Computer Communications 25 (2002) 303–312

computer communications

www.elsevier.com/locate/comcom

High-performance IP routing table lookup

Pi-Chung Wang*, Chia-Tai Chan, Yaw-Chung Chen

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu 30050, Taiwan, ROC

Received 11 September 2000; revised 8 May 2001; accepted 29 May 2001

Abstract

Nowadays, the commonly used table lookup scheme for IP routing is based on the so-called classless interdomain routing (CIDR). With CIDR, routers must find out the best matching prefix (BMP) for IP packets forwarding, which complicates the IP lookup. Currently, this process is mainly done in software and several schemes have been proposed for hardware implementation. Since the IP lookup performance is a major design issue for the new generation routers, in this article we propose a fast IP address lookup scheme, which significantly reduces the forwarding table to fit into SRAM with very low cost. It can also be implemented in hardware using the pipeline technique. By using our proposed method, the required memory space can be 90–170 Kb less than the previous scheme and three memory accesses for a lookup in the worst case. When implemented in hardware pipeline architecture, our mechanism can achieve one routing lookup per memory access.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: IP routing; Classless interdomain routing; Pipeline technique

1. Introduction

Speeding up the packet forwarding in the Internet backbone requires high-speed transmission links and high performance routers. The transmission technology keeps evolving and provision of gigabit fiber links is commonly available. Consequently, the key to increase the capacity of the Internet lies in fast routers [5]. A multi-gigabit router must have enough internal bandwidth to switch packets between its interfaces at multi-gigabit rates and enough packet processing power to forward multiple millions of packets per second (MPPS) [6]. Switching in the router has been studied extensively in the ATM community and solutions for fast packet processing are commercially available. As a result, the major obstacle remaining for the high performance router design is the slow, multi-memory-access IP lookup procedure. A router must search forwarding tables using the DA (destination address) as the key, and determine which table entry represents the best route to forward the packet to its destination. Since the development of CIDR in 1993 [1], IP routes have been identified by a ⟨routing prefix, prefix length⟩ pair, where the prefix length varies from 1 to 32 bits. Due to the fact that table entries have variable lengths and that multiple entries may represent the valid routes to the same destination, the search may be time consuming, especially in a backbone router with a large number of table entries.

While designing the simple and fast routing lookup scheme, the following considerations should be considered simultaneously.

- *Simplicity*: It should make the data structure and the routing lookup operation as simple as possible, such as using pre-computation to avoid search complexity shown in Ref. [13].
- *Reducing routing-table space*: If we can reduce routing-table size to fit into the high speed SRAM, the performance can be promoted significantly.
- *Parallelism*: To provide next generation tera-bit/s high-speed transmission, pipelining will play an important role. Therefore, another issue is to exploit the high parallelism within the routing-table lookups.

Although the existing related works have their advantages, however, those approaches either use complicated data structures, which result in high complexity for updating the routing entries, such as Refs. [12,13], or they are not scalable to IPv6, such as Refs. [7–9]. This work presents a fast longest-prefix matching scheme for IP switch routers. Our scheme needs relatively small SRAM and can be implemented using hardware pipeline architecture. Based on our proposed scheme, the forwarding table would be small enough to fit into SRAM at a very low cost. The required

* Corresponding author. Tel.: +886-35-731851; fax: +886-35-727842.
E-mail address: pcwang@csie.nctu.edu.tw (P.-C. Wang).

0140-3664/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0140-3664(01)00360-7

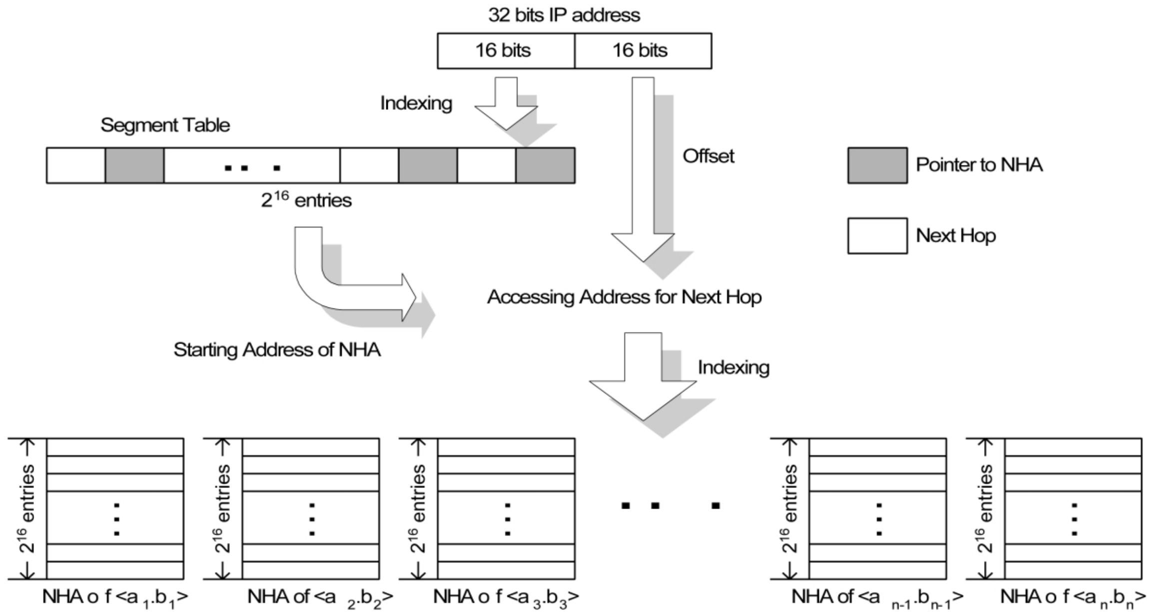


Fig. 1. Indirect-lookup mechanism.

memory space of the resulting forwarding table can be 90–170 Kb less than the previous scheme. Most of the address lookups can be accomplished in single memory access. In the worst case, the number of memory accesses for a lookup is three. When implemented in hardware, the proposed scheme can achieve one routing lookup within one memory access time.

The rest of the paper is organized as follows. Section 2 describes the previous work. The proposed longest prefix matching scheme and the hardware architecture is presented in Section 3. The performance analysis of the proposed scheme is addressed in Section 4. Finally, a concluding remark is given in Section 5.

2. Related works

Several works have been proposed with novel data structures to reduce the complexity of longest-prefix matching lookups [11–13]. These data structures and their accompanying algorithms are designed primarily for software implementation, they are usually unable to complete a lookup in few memory-access-time. The most straightforward way to implement a lookup scheme is to build a forwarding table for all possible IP addresses. However, the size of the forwarding table (next-hop array; NHA) is too large (2^{32} entries) to be practical.

To reduce the size of the forwarding table, an indirect

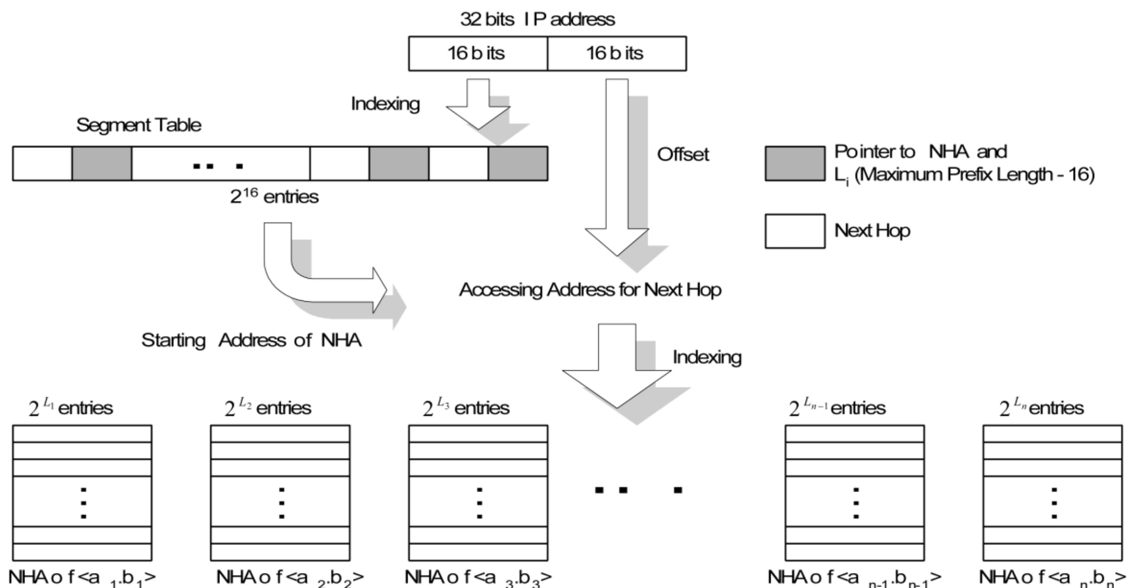


Fig. 2. Indirect-lookup mechanism with variable offset length.

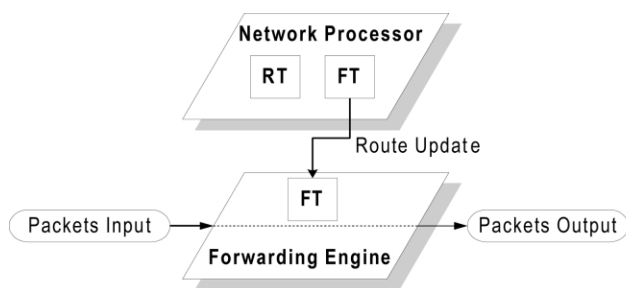


Fig. 3. Architecture of the IP router.

lookup mechanism, as shown in Fig. 1 [8], can be employed. Each IP address is split into two parts: the segment (16-bit) and the offset (16-bit). The segment table has 64 K entries which record either the next hop of the route or pointers pointing to the associated NHAs. Each NHA consists of 64 K entries, each entry records the next hop for the destination IP address. This scheme uses a maximum of two memory accesses for a lookup in a 33-Mb forwarding table. By adding an intermediate-length table, the forwarding table can be reduced to 9 Mb; however, the maximum number of memory accesses for a lookup would be increased to three. When implemented in hardware pipeline, the scheme can accomplish one route lookup every memory access and achieve up to 20 million lookups per second.

In Ref. [9], Huang et al. further reduced the size of the associated NHA by considering the distribution of the prefixes belonging to the same segment, as demonstrated in Fig. 2. With 58,000 routing prefixes, it results in about 1.3 million entries in the NHA. With 1 b per entry, the total required memory size (including segment table) is more than 1.5 Mb, and the required memory accesses is two. By using the compression, the required memory size can be reduced to 590 Kb, but the number of memory accesses is increased to three. The time complexity for building the so-called Code Word Array (CWA) and the compressed NHA (CNHA) is $O(n \log n)$ [9] where n denotes the number of prefixes in a segment. However, due to the characteristic of compression, it is necessary to rebuild the CNHA for updating the forwarding table. Since the routing updates may occur every few seconds, the performance might degrade severely due to the memory bandwidth contention.

3. The proposed scheme

Fig. 3 conceptually depicts the architecture of the IP router, which mainly consists of a network processor and a forwarding engine. The forwarding engine employs a forwarding table (FT) downloaded from the network processor to make the routing decision. The network processor executes the routing protocols, such as RIP and OSPF, and maintains a routing table (RT) and a dynamic forwarding table for fast updates. Once the route updates, the forwarding table in the network processor will be renewed. Consequently, the forwarding engine will update its table based on the update information from the network processor without downloading a new forwarding table. Although routing updates may occur frequently, forwarding table update for each routing update is unnecessary. This is because routing protocols, such as RIP and OSPF, need minutes to converge, and the forwarding tables can grow a little stale and need to be updated only once every 30–60 s [14].

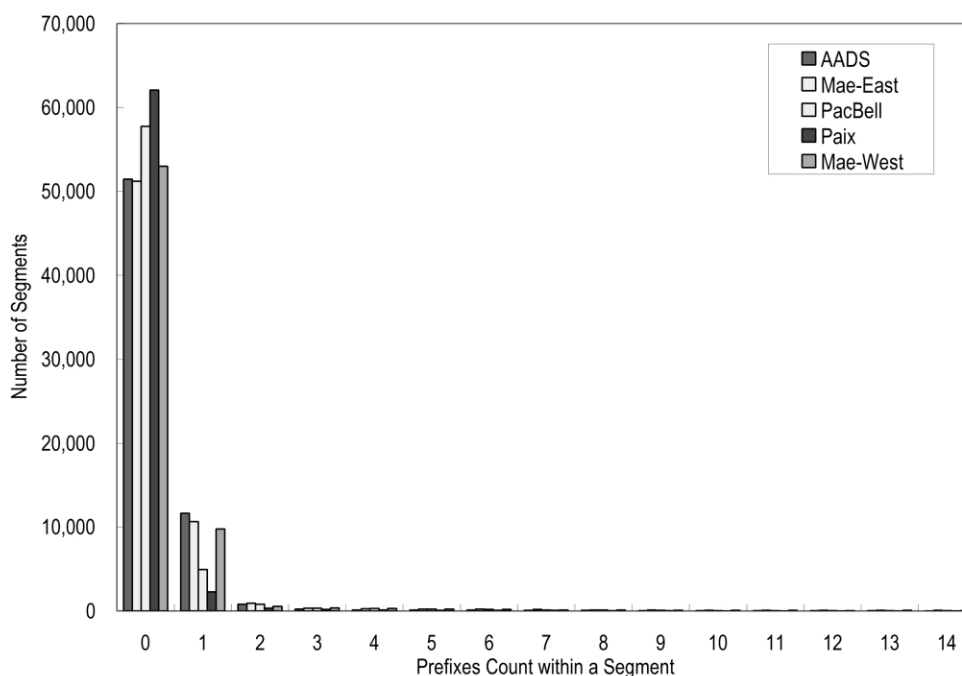


Fig. 4. Prefix count distribution.

Table 1
The sample routing prefixes

Routing prefix	Prefix length
63.192.0	20
63.192.16	20
63.192.32	19
63.192.64	19
63.192.96	20
63.192.112	20

3.1. NHA construction algorithm

Although the number of Internet hosts has increased exponentially, the routing prefix is still very sparse. For example, there are about 58,000 routing prefixes in current backbone router but 65,536 segments totally. For each segment, there is less than one routing prefix contained in a segment on average. Moreover, we found that for most segments, there is no routing prefix to define the route, as shown in Fig. 4. In fact, only few segments contain multiple routing prefixes, we call that routing locality. For the nearer area in topology, the number of routing prefixes becomes higher, and the prefixes in the associated segment become more diverse. Conversely, for the farther region, there are less and orderly routing prefixes within a segment. Therefore, it is possible to arrange prefixes for most segments and reduce the required memory.

By observing the routing information in Table 1, we can find that all routing prefixes belong to the same segment ⟨63.192⟩. In Huang's algorithm, the total required entries in an NHA would be $2^{(20-16)} = 2^4$. If we further analyze these prefixes, we can find that the first 17 bits ⟨00111111110000000⟩ of these prefixes are the same. Since the longest prefix length is 20, this means that we only have to record the 3-bits variation by building the NHA with $2^{(20-17)} = 2^3$ – entries for these six prefixes. The longer the common part is, the more the NHA entries can be reduced. The only one problem is how to carry the extra prefix information. To deal with this issue, we can add few prefix bits and its length to the entry of the segment table. This will increase the size of the segment table as trade off. The resulting NHA is shown in Fig. 5. Those entries with no routing prefix information, such as P_3 and P_5, should be filled with a default route.

The formal algorithm to construct the NHA for a segment is given below. Let l_i and h_i be the length and output port

Routing Prefix	Prefix Length
63.192.0	20
63.192.16	20
63.192.32	19
63.192.64	19
63.192.96	20
63.192. 112	20

Fig. 5. The resulting NHA from the sample routing prefixes.

identifier of a routing prefix p_i, respectively. The *cprefix* represents the common part of routing prefixes beyond the first 16 bits in a segment, and *clength* is the length of *cprefix*. *Mlength* is equal to the longest prefix length in the segment minus 16, i.e. $clength \leq Mlength$. Let $p_i(x,y)$ represent the bit pattern of p_i from the xth bit to the yth bit. If the input prefix length is equal to *Mlength*, only one entry would be updated. Otherwise, all entries with the prefix equal to the bit pattern $p_i(clength + 16, l_i)$ will be updated with its output port identifier h_i. Thus, the range of updated entries is from $V(p_i(clength + 16, l_i)2^{Mlength-l_i}$ to $V(p_i(clength + 16, l_i))2^{Mlength-l_i} + (2^{Mlength-l_i} - 1)$, where $V(p_i(a,b))$ pattern $p_i(a,b)$. Besides, most routers have a default route with zero prefix length which matches all addresses. The default route is used consequently if no other prefix matches. Thus, the table is initially assigned the default route for possible reference to these entries.

3.1.1. NHA-construction algorithm

Input: The set of routing prefixes of a segment.
Output: The corresponding NHA of this segment.
Step 1. Let l_i and h_i be the length and output port identifier of a routing prefix p_i, respectively.
Step 2. Let $P = \{p_0, p_1, \ldots, p_{n-1}\}$ be the set of sorted prefixes of an input segment. For any pair of prefixes p_i and p_j in the set, $i < j$ if and only if $l_i < l_j$.
Step 3. Assign $cprefix = p_0(17, l_0)$, $clength = l_0$ and $Mlength = l_{n-1} - 16$.
Step 4. For i = 0 to n − 1 do
  cprefix = common bits between the $p_i(17,l_i)$ and cprefix.
  clength = the length of cprefix.
Step 5. Construct the NHA with size of $2^{Mlength-clength}$ entries, and set initial value to default route.
Step 6. For i = 0 to n − 1 do
Calculate the range of updated entries and assign h_i.
Step 7. Stop.

Let us use an example to show how this algorithm works. Consider the set of sorted prefixes in Fig. 6. In Step 4, all prefixes are examined once for deciding the *cprefix* and *clength* values, and it results in 0 for *cprefix* and 1 for *clength*. Since the *Mlength* is equal to 8, the constructed NHA is with $2^{8-1} = 128$ entries. After constructing the NHA, the table is assigned with default route as the initial value because the routing prefixes cannot cover every entry in NHA. Then the first prefix ⟨24.48.8/22/10⟩ will be fetched. The 8th ($= 2 \times 2^{(8-6)}$) to 11th ($= 2 \times 2^{(8-6)} + 2^{(8-6)} - 1$) entries are the associated entries for ⟨24.48.8/22/10⟩ and will be overwritten with the output port identifier 10. The process is repeated for other prefixes. When processing the prefix ⟨24.48.9/24/7⟩, one can find that the value of the 9th entry is 10, which is defined by prefix ⟨24.48.8/22/10⟩. But the prefix ⟨24.48.9/24/7⟩ is a longer

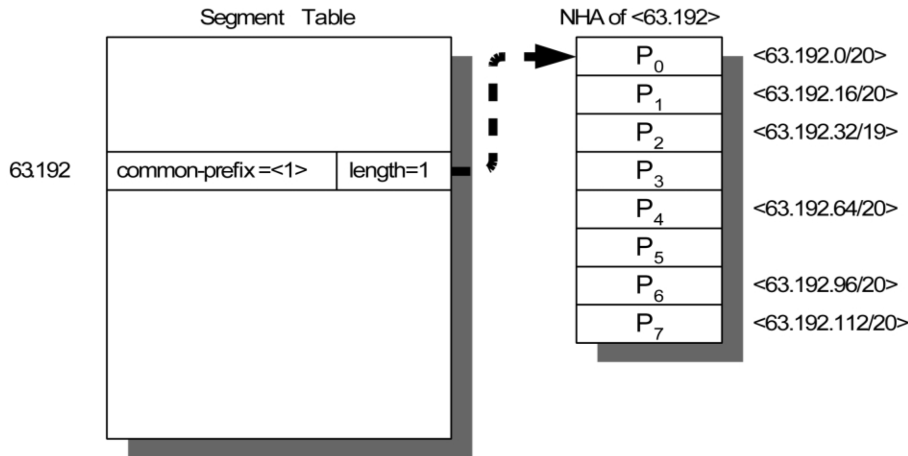


Fig. 6. NHA construction example.

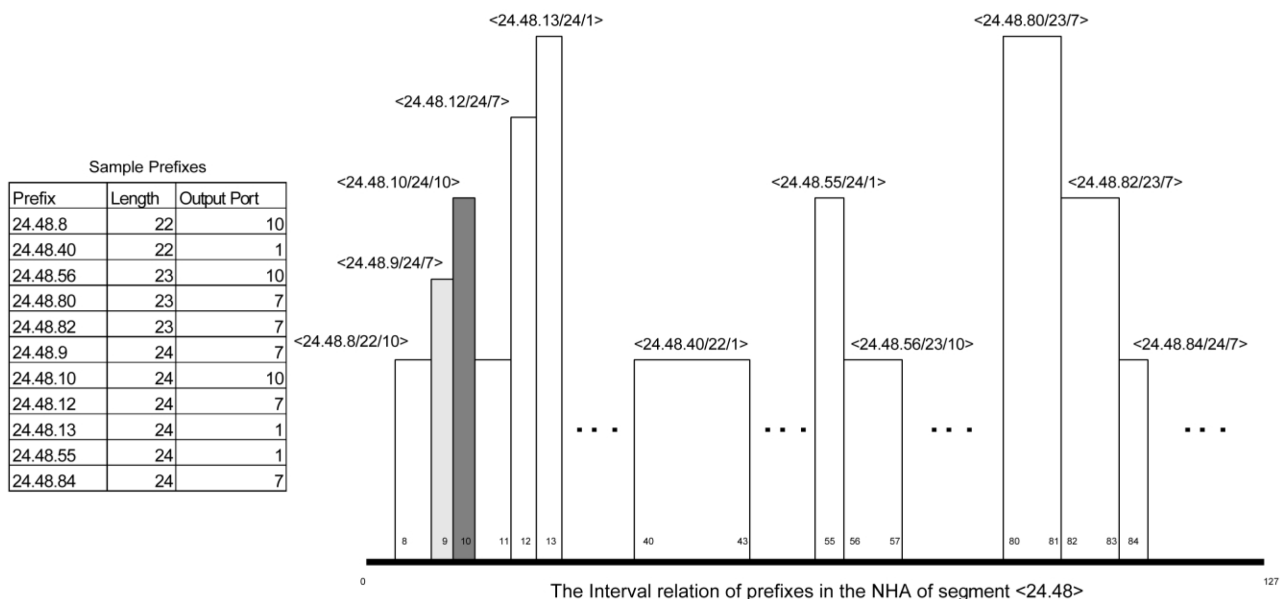
one. To satisfy the longest prefix matching, the 9th entry will be overwritten with 7 again. Since the prefixes have been sorted by their lengths, this process can be done trivially.

Obviously, the computation cost is low. If a new routing prefix is received, it will recalculate *Mlength*, *cprefix*, *clength*, which is the same as Step 4, and rebuilds the NHA if its size is changed. By applying the proposed algorithm, the entries of the generated NHA can be 25% less than Huang's algorithm. The detailed organization of the segment table and the hardware architecture will be provided in Section 3.4. The effect of the *cprefix* is also demonstrated in Section 4.

3.2. NHA compression algorithm

In the proposed NHA construction algorithm, the size of the generated forwarding table would be around 900 Kb with 3 bits *cprefix* and one byte per entry. Moreover, the NHA can be further compressed based on the distribution of output port count within an NHA. In Fig. 7, we show the distribution which is generated from traces of several main NAPs. We can find that approximately 54% segments consist of less than two output ports, where only one bit is needed to identify, and 96% segments can be encoded with less than two bits. And also, the maximum number of output ports in a segment is 20. Thus, we can encode the output port identifiers within an NHA to reduce its size.

If we assume that for each segment there are m possible output ports at the most (including the default route if necessary), then we can use b bits, where $2^{b-1} < m - 1 \leq 2^b$, to encode the output port identifiers. In addition, we need to establish another index table which records the physical output port addresses, as shown in Fig. 8. An extra field,



Sample Prefixes

Prefix	Length	Output Port
24.48.8	22	10
24.48.40	22	1
24.48.56	23	10
24.48.80	23	7
24.48.82	23	7
24.48.9	24	7
24.48.10	24	10
24.48.12	24	7
24.48.13	24	1
24.48.55	24	1
24.48.84	24	7

Fig. 7. The distribution of Output Port Count within a segment.

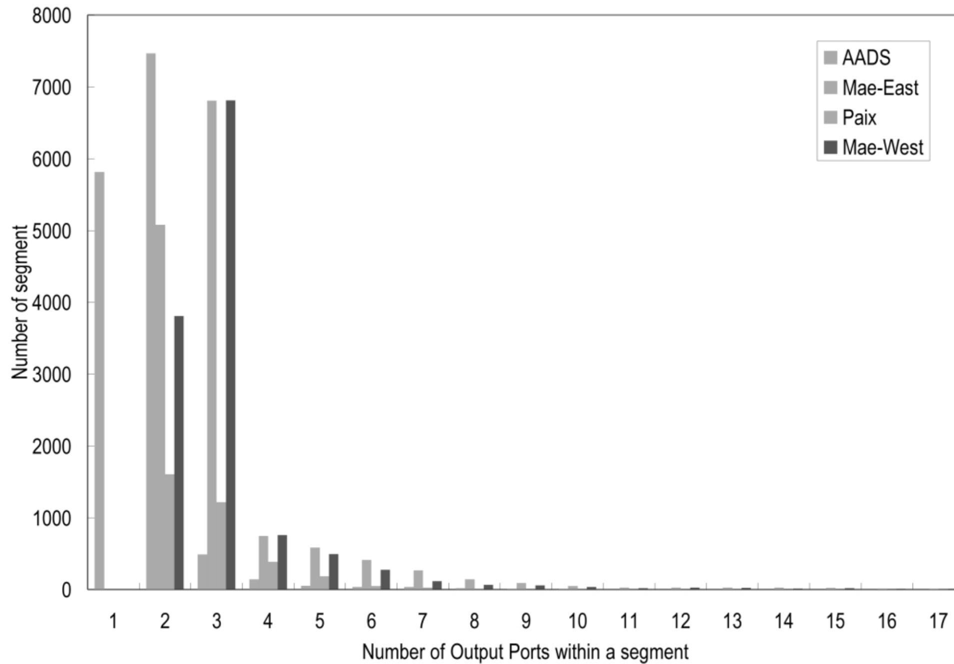


Fig. 8. An NHA compression example.

cbits with *b* bits is appended to the entry of the segment table to indicate the encoded bits. But for segments with only one output port, we can compress its NHA into one entry since the whole NHA is with only one value. In this case, we don't really have to create the NHA, instead, we assign the output port value in the pointer/next_hop field and the rest of the field is the same. In the previous trace, the entries of most NHAs can be compressed to less than 2 bits, and the longest entry in this trace is 5 bits. The index table can be appended to the tail of NHA and be searched based on the address of NHA plus its size. To update the forwarding table, we can add an entry in the index table and some necessary entries in the NHA rather than updating the whole table.

The algorithm for compressed NHA construction is shown in below. There are two extra steps in the NHA construction algorithm. The first one is to find out the possible output ports for the prefixes in a segment. However, to avoid unnecessary searching process, we use a simple vector to record the output ports temporarily. Each output port is mapped to one bit in the bit vector, and the vector is initially filled with 0. When the prefix with output port *k* is examined, the *k*th bit of the vector will be marked as 1. After examining all prefixes, the sorted index table can be built directly from the vector. Consequently, the *cbits* is calculated and then the compressed NHA space is allocated. The second extra step is to map the original output port to the index table. Since the index table has been sorted, the time complexity for mapping is $O(\log m)$, where m is the number of entries of the index table. Obviously, m is always less than n(number of prefixes), thus the time complexity is $O(n \log n)$.

3.2.1. Compressed-NHA construction algorithm

Input: The set of sorted routing prefixes of a segment.
Output: The corresponding compressed NHA of this segment.
Step 1. For each prefix, calculate cprefix, Mlength and clength. And also, records the output port value in the output port vector.
Step 2. Construct the index table from the output port vector and calculate the cbits value according to the output port count.
Step 3. Construct the NHA with size $2^{Mlength-clength} \times$ *cbits.*
Step 4. For each routing prefix
Search the index table for index value of output port.
Calculate the range of updated entries in NHA and fill with index value.
Step 5. Stop.

The biggest advantage of the proposed compression scheme is the avoidance of bit-wise operation. As described in Ref. [9], the lookup algorithm has to perform bit-by-bit test to CWA for 1s, which will cost 32 clock cycles (with 32 bits segmented-bitmap) in the worst case. With the proposed algorithm, this operation will spend only one clock cycle.

3.3. Hardware architecture

A feasible high-level hardware architecture of the proposed lookup scheme is shown in Fig. 9. The entry of

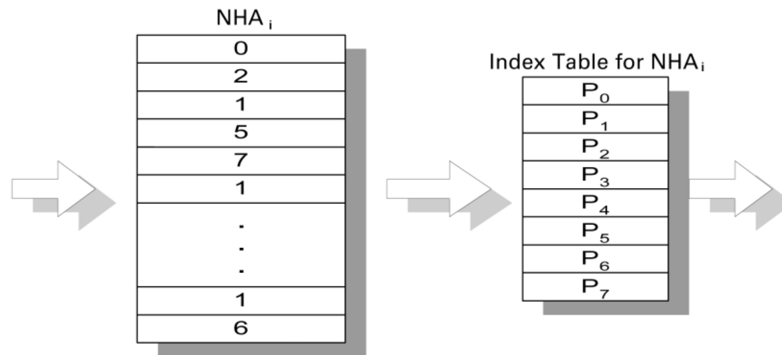


Fig. 9. Hardware architecture of the proposed lookup scheme.

the segment table consists of 5 parts: pointer/next hop, *cprefix*, *clength*, *Mlength* and *cbits*. The length of pointer/next hop is 20 bits which can map up to 1 Mega memory addresses, this is more than five times the space required for storing the whole NHAs generated from the real world routing tables. Since the maximum prefix length minus the length of segment (16) is smaller than 16, the length of *Mlength* is 4 bits. Then we consider the left three fields. The maximum number of output ports from a segment is 20, it needs 5 bits to represent the addresses. Thus the bit count of *cbits* is 3. If we use *cprefix* with length 3, then the *clength* is 2 bits. Therefore, the length of each entry is 32 bits, and the size of the segment table is $2^{16} \times 32$ bits, i.e. 256 Kb.

When a DA is fetched, its first 16 bits are used as an index to the segment table. The value of pointer/next_hop field in the corresponding entry records the next hop if it is smaller than 256. The value will be forwarded to the selector directly. Otherwise, it records the starting address of the associated NHA. Then the bit pattern DA(*clength*, *Mlength*) is used to compare with the *cprefix* stored in this entry. If two values are identical, then $V(\mathrm{DA}(clength, Mlength))$ plus the value of the pointer is used to access the NHA. Or the default output port will be forwarded to the selector. The value in the NHA is the index to the associated index table, which is appended to the NHA. Thus, the physical output port value can be fetched by adding the starting address of NHA with its size and the index value.

An illustrative example is presented to demonstrate how the forwarding table is constructed according to the proposed architecture, as shown in Fig. 10. These prefixes belong to segments ⟨24.40⟩, ⟨24.48⟩ and ⟨24.112⟩, respectively. For the segment ⟨24.40⟩, the *cprefix* is 00 and the *clength* is 2. Thus, the resulting NHA consists of only two-entries. Notice that default route is not necessary in this case, because these two entries are covered by the given prefixes. Therefore, there are only two values in the index table. But for the segment ⟨24.48⟩, not all entries are assigned with a dedicated output port, which can be

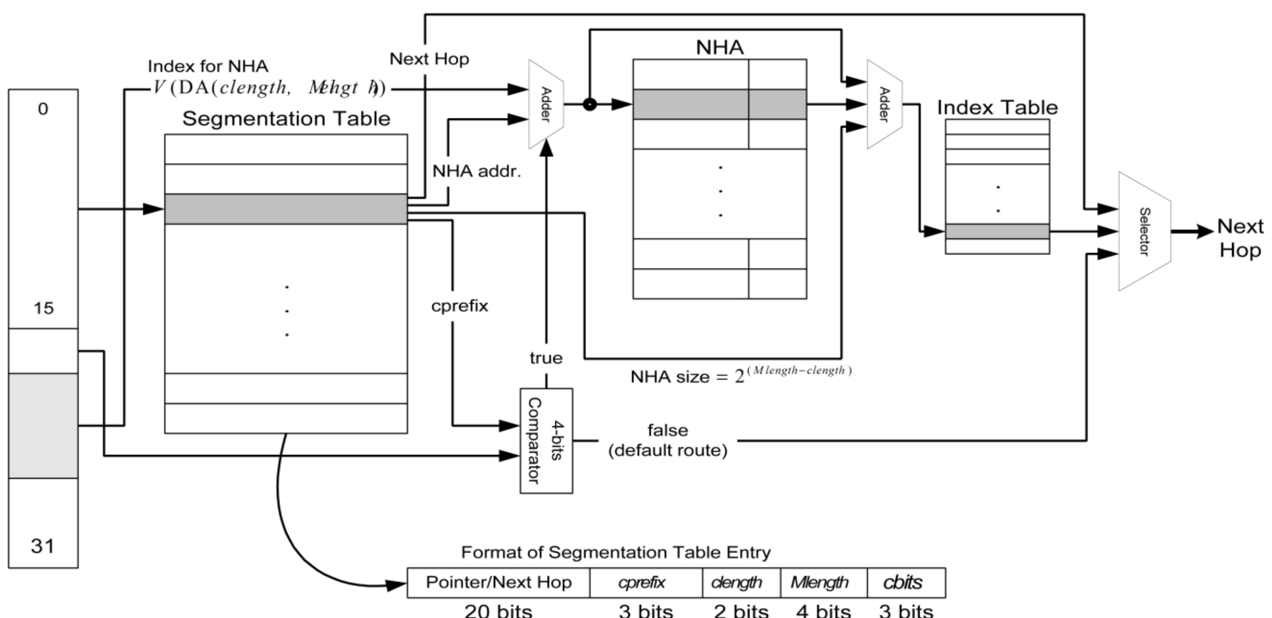


Fig. 10. Forwarding-table example.

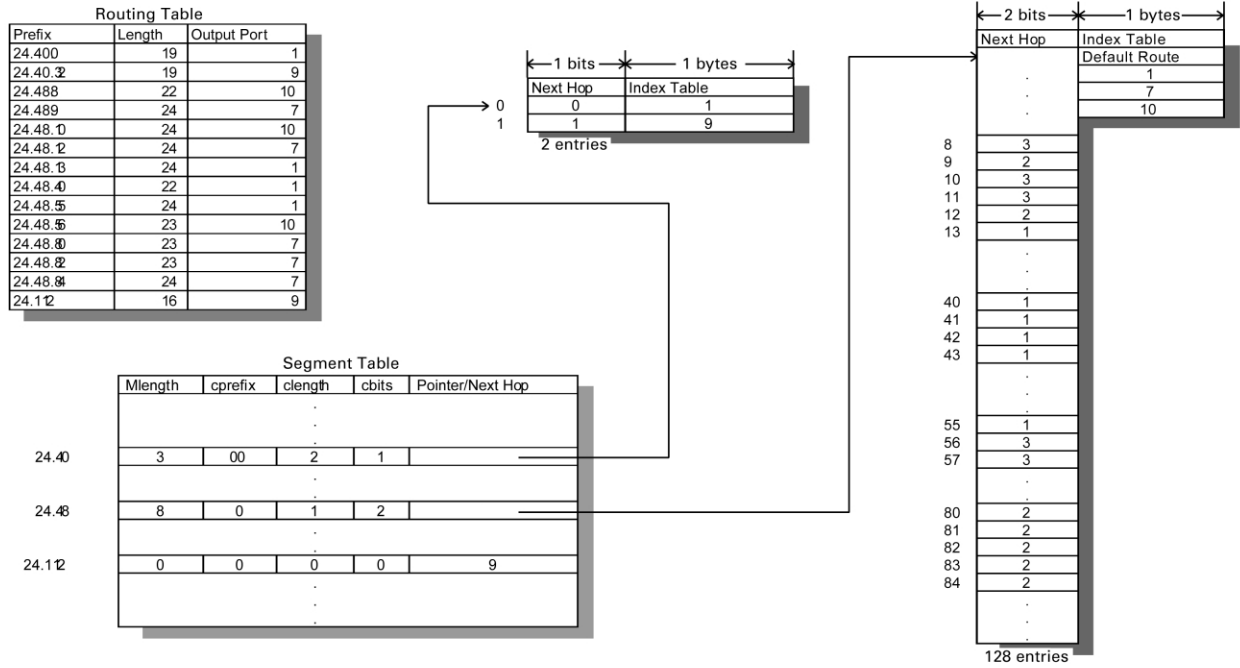


Fig. 11. The pipelining IP lookup diagram.

observed from Fig. 6. Thus the default route is added to the index table.

3.4. Pipelining hardware architecture

As described above, at most three memory access time is needed to accomplish an IP-address lookup. If we can implement this architecture in pipelining hardware, the forwarding throughput can be improved to one routing lookup per memory access. However, due to the structural hazard caused by compression scheme, this cannot be done trivially. Therefore, we propose a simple hardware implementation for the forwarding engine in this subsection.

Since we append the index table to the tail of NHA, it has to perform another memory access for output port value on the same memory space, which is the same as Huang's algorithm. There are two solutions to solve this problem. The first one is using the dual port memory to implement the NHA storage with high cost. Another solution is to put the index table in additional memory. But the address of index table is not recorded in the segment table entry. Therefore, the main issue here is how to address the index table without extra cost.

The most efficient way to address the index table is to utilize the first n bits of NHA starting address. From the given routing table, we find that there are about 10,000 index table entries at most. Therefore, 14 bits is enough to address the index table. But for each NHA, the minimum block is $2^{20-14} = 64$ b since the first 14 bits of NHA address must be different. Obviously, this is storage-inefficiency. On the other hand, if we use more bits for addressing, the utilization of index table memory might be low, but it increases the NHA memory utilization. Since the required NHA space is much larger than the index table space, we use 16 bits as index table address (i.e. 16 b per block) with 64 Kb memory to store the index table. The detailed NHA addressing algorithm is shown as below:

3.4.1. Compressed-NHA allocation algorithm

SA_i: *The Starting Address of* NHA_i;
EA_i: *The Ending Address of* NHA_i;
SA'_i: *The Starting Address of index table i;*
EA'_i: *The Ending Address of index table i;*

For each compressed NHA_i

Step 1. Fill the NHA from the SA_i with associated output port value;
Step 2. Fill the index table from SA'_i, where $SA'_i = SA_i/16$;
Step 3. $EA_i = SA_i +$ the size of NHA_i;
Step 4. $EA'_i = SA'_i +$ the size of index table i;
Step 5. $\mathrm{SA}_{i+1} = \max(EA_i + 1, (EA'_i + 1) \times 16)$;

To implement the pipelining hardware, we can use three SRAMs with 256, 256 and 64 Kb, respectively. Because the three memory accesses are performed on three different SRAMs, thus the IP lookups can be executed concurrently, as shown in Fig. 11.

4. Performance analysis

Our proposed IP address lookup scheme aims at two

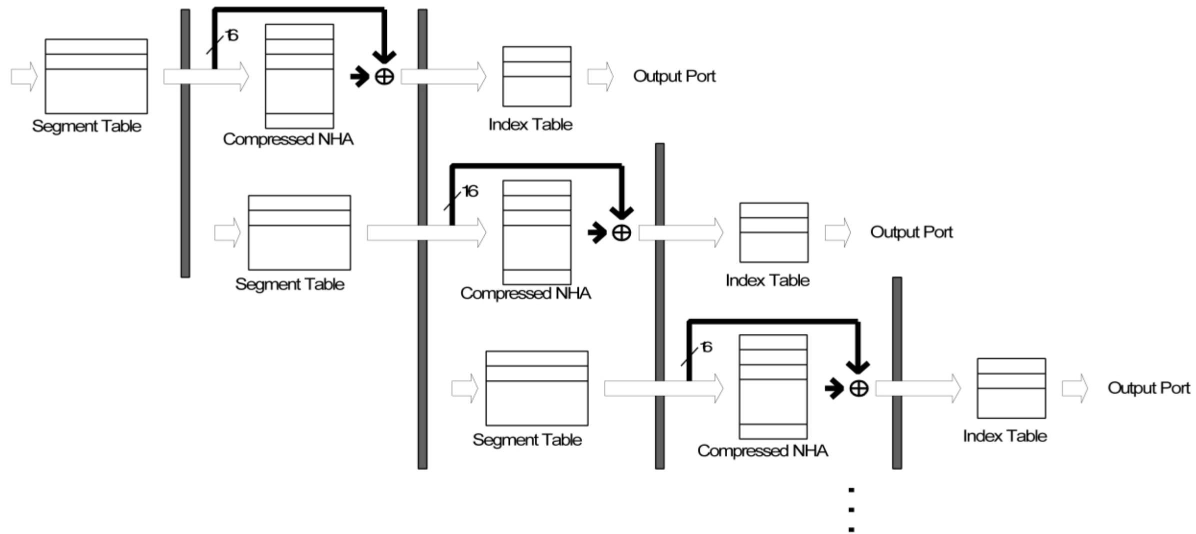


Fig. 12. Effect of the length of *cprefix*.

targets, one is to reduce the memory space needed for the forwarding tables, and the other to speed up the IP lookup process. Through simulation, we show that the proposed scheme features a low memory requirement while achieving a very high IP lookup performance. Note that the current backbone routers have a routing table with about 58,000 entries [10]. We use the logs of publicly available routing table as a basis for comparison. These tables are offered by the IPMA project [4] (7/26/2000), they provide a daily snap shot of the routing tables used by some major Network Access Points (NAPs).

To further realize the effect of the extra *cprefix*, we use the trace available in the router of Mae-East NAP to present the relation between the total number of entries in NHAs and the bits of *cprefix*, as shown in Fig. 12. Obviously, when the length of *cprefix* reduces to 0, the number of NHA entries using the proposed scheme is less than that of Huang's algorithm, this is due to the compression of 1-output port NHAs. Moreover, by coupling the effect of *cprefix*, the number of entries can be 28% less than that needed in Huang's algorithm. Moreover, the total size of forwarding table can be as small as 436 Kb, which is 152 Kb less than that in Huang's algorithm. Although as the length of *cprefix* increases, the number of NHA entries is reduced; it has to increase the entry length of the segment table, this results in larger segment table. From the observation of Fig. 12, the suitable *cprefix* length should be no more than 3 by considering both segment table size and number of NHA entries.

In Table 2, we log five traces to build the forwarding table for illustrating the effect of the proposed scheme. From the numerical result, we can find that the proposed scheme can reduce the required memory space effectively. In the most significant case (PacBell), the proposed scheme can save the required storage as large as 170 Kb. Notice that the more entries there are, the more memory reduction can be achieved.

We compare the proposed scheme with existing algorithms in Table 3 consequently. We scaled all results of previous works with software implementation to 300 MHz CPU to ease the comparison. The worst-case lookup time of LC tries [11] is not addressed in the literature so we fill it with average lookup time. Obviously, the proposed scheme outperforms than existing schemes in lookup speed while the size of forwarding table is smaller than most of them. Notably, the number of table entries used in the proposed scheme (58,101) is much larger than those used in the previous schemes (about 40,000).

Table 2
The comparison of memory requirements

Site	Routing entries	Memory usage of proposed scheme (Kb)	Memory usage of Huang's algorithm (Kb)
AADS	24,770	347	441
Mae-East	58,101	436	588
PacBell	32,388	390	504
Paix	13,395	306	475
Mae-West	36,943	310	398

5. Conclusions and future work

One of the major bottleneck in the Internet router design is the slow, software-based IP address lookup process. In this paper, we propose a fast IP-address lookup scheme that requires less memory space, and it's implementation is feasible with high-speed SRAM. We also provide a pipelining hardware implementation. By employing an efficient compression scheme, the size of the forwarding table can be largely reduced. In the most significant case, the size of

Table 3
Comparisons with other existing works

Schemes	Worst case lookup time (ns)	Memory required (Kb)
Patrica trie	1650	3262
Binary Search on hash tables	650	1600
Lulea scheme	409	160
Multiway Search	330	950
LC tries	500[a]	464
Multibit trie	236	640
Proposed scheme	30	436

[a] This is the average performance since the worst-case performance is not addressed.

the resulting forwarding table can be 170 Kb less than the previous scheme. Most of the address lookups can be done in one memory access, with the worst case being three. When implemented in pipelining hardware architecture, the proposed mechanism can accomplish one routing lookup in one single memory access time. With state-of-the-art SRAM technology, our scheme can provide more than 100×10^6 routing lookups per second. We have demonstrated that the proposed scheme reduces the memory size significantly and also outperforms the existing schemes in lookup speed, but the routing-table reconstruction is another issue for the backbone router since it requires frequently route updates. Our future work will focus on the fast routing-table update algorithm.

References

[1] Y. Rekhter, T. Li, An architecture for IP address allocation with CIDR, RFC 1518, September 1993.

[4] Merit Networks, Inc., Internet performance measurement and analysis (IPMA) statistics and daily reports, see http://www.merit.edu/ipma/routing_table/.

[5] S. Keshav, R. Sharma, Issues and trends in router design, IEEE Commun. Mag. 36 (5) (1998) 144–151.

[6] C. Partridge, et al., A 50-Gb/s IP router, IEEE/ACM Trans. Network. 6 (3) (1998) 237–248.

[7] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, Proc. ACM SIGCOMM'97, Cannes, France, Sept. 1997, pp. 3–14.

[8] P. Gupta, S. Lin, N. McKeown, Routing lookups in hardware at memory access speeds, Proc. IEEE INFOCOM'98, San Francisco, USA, March 1998.

[9] N.F. Huang, S.M. Zhao, J.Y. Pan, A fast IP routing lookup scheme for gigabit switch routers, Proc. IEEE INFOCOM'99, New York, USA, March 1999.

[10] P.C. Wang, C.T. Chan, Y.C. Chen, A fast IP lookup scheme for high-speed networks, IEEE ICC'2000, pp. 1140–1144.

[11] S. Nilsson, G. Karlsson, Fast address lookup for internet routers, Proc. IFIP Fourth International Conference on Broadband Communications (BC'98), 1998, pp. 11–22.

[12] M. Waldvogel, G. Vargnese, J. Turner, B. Plattner, Scalable high speed IP routing lookups, Proc. ACM SIGCOMM'97, Cannes, France, Sept. 1997, pp. 25–36.

[13] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Trans. Network. 7 (4) (1999) 324–334.

[14] C. Labovits, G.R. Malan, F. Jahanian, Internet routing instability, Proc. ACM SIGCOMM'97, France, pp. 115–126.