2005 IEEE International
Symposium on Signal Processing
and Information Technology

IP LOOK-UP WITH TIME OR MEMORY GUARANTEE AND LOW UPDATE TIME[1]

G.T. Kousiouris and D.N. Serpanos
Dept. of Electrical and Computer Engineering
University of Patras
GR-26504 Patras, Greece
{gkoysioyrh,serpanos}@ee.upatras.gr

ABSTRACT

IP look-up is one of the more important processes that take place during the transmission of an IP packet from its source to its destination. In every intermediate node a choice must be made as to which exit port the packet must be forwarded, after consulting the routing table. We propose two algorithms with different requirements. The ARPA algorithm emphasizes on memory management and update time for the routing table by using hashing and double linked lists, while the HYDRA algorithm achieves a steady, guaranteed performance in look-up time regardless of the size of the data structure by the use of an index structure, while greatly improving the update time of the expansion-compression algorithm.

1. INTRODUCTION

The IP look-up problem is the basic task that needs to be performed during the transmission of IP packets through the vast Internet structure. When a packet arrives at an Internet routing node, the destination address is extracted from the header and is compared with the contents of the routing table in order to decide to which exit port of the router it should be forwarded [3]. During this stage, there is a basic demand that needs to be met and is known as the solution to the Longest Prefix Matching (LPM) problem. In the structure of the routing table there may be two or more entries that match the incoming packet address [6,7,8]. This can happen, because the contents of the table are stored in a compressed form with the use of "don't care" bits (X bits) in order to decrease the size of the memory requirements [9].

Figure 1: Use of "don't care" (X) bits

Due to the Internet distributed architecture [3] the entries of the table have all their "don't care" bits in groups at the least significant bits of the address. This kind of storage creates the problem of possible multiple matching. The LPM problem consists of the need to finally choose the entry with the largest number of defined bits that matches the address of the packet.

A CLASS ADDRESS	10101010XXXXXXXXXXXXXXXXXXXXXXX
B CLASS ADDRESS	1010101010101010XXXXXXXXXXXXXXXX
CIDR ADDRESS (continuous C addresses)	101010101010101010101XXXXXXXXXXX
C CLASS ADDRESS	10101010101010101010101XXXXXXXX

Figure 2: Form of routing table entries

The best solution that has been proposed until now is the use of TCAM [9] memories to store the structure of the routing table in an array form. CAM memories perform the search in a parallel way, so the result occurs in one memory cycle only, providing the fastest result in the "search for match" process. TCAM is the memory that has the capability of storing not only the binary digits 0

[1] This work has been partially supported by the European Commission through the IST Network of Excellence (NoE) E-NEXT, FP6-506869.

and 1 but also the "don't care" bits, i.e. it can store the form of data that is used in routing tables. The basic problem here is the cost of hardware. Hardware solutions are expensive and static; also, because of the rapid improvement in CPU, memory speed and cost they can become obsolete relatively easily. Furthermore, with a change of protocol (e.g. from IPv4 to IPv6) these solutions are problematic. Another disadvantage is the high update time [6]. When a new entry needs to be inserted into the routing table, one must find the right point of insertion according to the priority of the entries. This priority is defined by the number and position of the X bits. The fewer these bits are, the higher the priority of the entry. Among equal numbers of X bits, the one that has them in the least significant bits is the most important. So, we must search linearly to find the entry point in the array and then move one position lower all the entries below.

Software solutions are generally much more flexible and can become parametric in order to adjust the trade-off between memory and speed that one desires. They don't perform as well as hardware solutions, but are much faster to design and implement and they have much lower cost, longer lifetime and higher adaptability. For the IP look-up problem, known fast algorithms, like binary search [4,12], don't work because of the existence of X bits and the LPM demand. The most promising method that guarantees a bounded time for match is the expansion-compression method [6,7], but it has a high memory waste and update time, due to the expansion phase. The HYDRA algorithm that we propose reduces the waste of memory and saves a great amount of time needed for creating and updating the data structure, while ensuring a maximum time limit for the IP look-up. The ARPA algorithm gives the best solution when memory management and update time are our basic concern and very satisfying look-up time when the routing table is relatively small.

2. ARPA ALGORITHM

In TCAM hardware implementations there is the problem of the static use of memory due to the inevitable use of the array structure. Furthermore, there is the update problem that was analyzed earlier which inserts a serious amount of delay in cases where updates are very frequent. In the ARPA algorithm we save the time for the second stage of the update (movement of all the subsequent entries) by using double linked lists in which the insertion time is minimal as we only need to change the pointers once the insertion point is found. For the first stage (the entry point) we use the following structure

Figure 3: ARPA data structure

We divide the structure in 33 lists depending on the number of X bits that the entries contain. The higher priority entries (the fully defined ones) are placed in the beginning. The position in the list is defined by the position of the X bits. The entries with X at the least significant bits are placed in the beginning of the lists. By using this structure, we do not have to search linearly throughout the whole number of entries but only throughout the list with the same number of X bits as the new entry for insertion.

For the search for match function we search linearly through the structure and because of its form, the first match will also be the best one. The performance of this stage obviously depends strongly on the size of the data structure.

Operation (Update)	ARPA	TCAM
Phase 1-Priority level 1 (number of X bits)	O(c1)	O(n)
Phase 2-Priority level 2 (position of X bits inside address)-Entry point	O(n)	O(n)
Phase 3-Insertion into the structure	O(c2)	O(n)

Figure 4: Comparison of update for ARPA and TCAM

The ARPA algorithm guarantees that only the necessary memory will be allocated and greatly improves update times.

3. HYDRA ALGORITHM

For the second part, we chose a methodology that guarantees a maximum time for the "search-for-match" process. The HYDRA algorithm is a combination of the expansion-compression method [6,7] and the index structure [5] that is commonly used in databases. Hashing in order to reduce memory requirements, as presented in [13] and [14], is more likely to increase the complexity and needed time for calculations, while the index solution and the dynamic memory allocation gives us an advantage in memory efficiency, so that we can use the IP addresses (or parts of them) directly as pointers. The basic data structure component is an array of 256 pointers to the next index level plus a "base" where the exit port is stored. The structure consists of 4 levels plus the final saving place for the exit port, in case there is no compression. Routing

table entries, as we saw, have all their X bits in compact numbers in the least significant bits of the address. To create the structure, we define the first level which consists of only one basic data component (the array plus the base). Then, we sort all the entries according to their priority in ascending order, in order to meet the LPM demand. Afterwards, we start expanding each entry but not in full as in the classic expansion-compression method for all the possible combinations of all the X bits, but only for the bits until we reach a multiple of 8 number of remaining X bits. We do this to save time during updating (early measurements showed that the time of insertion for a B-class entry was unacceptable, in the range of minutes, so for an A-class would be 256 times that time) but also the memory peak that would be necessary in the normal version. All the entries would first need to be expanded in full and then compressed during the final stage of the creation of the structure. In Figures 5 and 6 we show the limited expansion and insertion into the structure of an entry with 9 X bits. The expansion occurs only for the 9^{th} bit, while the last 8 bits remain unharmed and the insertion takes place only for 3 index levels and 2 entries. In the normal expansion-compression method we would have to insert $512(=2^9)$ entries and afterwards, during compression, we would end up in the same form of the structure as now.

TABLE ENTRY	111111111010101011111111XXXXXXXXX
INDEX LEVEL 0	11111111
INDEX LEVEL 1	10101010
INDEX LEVEL 2 (Expansion A)	11111111
INDEX LEVEL 2 (Expansion B)	11111110

Figure 5: Example of limited expansion

After this limited expansion, the entry has a part with fully defined groups of 8-bit and fully undefined X 8-bit groups. We take the i^{th} fully defined 8-bit group and access the position of the array of the i^{th} level of the index using as a key the numerical value that derives from these 8 bits (range 0-255). In this position we find the pointer to an array of the next level if this has been previously created or allocate memory dynamically for the creation of this path, if this pointer is null. We carry on until the defined 8-bit groups end. Then, we store immediately in the next level at the base of the data structure component the exit port.

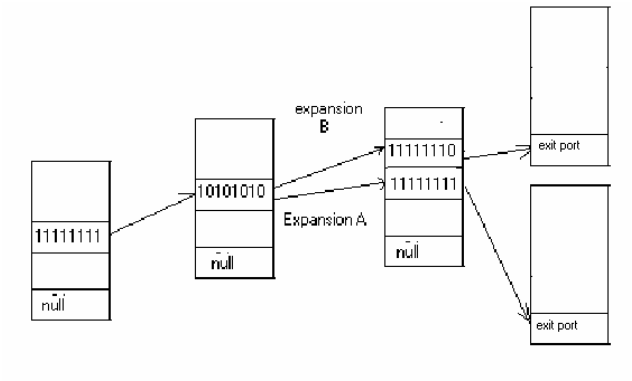

Figure 6: Insertion of limited expansion entry in structure

This means that we enter the data compressed so there is no need for further compression, another time-saving feature of this algorithm. The division of the entries in 8-bit groups was chosen due to the fact that it complies with the standard A, B, C classes of IP addressing for the expectation of better and faster compression. This division can be performed in whatever number one desires, according to the speed and memory requirements. Small groups mean more memory accesses but also more efficient use of memory and large groups the opposite. In this case, the maximum number of accesses is 5 (4 levels plus the final position of the exit port). For the look-up procedure, we simply divide the IP address in groups of 8 bits and access each level of the structure according to the numerical value of each group. If the pointer in the array of a level is null the result of the search (exit port) is at the "base" of the structure; if it is defined, then we follow the rest of the path.

The main disadvantage of this algorithm is that, in the update stage we must hold an updated file of the earlier routing table entries and perform the update again in case there are overlapping entries that arrive dynamically, to ensure that the LPM demand is met. But with the limited expansion solution this task is carried out in a very satisfying time (for 256 table entries this time is less than 2 seconds in our experiments). The memory requirement for this algorithm depends on the statistical properties of the routing table entries. In the cases that we studied it varied from 250 to 300 KB.

4. MEASUREMENTS

Due to the fact that, using C, time measurement capabilities are limited (1/100 second units), we used a very small trace file and performed the IP look-up process ("search-for-match") for a large number of times for each trace. Furthermore, this removes the hard disk access delay which would occur with a large trace file and which is not realistic because, in real time, the IP address of the

incoming packet would be stored in a buffer of the line interface. This buffer is usually a very fast memory.

For the routing table entries we used three different configurations as a WAN, a MAN and a LAN router. For the WAN router, we used 40 entries with the majority of them belonging to A and B classes. For a MAN router, we used 40 entries with the majority of them belonging to C class and CIDR. For these two cases we created the fully defined trace file from these 40 entries plus 10 miss addresses, so the hit ratio was 80%. We justified such a ratio because normally the router, as a result of the distributed architecture of the Internet, deals with packets that have destinations near its "neighborhood", so it is reasonable for the hit ratio to be that high. For each of these 50 traces the "search-for-match" function was executed 1,000,000 times and afterwards we took the arithmetic mean and the divergence.

For the LAN routing table we used 256 entries (all the possible addresses that can occur from a C-class address) and defined the trace file's hit ratio at 20%, due to the fact that it is more reasonable for one to communicate with the outside world more frequently rather than with the local addresses. The size of the trace file was 50 IP addresses, each of them executing for 1,000,000 times. The tests were carried out on a CELERON 800 MHz computer with 64 MB of RAM and the results are shown in the tables below. The simulation was also executed in a Pentium II-333 Mhz computer with 128 MB RAM. The delays were almost double.

Search for match (ARPA mean)	28.11765 μsec
Search for match (ARPA divergence)	20.41522 μsec
Search for match (HYDRA mean)	17.45098 μsec
Search for match (HYDRA divergence)	0.4951 μsec
Update (ARPA)	< 0.5 sec
Update (HYDRA)	1 sec

Table 1: LAN Router Configuration

Search for match (ARPA mean)	6.4705 μsec
Search for match (ARPA divergence)	2.7335 μsec
Search for match (HYDRA mean)	17.43 μsec
Search for match (HYDRA divergence)	0.49 μsec
Update (ARPA)	< 0.5 sec
Update (HYDRA)	< 0.5 sec

Table 2: MAN Router Configuration

Search for match (ARPA mean)	7.431 μsec
Search for match (ARPA divergence)	2.495 μsec
Search for match (HYDRA mean)	17.4313 μsec
Search for match (HYDRA divergence)	0.490581 μsec
Update (ARPA)	< 0.5 sec
Update (HYDRA)	< 0.5 sec

Table 3: WAN Router Configuration

5. CONCLUSIONS

From the results we conclude what was theoretically expected. The HYDRA algorithm gives a very steady look-up time, regardless of the size of the structure and its update time is worse than ARPA's but still at very satisfying levels. On the other hand, the ARPA algorithm has a very good update time even in large structures but its performance in the look-up deteriorates as the structure grows. However, for small structures with high update load, as in mobile users, this algorithm could be a very promising solution.

Furthermore, we must notice that the same algorithm without any change improved its performance time in half, only by running in a better computer. This illustrates the high level of flexibility of software and the advantages from the rapid improvement of CPU performance that we can exploit over hardware solutions. Hardware usually expresses the state-of-the-art solution that is used in situations where high-end performance is necessary; but, in cases where high performance is not necessary and cost and time of implementation are crucial, software can be a very competitive alternative.

6. FUTURE WORK

For the future, one interesting aspect of these algorithms, especially for HYDRA, would be to implement a part of the calculations needed for the algorithms in hardware. A very large amount of time is spent in the division of the IP address and the calculation of its numerical value from the character string in which it is stored for the needs of the measurement process. For the ARPA algorithm, due to the fact that it requires very small amount of memory we want to evaluate keeping the structure permanently in cache memories.

7. REFERENCES

[1] D. Wood, "*Data Structures, Algorithms and Performance"*, Addison-Wesley Publishing Company, 1993.

[2] C.A. Shaffer, "*A Practical Introduction to Data Structures and Algorithm Analysis"*, Prentice Hall, 1998.

[3] A.S. Tanenbaum, "*Computer Networks"*, Prentice Hall Publications, 1996.

[4] N. Wirth, "*Algorithms and Data Structures"*, Prentice-Hall (UK), London, 1986.

[5] R. Elmasri and S.B. Navathe, "*Fundamentals of Database Systems—Vol I"*, Addison-Wesley Publishing Company, 2000.

[6] P. Poulios, "*Development of IP Packet Routing System"*, M.Sc. Thesis, University of Patras, 2003.

[7] P. Crescenzi, L. Dardini and R. Grossi, "*IP Address Look-up made Fast and Simple"*, Proceedings of ESA'99 Conference, January 1999.

[8] P. Gupta, S. Lin and N. McKeown, "*Routing Look-ups in Hardware at Memory Access Speed"*, Proceedings of INFOCOM '98, 1998.

[9] V.C. Ravikumar, R. Mahapatra and L.N. Bhuyan, "*EaseCAM: An Energy and Storage Efficient TCAM-based Architecture"*, IEEE Transactions on Computers, Vol. 54, No. 5, May 2005, pp. 521-533.

[10] P. Aitken and B. Jones, "*Teach Yourself C in 21 Days"*, SAMS Publishing Corporation, 2002.

[11] B. Kernighan and D. Ritchie, "*The C Programming Language"*, 2nd Edition, Prentice Hall Software Series, 1988.

[12] N. Futamura, "*Fast String Search Algorithms"*, Workshop on Bioinformatics and Computational Biology—IEEE International Conference on High Performance Computing, 2002.

[13] H. Lim, J.-H. Seo and Y.-J. Jung, "*High Speed IP Address Lookup Architecture using Hashing"*, IEEE Communications Letters, Vol. 7, No. 10, October 2003, pp. 502-504.

[14] Y. Jung and B. Lee, "*A Parallel Multiple Hashing Architecture for IP Address Lookup"*, Proceedings of IEEE Workshop on High Performance Switching and Routing, (HPSR), 2004.