Fast address lookup for Internet routers

Stefan Nilsson
Department of Computer Science, Helsinki University of Technology
P.O. Box 1100, FIN-02015 HUT, Finland
e-mail: `Stefan.Nilsson@hut.fi`

and

Gunnar Karlsson
Swedish Institute of Computer Science
P.O. Box 1263, SE-164 29 Kista, Sweden
e-mail: `gk@sics.se`

Abstract

We consider the problem of organizing address tables for internet routers to enable fast searching. Our proposal is to to build an efficient, compact and easily searchable implementation of an IP routing table by using an *LC-trie*, a trie structure with combined path and level compression. The depth of this structure increases very slowly as function of the number of entries in the table. A node can be coded in only four bytes and the size of the main search structure never exceeds 256 kB for the tables in the US core routers. We present a software implementation that can sustain approximately half a million lookups per second on a 133 MHz Pentium personal computer, and two million lookups per second on a more powerful SUN Sparc Ultra II workstation.

1 Introduction

The high and steadily increasing demand for Internet service has lead to a new version of the internet protocol to avoid the imminent starvation of the present address space. The new version 6 being standardized will replace the current 32-bit addresses with a virtually inexhaustible 128-bit address space. Another consequence of the growth is the need for higher transmission and switching capacities. Links may readily be upgraded in speed and added in number to increase the network's transmission capacity. Equipment for the synchronous digital hierarchy is now being deployed to provide link rates of 155.5 Mb/s, 622 Mb/s and 2.5 Gb/s. Increases in switching capacity are not equally accessible and the bottlenecks in today's Internet are chiefly the routers.

With the upgrade of the transmission infrastructure follows that routers have to interconnect both more and faster links. But few if any of today's routers can provide switching with aggregate throughput of 50 to 100 Gb/s, as needed for a few tens of ports with bit rates of up to 2.5 Gb/s. Yet this can be accomplished by fast packet switches, as proven by the availability of high-capacity ATM switches on the market. It is important to note that there are no fundamental limits as to why the same performance cannot be expected for IP routers. The problems are rather pertaining to design choices and to practical implementation.

In this paper we consider address lookup as one of the basic functions of a router, along with buffering, scheduling and switching of packets. All of these functions have to be studied in order to increase throughput. The variable lengths of IP packets complicate buffer management and scheduling compared to protocol-data units of fixed length, but an average packet length far longer than a cell may compensate by reducing the number of such operations per second for a given link rate. Switching of packets could be made in parallel if routers incorporate space-division interconnection fabrics rather than the customarily used data buses. The remaining function that has been deemed critical for the viability of multi-Gb/s routers is the address lookup [11].

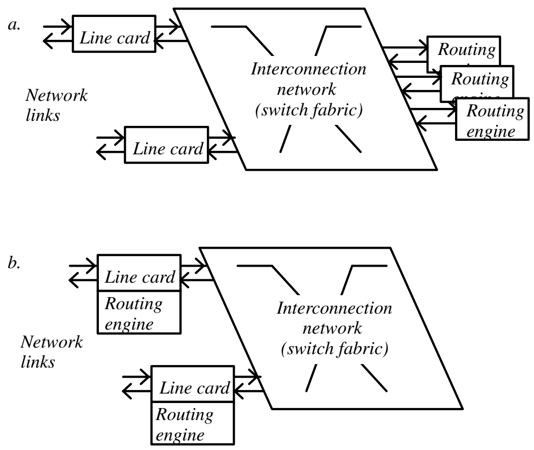

Figure 1: Two router architectures: a. one or more centralized routing engines, b. one routing engine per port.

The address lookup has traditionally been performed centrally by a processor that serves all input ports of the router (Figure 1a). An input module sends the header of a packet to the processor (often called a forwarding or routing "engine"). The processor uses the destination address to determine the output port for the packet and its next-hop address (it also modifies the header). When the header has been returned along with the routing information, the input module forwards the packet across the interconnection network to the designated output port. The next-hop address is used at the output port to determine the link address of the packet, in case the link is shared by multiple parties (as an Ethernet, Token Ring or FDDI network), and it is consequently not needed if the output connects to a point-to-point link.

When a single routing engine cannot keep up with the requests from the input ports, the remedy has been to employ multiple engines in parallel. An advantage of having engines serving as one pool is the ease of balancing loads from the ports when they have different speeds and utilization levels. The disadvantage is the round-trip transfer of packet headers across the interconnection network. This load is concentrated to a few outputs and can therefore be problematic for a space-division network. When the network links are upgraded to higher bit rates, the natural modification is to place one routing engine on each input module (Figure 1b) simply because there will not be much idle time in the processing to share with other ports. This means that each engine only needs to offer a rate of address resolutions appropriate for the link, and that the interconnection network is not loaded unnecessarily by the transfer of packet headers (which will become a multiple of 40 bytes long by the introduction of IP version 6). To support a 2.5 Gb/s link means that 1.25 million lookups should be sustained on the average for today's mean packet size of 250 bytes.

In this paper we present an efficient organization of IP routing tables that allows fast address lookup in software. Our implementation can process approximately half a million addresses per second on a standard 133 MHz Pentium personal computer. The performance scales nicely to fully exploit faster memory and processor clock rates, which is illustrated by the fact that a SUN Sparc Ultra II workstation can perform 2 million lookups per second. The advantage with a software solution, such as ours, is that the processor can run separate lookup routines for multicast and flow-based routing (based on source address and a 24-bit long flow label in IP version 6). It can also process various routing options as specified by extension headers. The processor's caching protocol automatically exploit temporal correlations in packet destinations by keeping the most accessed parts of the data structure in the on-chip cache. A low-cost implementation could consist of a field-programmable gate array, instead of a microprocessor, and less than a megabyte of random-access memory.

2 Address lookup for the Internet protocol

An IP address has traditionally consisted of one to five bits specifying an address class, a network identifier and a host identifier. The network identifier points to a network within an internet, and the host identifier points to a specific computer on that network. Routing is based on the network identifier solely. Each of the classes A through C had a predetermined length of the network identifier which made the address lookup in routers straightforward.

This class-based structure has been abandoned in favor of the classless inter-domain routing (CIDR) [8]. An IP address can now be split into network and host identifiers at almost any point. Address lookup is done by matching a given address to bit strings of variable lengths (prefixes) that are stored in the routing table. If more than one valid prefix matches an address in a routing table, the information associated with the longest prefix is used to forward the packet. For instance, the address 222.21.67.68 matches 222.21.64.0 with 18 bits and 222.16.0.0 with 12 bits. The first of these two is longer and should consequently be used for forwarding the packet.

The matching prefix is not necessarily the same in all routers for one and the same IP address. This allows the tables to be smaller in size since a large number of destinations can be represented by a short network identifier. Most routers use a default route which is given by a prefix of length zero. It matches all addresses and is used if no other prefix match. The core routers in the Internet backbone are not allowed to use default routes and their address tables tend to be larger than in other routers. We have used tables from these routers in our evaluation to ensure that we test with realistic worst cases. The result of the lookup is the port number and next-hop address. A next-hop address is not needed for a point-to-point link and the corresponding routing-table entry would only need to contain the port number. Even when next-hop addresses are needed, there are usually fewer distinct such addresses than there are entries in the routing table. The table can therefore contain a pointer to an array which lists the next-hop addresses in use.

The address structure for IP version 6 is not fully decided even for unicast addresses. It is, however, suggested to keep the variable-length network identifiers (or subnetwork identifiers) [10]. Thus, a subnetwork can be identified by some n bits in a router, while the remaining $128 - n$ bits form the interface identifier (replacing the host identifier of version 4). Our data structure has been designed to handle version 6 addresses when needed.

There has been remarkably little interest in the organization of routing tables both for hardware and software based searches during the last years. Hardware implementation, which we do not consider here, is discussed in [17]. We therefore place our proposal in relation only to the three most recent works from the literature. Our search structure and implementation is akin to and has been inspired by the work of Brodnik et al. [4]. They use a different data structure from ours and are concerned with the size of the trie to ensure that it fits in a processor's on-chip cache memory. As a consequence, it is not immediately clear if the structure will scale to the longer addresses of IP version 6. A main idea of their work is to quantify the prefix lengths to levels of 16, 24 and 32 bits. This exploits the old class-based address structure but may suffer from the ongoing redistribution of addresses that tends to smooth the distribution of the prefix lengths. Rather than expanding the prefixes to a few levels, we use level-compression to reduce the size of the trie. Thus, we obtain similar performance in our simulations with a more general structure and without making any assumptions about the address structure.

Waldvogel et al. [15] use a different technique. Prefixes of fixed lengths are stored in separate hash tables and the search operation is implemented using binary search on the number of bits. Using 16 and 24 bits for the first two hash table lookups this search strategy is very efficient for current routing tables. However, for a routing table with longer prefixes and a smother distribution of prefix lengths this approach may not be as attractive.

The work on prefix matching by Doeringer, Karjoth, and Nassehi [6] uses a trie structure. One of their concerns is to allow fully dynamic updates. This results in a large space overhead and less than optimum performance. The nodes of the trie structure contains five pointers and one index. More general but slightly dated works on Gb/s routers that could be of interest to the reader are presented in [3, 14, 16].

nbr	string
0	0000
1	0001
2	00101
3	010
4	0110
5	0111
6	100
7	101000
8	101001
9	10101
10	10110
11	10111
12	110
13	11101000
14	11101001

Figure 2: Binary strings to be stored in a trie structure.

3 Level-compressed tries

The *trie* [7] is a general purpose data structure for storing strings. The idea is very simple: each string is represented by a leaf in a tree structure and the value of the string corresponds to the path from the root of the tree to the leaf. Consider a small example. The binary strings in Figure 2 correspond to the trie in Figure 3a. In particular, the string `010` corresponds to the path starting at the root and ending in leaf number 3: first a left-turn (`0`), then a right-turn (`1`), and finally a turn to the left (`0`). For simplicity, we will assume that the set of strings to be stored in a trie is prefix-free, no string may be a proper prefix of another string. We postpone the discussion of how to represent prefixes to the next section.

This simple structure is not very efficient. The number of nodes may be large and the average depth (the average length of a path from the root to a leaf) may be long. The traditional technique to overcome this problem is to use *path compression*, each internal node with only one child is removed. Of course, we have to somehow record which nodes are missing. A simple technique is to store a number, the *skip value*, in each node that indicates how many bits that have been skipped on the path. A path-compressed binary trie is sometimes referred to as a Patricia tree [9]. The path-compressed version of the trie in Figure 3a is shown in Figure 3b. The total number of nodes in a path-compressed binary trie is exactly $2n - 1$, where n is the number of leaves in the trie. The statistical properties of this trie structure are very well understood [5, 12]. For a large class of distributions path compression does not give an asymptotic reduction of the average depth. Even so, path compression is very important in practice, since it often gives a significant overall size reduction.

One might think of path compression as a way to compress the parts of the trie that are sparsely populated. *Level compression* [1] is a recently introduced technique for compressing parts of the trie that are densely populated. The idea is to replace the i highest complete levels of the binary trie with a single node of degree 2^i; this replacement is performed recursively on each subtrie. The level-compressed version, the *LC-trie*, of the trie in Figure 3b is shown in Figure 3c.

For an independent random sample with a density function that is bounded from above and below the expected average depth of an LC-trie is $\Theta(\log^* n)$, where $\log^* n$ is the iterated logarithm function, $\log^* n = 1 + \log^*(\log n)$, if $n > 1$, and $\log^* n = 0$ otherwise. For data from a Bernoulli-type process with character probabilities not all equal, the expected average depth is $\Theta(\log \log n)$ [2].

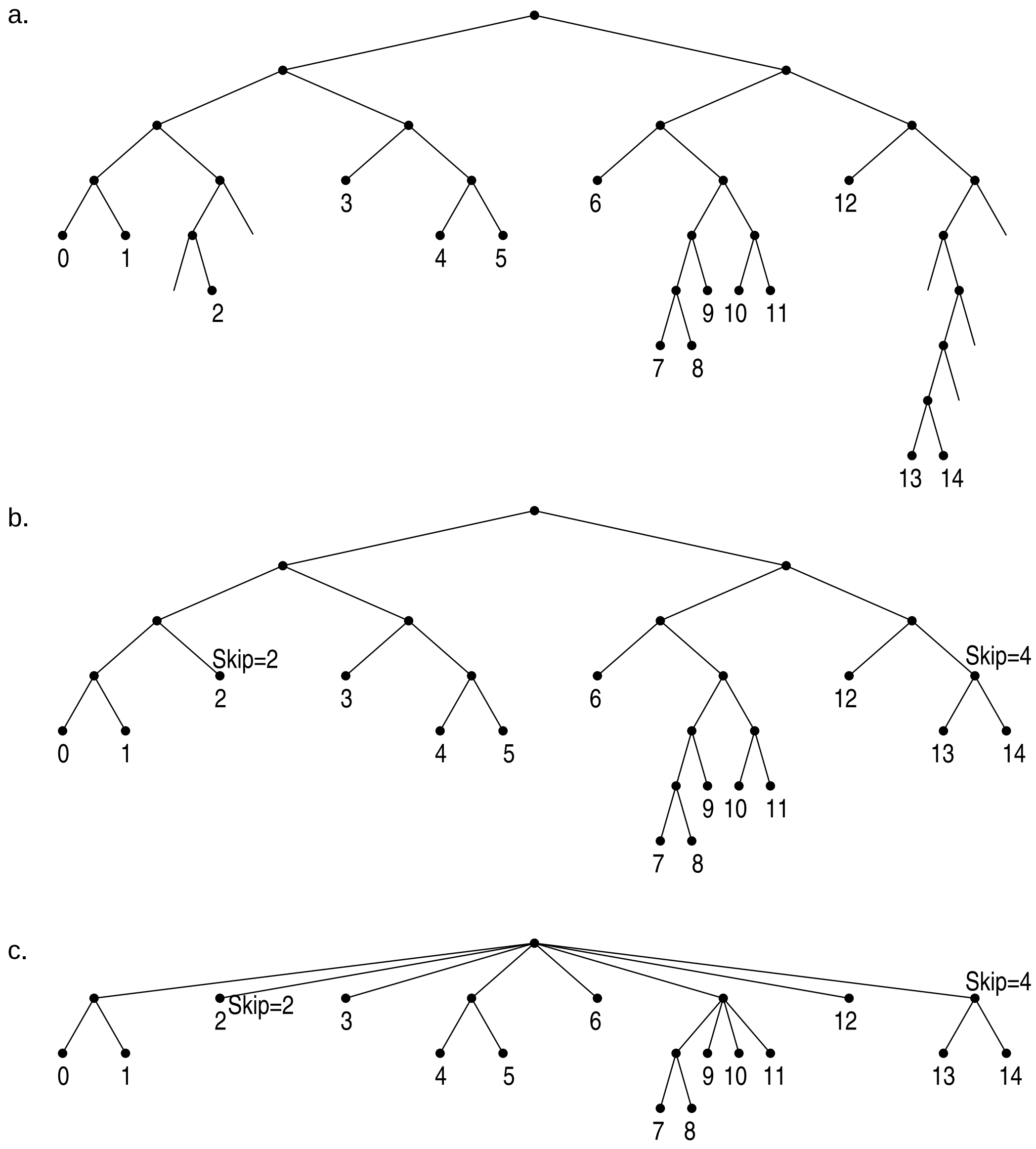

Figure 3: a. Binary trie, b. Path-compressed trie, c. LC-trie.

	branch	skip	pointer
0	3	0	1
1	1	0	9
2	0	2	2
3	0	0	3
4	1	0	11
5	0	0	6
6	2	0	13
7	0	0	12
8	1	4	17
9	0	0	0
10	0	0	1
11	0	0	4
12	0	0	5
13	1	0	19
14	0	0	9
15	0	0	10
16	0	0	11
17	0	0	13
18	0	0	14
19	0	0	7
20	0	0	8

Figure 4: Array representation of the LC-trie in Figure 3c.

Uncompressed tries and path-compressed tries both have expected average depth $\Theta(\log n)$ for these distributions.

If we want to achieve the efficiency promised by these theoretical bounds, it is of course important to represent the trie efficiently. The standard implementation of a trie, where a set of children pointers are stored at each internal node is not a good solution, since it has a large space overhead. This may be one explanation why trie structures have traditionally been considered to require much memory.

A space efficient alternative is to store the children of a node in consecutive memory locations. In this way, only a pointer to the leftmost child is needed. In fact, the nodes may be stored in an array and each node can be represented by a single word. In our implementation, the first 5 bits represent the *branching factor*, the number of descendants of the node. This number is always a power of 2 and hence, using 5 bits, the maximum branching factor that can be represented is 2^{31}. The next 7 bits represent the skip value. In this way, we can represent values in the range from 0 to 127, which is sufficient for IP version 6 addresses. This leaves 20 bits for the pointer to the leftmost child and hence, using this very compact 32 bit representation, we can store at least 2^{19} strings. Figure 4 shows the array representation of the LC-trie in Figure 3c, each entry represents a node. The nodes are numbered in breadth-first order starting at the root. The number in the branch column indicates the number of bits used for branching at each node. A value $k \geq 1$ indicates that the node has 2^k children. The value $k = 0$ indicates that the node is a leaf. The next column contains the skip value, the number of bits that can be skipped during a search operation. The value in the pointer column has two different interpretations. For an internal node, it is used as a pointer to the leftmost child; for a leaf it is used as a pointer to a base vector containing the compleat strings.

The search algorithm can be implemented very efficiently. Let `s` be the string searched for and let `EXTRACT(p, b, s)` be a function that returns the number given by the `b` bits starting at position `p` in the string `s`. We denote the array representing the tree by `T`. The root is stored in `T[0]`.

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

Note that the address returned only indicates a possible hit; the bits that have been skipped during the search may not match. Therefore we need to store the values of the strings separately and perform one additional comparison to check whether the search actually was successful.

As an example we search for the string `10110111`. We start at the root, node number 0. We see that the branching value is 3 and skip value is 0 and therefore we extract the first three bits from the search string. These 3 bits have the value 5 which is added to the pointer, leading to position 6 in the array. At this node the branching value is 2 and the skip value is 0 and therefore we extract the next two bits. They have the value 2. Adding 2 to the pointer we arrive at position 15. At this node the branching value is 0, which implies that it is a leaf. The pointer value 5 gives the position of the string in the base vector. Observe that it is necessary to check whether this constitutes a true hit. We need to compare the first 5 bits of the search string with the first 5 bits of a value stored in the base vector in the position indicated by the pointer (10) in the leaf. In fact, our table (Figure 2) contains a prefix `10110` matching the string and the search was successful.

4 Representation of a routing table

The routing table consists of four parts. At the heart of the data structure we have an *LC-trie* implemented as discussed in the previous section. The leaves of this trie contain pointers into a *base vector*, where the complete strings are stored. Furthermore we have a *next-hop table*, an array containing all possible next-hops addresses, and a special *prefix vector*, which contains information about strings that are proper prefixes of other strings. This is needed because internal nodes of the LC-trie do not contain pointers to the base vector.

The base vector is typically the largest of these structures. Each entry contains a string. In the current implementation it occupies 32 bits, but it can of course easily be extended to the 128 bits required in IP version 6. Notice that it is not necessary to store the length of the string, since the length will be known to the search routine after the traversal of the LC-trie. Each entry also contains two pointers: one pointer into the next-hop table and one pointer into the prefix table. The search routine follow the next-hop pointer if the search was successful. If not, the search routine tries to match a prefix of the string with the entries in the prefix table. The prefix pointer has the special value -1 if no prefix of the string is present.

The prefix table is also very simple. Each entry contains a number that indicates the length of the prefix. The actual value need not be explicitly stored, since it is always a proper prefix of the corresponding value in the base vector. As in the base vector each entry also contains two pointers: one pointer into the next-hop table and one pointer into the prefix table. The prefix pointer is needed, since it might happen that a path in the trie contains more than one prefix.

The main part of the search is spent within the trie. In our experiments the average depth of the trie is typically close to 6 and one memory lookup is performed for each node traversed. The second step is to access the base vector. This accounts for one additional memory lookup. If the string is found at this point, one final lookup in the next-hop table is made. This memory access will be fast since the next-hop table is typically very small, in our experiments less than 60 entries, and can be expected to reside in cache memory.

Site	Routing Entries	Next-Hops	Number of Entries			Aver. Depth	Lookups	
			Trie	Base	Prefix		Sparc	PC
Mae East	39 819	56	62 991	38 380	1439	5.92	1.5	0.42
Mae West	14 618	55	24 143	14 291	327	6.08	2.1	0.48
AADS	20 299	19	33 159	19 846	453	6.28	1.6	0.45
Pac Bell	20 611	3	33 591	20 171	440	6.32	1.9	0.45
FUNET	41 578	20	63 623	39 765	1813	8.31	2.3	0.58

Table 1: Experimental data. The speed is measured in million lookups per second.

Finally, if the string searched for does not match the string in the base vector, an addition lookup in the prefix vector will have to be made. Also this vector is typically very small, in our experiments it contains less than 2000 entries, and it is rarely accessed more than once per lookup. In all the routing tables that we have examined we have found only a few multiple prefixes. Conceptually a prefix corresponds to an exception in the address space. Each entry in the routing table defines a set of addresses that share the same routing table entry. In such an address set, a longer match corresponds to a subset of addresses that should be routed differently. Overlapping prefixes could also be avoided by expanding the shorter prefixes. For instance, instead of having the overlapping prefixes 101 and 10111 in a table, 101 could be expanded to 10100, 10101 and 10110 which all would point to the same routing information. The expansion increases the trie size and we have therefore chosen to use a separate prefix vector. Furthermore, we expect this special case to disappear with the introduction of IP version 6 since the new address space will be so large as to make it possible to allocate addresses in a strictly hierarchical fashion.

5 Results

The measurements were performed on two different machines: a SUN Ultra Sparc II with two 296-MHz processors and 512 MB of RAM, and a personal computer with a 133-MHz Pentium processor and 32 MB of RAM. The programs are written in the C programming language and have been compiled with the `gcc` compiler using optimization level `-O4`. We used routing tables provided by the Internet Performance Measurement and Analysis project (URL `http://www.merit.edu/ipma`). We have used the routing tables for Mae East, Mae West, AADS, and Pac Bell from the 24th of August, 1997. We did not have access to the actual traffic being routed according to these tables and therefore the traffic is simulated: we simply use a random permutations of all entries in a routing table. The entries were extended to 32 bits numbers by adding zeroes (this should not affect the measurements, since these bits are never inspected by the search routine). We have, however, been able to test our algorithm on a routing table with recorded traces of the actual packet destinations. The router is part of the Finnish University and Research Network (FUNET). Real traffic gives better results than runs of randomly generated destinations and owes to dependencies in the destination addresses. The time measurements have been performed on sequences of lookup operations, where each lookup includes fetching the address from an array, performing the routing table lookup, accessing the nexthop table and assigning the result to a volatile variable.

Some of the entries in the routing tables contain multiple next-hops. In this case, the first one listed was selected as the next-hop address for the routing table, since we only considered one next-hop address per entry in the routing table. There were also a few entries in the routing tables that did not contain a corresponding next-hop address. These entries were routed to a special next-hop address different from the ones found in the routing table.

Table 1 gives a summary of the results. It shows the number of unique entries in the routing table, the number of next-hop addresses, the size of our data structure, the average of the trie, and the number of lookups measured in million lookups per second. In our current implementation an entry in the trie occupies 4 bytes, while the entries in the base vector and the prefix vector occupy 12 bytes each. The largest table, FUNET, still occupies less than 0.8 MB of memory, of which the trie-part is less than 256 kB.

The average throughput corresponding to the number of lookups per second is found by multiplying it with the average packet size, which currently is around 250 bytes. The routing system could be modeled as a G/G/1 queue in order to find the number of pending lookups as a function of the routing system's load. The arrival process would be given by the trimodal packet-length distribution (peaks around 40, 550 and 1500 bytes) divided by the link rate and the service distribution by the lookup times.

6 Final remarks and conclusions

The Internet has in practice become the long-sought broadband integrated services digital network, in the meaning of a global communication infrastructure for multimedia services [13]. The intention was, however, that the B-ISDN should have been based on the asynchronous transfer mode rather than on IP. Now when that raison d'être for ATM is disappearing there is naturally great interest in salvaging the enormous investment that has been made by finding a role for ATM as carrier of IP packets. Thus, we find IP routing combined with ATM switching. The idea is to establish a virtual circuit for a flow of packets in order to amortize the cost of IP address lookup over several packets [11]. We believe that pure IP routing is a competitive alternative and have shown that software-based address lookup can be performed sufficiently fast for multi-Gb/s systems.

We have demonstrated how IP routing tables can be succinctly represented and efficiently searched by structuring them as level-compressed tries. Our data structure is fully general and does not rely on the old class-based structure of the address format for its efficiency. Even though the data structure does not make explicit assumptions about the distribution of the address, it does adapt gracefully: path compression compacts the sparse parts of the trie and level compression packs the dense parts. The average depth of the trie grows very slowly. This is in accordance with theoretical results. Recall that the average depth of an LC-trie is $\Theta(\log\log n)$ for a large class of distributions. Actually, our experiments show that in some cases the average depth is smaller for a larger table. This can be explained by the fact that a larger table might be more densely populated and hence the level compression will be more efficient. The inner loop of the search algorithm is very tight; it contains only one addressing operation and a few very basic operations, such as shift and addition. Furthermore, the trie can be stored very compactly, using only one 32-bit machine word per node. In fact, the largest trie in our experiments consumed less than 256 kB of memory. The base vector is larger, but is only accessed once per lookup.

The results show that the routinely made statements about possible processing speeds for IP addresses, such as those put forward in [11], are not valid. In many cases, for instance in [4], the trie implementations cited do not reflect the state of the art. Thus, we argue that our new data structure is superior to earlier presented software methods for the organization of IP routing tables.

Acknowledgments

We thank Erja Kinnunen and Pekka Kytölaakso of the Center for Scientific Computing at Helsinki University of Technology for providing the FUNET routing table and associated packet traces.

This research was done when G. Karlsson was visiting professor at the Telecommunication Software and Multimedia Laboratory at the Helsinki University of Technology. This support is gratefully acknowledged.

References

[1] Arne Andersson and Stefan Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993.

[2] Arne Andersson and Stefan Nilsson. Faster searching in tries and quadtrees – an analysis of level compression. In *Proceedings of the Second Annual European Symposium on Algorithms*, pages 82–93, 1994. LNCS 855.

[3] A. Asthana, C. Delph, H. V. Jagadish, and P. Krzyzanowski. Towards a gigabit IP router. *Journal of High Speed Networks*, 1(4):281–288, 1992.

[4] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. *ACM Computer Communication Review*, 27(4):3–14, October 1997.

[5] Luc Devroye. A note on the average depth of tries. *Computing*, 28(4):367–371, 1982.

[6] Willibald Doeringer, Günter Karjoth, and Mehdi Nassehi. Routing on longest-matching prefixes. *IEEE/ACM Transanctions on Networking*, 4(1):86–97, February 1996.

[7] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[8] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. Request for Comments: 1519, September 1993.

[9] Gaston H. Gonnet and Ricardo A. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, second edition, 1991.

[10] R. Hinden and S. Deering. IP version 6 addressing architecture. Request for Comments: 1884, December 1995.

[11] P. Newman, G. Minshall, T. Lyon, and L. Huston. IP switching and gigabit routers. *IEEE Communications Magazine*, 35(1):64–69, January 1997.

[12] Bonita Rais, Philippe Jacquet, and Wojciech Szpankowski. Limiting distribution for the depth in Patricia tries. *SIAM Journal on Discrete Mathematics*, 6(2):197–213, 1993.

[13] W. David Sincoskie. Viewpoint: Broadband ISDN is happening – except it's spelled IP. *IEEE Spectrum*, 34(1):32–33, 1997.

[14] A. Tantawy, O. Koufopavlou, and M. Zittertbart. On the design of a multigigabit IP router. *Journal of High Speed Networks*, 3(3), 1994.

[15] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. *ACM Computer Communication Review*, 27(4):25–36, October 1997.

[16] R. J. Walsh and C. M. Özveren. The gigaswitch control processor. *IEEE Network*, 9(1):36–43, January/February 1995.

[17] C. A. Zukowski and T. Pei. Putting routing tables into silicon. *IEEE Network*, pages 42–50, January 1992.