required for each subnet mask that is used. Basically, CAMs are specifically suited for the detection of fixed-length patterns. Moreover, a CAM usually provides only one search mask for all entries. In the context of the Internet this does not apply to routing-table lookup due to variable length prefixes and, consequently, numerous CAMs may be needed. Furthermore, it needs to be stressed that today's CAMs are somewhat small and very expensive compared with conventional and enhanced dynamic or static memory. However, it also needs to be mentioned that some vendors appear to have developed specialized CAMs suited for IP routing-table lookup.

(ii) *Specialized route lookup hardware*

A general alternative to the usage of CAMs can be seen in the development of specialized hardware solutions that address the specific characteristics of Internet routing-table lookup. They build upon the algorithms presented above.

The approach presented in Gupta *et al.* (1998) is a straightforward implementation using few distinct prefix lengths in hardware. The goal is to implement lookup with a high probability with a single memory access. In order to support this, pipelining is introduced. Memory is traded in favour of performance. This is driven by monetary costs and not by access costs (e.g. dynamic random access memory (DRAM) compared with cache). The basic idea of Gupta *et al.* (1998) is to provide large DRAM in which tables corresponding to dedicated prefix lengths are implemented. Since prefixes of length 24 or below represent the majority of used prefixes, all these prefixes are represented in a single table, which provides an entry for each prefix. Consequently, in most cases, the prefix can be identified with a single memory access. All prefixes that are longer than 24 bits are stored in a separate table. In case no hit was found in the first table, an index to this second table is provided that is combined with the remaining 8 bits of the address. The achieved lookup performance allows for about 20 million lookups per second when pipelining is applied. In contrast to many other approaches, updates are not very costly. The performance can further be improved if additional 'intermediate' tables are added and more pipeline stages are applied.

Refinements of Gupta *et al.* (1998) are presented in Huang *et al.* (1999), especially with respect to memory consumption. Instead of several Mbytes, only about 500 Kbytes of memory are needed and, thus, the usage of static random access memory (SRAM) is feasible. Lookup performance is increased to about 100 million lookups per second with pipelining and proper SRAMs. The difference from Gupta *et al.* (1998) is mainly based on a different usage of prefix lengths. The length of 16 bits instead of 24 bits is used for the basic table. Each lookup in this table either results in a pointer to the next hop in case of a hit or in a pointer to the associated next hop array with $2^{16}$ entries each. Huang *et al.* (1999) introduce variable offset lengths to the construction of the next hop array to reduce memory consumption by taking prefix length distribution into account. The table can be compressed to further reduce memory consumption. In the worst case, three memory accesses are needed for a single lookup. The resulting table is small enough to fit into SRAM, whereas the table produced in Gupta *et al.* (1998) is huge and can only be implemented in DRAM.

The basic data structure applied in Moestedt & Sjödin (1998) is based on a prefix trie with a few different levels (i.e. prefix lengths) being distinguished. Each node in the trie represents a prefix. Prefix expansion needs to be applied in order to convert prefixes to the predefined levels. The trie is completed by nodes that mark currently invalid prefixes. For an easy implementation in hardware, the trie is implemented as a linked list of different tables, each of them representing a prefix length. The IP address is subdivided in accordance with the prefix lengths supported in the prefix trie. The most significant bits are used to index the first table. In case of a hit, the entry points to the next hop table. In case of a miss, a pointer to the next table can be found. Pipelining can be applied which enables one lookup per memory cycle. Lookup rates of up to 50 million lookups per second can be achieved.

Subdividing the IP address into portions is also proposed in Zitterbart *et al.* (1997). Each node within a binary tree represents a part of the address and the corresponding portion of the subnet mask is associated with that node. The approach reduces the amount of memory needed to store the routing table, since parts of the tree may be shared by different addresses. Furthermore, due to the subdivision of the IP address, pipelining can be applied to increase performance. Moreover, the approach is not tight to prefix search only. Subnet masks could be selected independent of the prefix issue. With this approach, search speed nearly compares with memory speed. A performance of 10 million search steps per second is feasible with comparable slow SRAMs with 70 ns access time.

## 4. Identifier lookup

The previous section concentrated on prefix lookup for IP addresses. Due to additional functions located in IP routers, such as classification in order to support QoS requirements or multicast communication, routers may need to look up various other fields in order to identify a data flow consisting of the data exchanged between two or more application entities. Moestedt & Sjödin (1998) summarize all these examples as *identifier lookup* in contrast to address lookup, as discussed in § 3. The general difference between an identifier and an address is the hierarchical structure, which is only present in addresses, not in identifiers. Thus, identifiers do have a fixed length that needs to be investigated in contrast to variable length prefixes of addresses. Other applications of identifier lookup are multi-protocol label switching (MPLS) and layer 4 switching.

As depicted in figure 1, packet classification is another functionality that is located in the forwarding engine and, thus, may form a potential performance bottleneck. The complexity of classification depends on the number of fields that need to be considered. If RSVP (Braden 1997) is used as resource reservation protocol, various fields in the header need to be analysed, e.g. source and destination IP addresses as well as source and destination UDP ports. If IPv6 is used, a specific flow label field is sufficient to be investigated. In Nilsson & Karlsson (1999), it is argued that two pointers can simply be used to traverse the LC-trie, one for the source address and one for the destination address. Both addresses can be looked up in parallel. Hashing is proposed for the other fields (e.g. TOS field, UDP and TCP ports) since they have fixed length.

The situation becomes less complex if differentiated services as they are currently proposed in the IETF are used. Simply the per-hop behaviour (PHB) needs to be analysed for classification purposes. For traffic management purposes a lookup of the source address might also be required.

An increasingly important issue is the lookup of multicast IP addresses. Prefix search cannot be applied since they do not provide a prefix. The same holds for anycast addresses introduced by IPv6. In Nilsson & Karlsson (1999), however, it is suggested to view multicast IP addresses as prefixes of 32 bit length. Another difference to unicast addresses is the fact that multiple next-hop pointers may be associated with a single multicast address. A hardware-based approach for identifier lookup is proposed in Moestedt & Sjödin (1998). It simply uses various hash tables and hash functions in parallel with the restriction that an identifier can be at most located in one hash table.

## 5. Summary and outlook

A brief introduction into the basic building blocks of a router has been presented. The main focus of the discussion was on routing-table lookup since it may considerably limit router performance. Especially, the number of memory accesses, as well as the amount of memory needed, are important design criteria. Several algorithms for efficient routing-table lookup were presented: software-based and hardware-based approaches as well as approaches exploiting cache technology. With these new algorithms that have been developed over the last few years, an important performance boost can be observed. This may reduce the necessity of other approaches, such as IP switching and MPLS. Highly efficient routing-table lookup can be implemented in various ways. Software implementations provide more flexibility. On the other hand, with hardware implementations, memory can be organized according to the specific requirements. Moreover, even with cheap DRAM technology, high performance can be achieved with the requirement of large memory sizes.

Besides increasing the performance of the forwarding engine for backbone and enterprise routers, introducing application service provisioning into routers is currently discussed. This may lead to so-called programmable or active networks. Considerable changes in router design may be needed to support this direction, especially considering the operating system as well as the application programming interface (API). The main difference introduced with these technologies can be seen in the fact that service provisioning and router design and deployment are no longer tight together. So-called rapid service creation is supported, which provides dynamic deployment of services at strategic points in the network. Programs to be executed for service provisioning can either be downloaded into the router in advance or carried with the data units during data transfer.

## References

Braden, R. (ed.) 1997 *Resource ReserVation Protocol. Version 1: functional specification.* RFC 2205.

Cheung, G. & McCanne, S. 1999 Optimal routing table design for IP address lookups under memory constraints. In *IEEE Infocom '99*, pp. 1437–1444.

Chiueh, T. & Pradhan, P. 1999 High-performance IP routing table lookup using CPU caching. In *IEEE Infocom '99*, pp. 1421–1428.

Cormen, T., Leiserson, C. & Rivest, R. 1990 *Introduction to algorithms.* McGraw-Hill.

Degermark, M., Brodnik, A., Carlsson, S. & Pink, S. 1997 Small forwarding tables for fast routing lookups. In *ACM SIGCOMM '97, Cannes, France, September 1997*, pp. 3–14.

Gupta, P., Lin, S. & McKeown, N. 1998 Routing lookups in hardware at memory access speed. In *IEEE Infocom '98*, pp. 1240–1247.

Huang, N., Zhao, S., Pan, J. & Su, C. 1999 A fast IP routing lookup scheme for gigabit switching routers. In *IEEE Infocom '99*, pp. 1429–1436.

Keshav, S. & Sharma, R. 1998 Issues and trends in router design. *IEEE Commun. Mag.* **36**, 144–151.

Knuth, D. 1973 *The art of computer programming*, vol. 3, *Sorting and searching.* Addison-Wesley.

Koufopavlou, O., Tantawy, A. & Zitterbart, M. 1994 A comparison of gigabit router architectures. *J. High Speed Networks* **3**, 209–232.

Labovitz, C., Malan, G. & Jahanian, F. 1997 Internet routing instability. In *ACM SIGCOMM '97, Cannes, France, September 1997.*

Lampson, B., Srinivasan, V. & Varghese, G. 1998 IP lookups using multiway and multicolumn search. In *IEEE Infocom '98*, pp. 1248–1256.

McAuley, A. & Francis, P. 1993 Fast routing table lookups using CAMs. In *IEEE Infocom '93, San Francisco*, pp. 1382–1391.

Metz, C. 1998 IP routers: new tool for gigabit networking. In *IEEE Internet Computing, November/December 1998*, pp. 14–18.

Moestedt, A. & Sjödin, P. 1998 IP address lookup in hardware for high-speed routing. In *Hot Interconnects, Stanford, CA, USA*.

Nilsson, S. & Karlsson, G. 1998 Fast address lookup for internet routers. In *IFIP Conf. on Broadband Communications, Stuttgart, Germany, April 1998*, pp. 11–22.

Nilsson, S. & Karlsson, G. 1999 IP-address lookup using LC-tries. *IEEE J. Selected Areas Commun.* **17**, 1083–1092.

Partridge, C. 1998 Designing and building gigabit and terabit Internet routers. In *ACM Sigcomm '98, Vancouver, Canada, September 1998*, tutorial 2.

Partridge, C., *et al*. 1998 A 50 Gb/s IP router. *IEEE/ACM Trans. Networking* **6**, 237–248.

Pei, T. & Zukowski, C. 1992 Putting routing tables in silicon. *IEEE Network Mag* **6**, 42–50.

Srinivasan, V. & Varghese, G. 1998 Fast address lookups using controlled prefix expansion. In *ACM Sigmetrics '98*.

Tantawy, A. & Zitterbart, M. 1992 Multiprocessing in high performance IP routers. In *IFIP Workshop on Protocols for High Speed Networks, Stockholm, Sweden, May 1992*, pp. 235–254.

Waldvogel, M., Varghese, G., Turner, J. & Plattner, B. 1997 Scalable high speed IP routing lookups. In *ACM Sigcomm '97, Cannes, France, September 1997*, pp. 25–36.

Wittmann, R. & Zitterbart, M. 2000 *Multicast—applications and protocols*. San Francisco: Morgan Kaufman.

Wright, G. & Stevens, W. 1995 *TCP/IP illustrated*, vol. 2: *The implementation*. Addison-Wesley.

Zitterbart, M., Harbaum, T., Meier, D. & Brökelmann, D. 1997 Efficient routing table lookup for IPv6. In *IEEE Workshop on High Performance Communication Systems, Chalkidiki, Greece, June 1997*, pp. 1–9.