Frugal IP Lookup Based on a Parallel Search

Zoran Čiča and Aleksandra Smiljanić
School of Electrical Engineering, Belgrade University, Serbia
Email: cicasyl@etf.rs, aleksandra@etf.rs

***Abstract*—Lookup function in the IP routers has always been a topic of a great interest since it represents a potential bottleneck in improving Internet router's capacity. IP lookup stands for the search of the longest matching prefix in the lookup table for the given destination IP address. The lookup process must be fast in order to support increasing port bit-rates and the number of IP addresses. The lookup table updates must be also performed fast because they happen frequently. In this paper, we propose a new algorithm based on the parallel search implemented on the FPGA chip that finds the next hop information in the external memory. The lookup algorithm must support both the existing IPv4 protocol, as well as the future IPv6 protocol. We analyze the performance of the designed algorithm, and compare it with the existing lookup algorithms. Our proposed algorithm allows a fast search because it is parallelized within the FPGA chip. Also, it utilizes the memory more efficiently than other algorithms because it does not use the resources for the empty subtrees. The update process that the proposed algorithm performs is as fast as the search process. The proposed algorithm will be implemented and analyzed for both IPv4 and IPv6. It will be shown that it supports IPv6 effectively.**

I. Introduction

The number of hosts on the Internet is still increasing. Also the Internet traffic continuously grows. As a result of growth of the Internet population and traffic, high performance routers are being developed to be used on the Internet. High performance routers require fast IP lookups in order to avoid congestion. Also routing protocols such as OSPF, BGP, etc. often require updates of lookup tables. So, to avoid misrouting of packets and therefore their loss or increased delay, routers must perform fast updates of routing tables. The lookup processor is together with the scheduler, the most intricate part of the network processor as described in [1], [3], [4]. In [1]–[4], we implemented and assessed the performance of the scheduler design. In this paper, we propose the IP lookup processor that will easily integrate with other modules of the network processor which is based on the FPGA technology.

The fastest lookup solution is based on the ternary CAMs (Content Addressable Memory). Ternary CAM performs the search in only one cycle. It is achieved by the comparison of the given IP address with all the prefix entries in parallel, but downside is that they are expensive and, also, they are not very scalable. Other approaches are based on the lookup table with the trie structure. In this case, the lookup process consists of traversing through the trie structure in order to find the solution. The first trie structures were binary, but for faster performance multibit trie structures were introduced so the trie has less levels and therefore better worst case speed. Also, many techniques were used to improve the lookup speed such as the trie compression [5], [6], leaf pushing [7], prefix transformation, hash functions [8] etc. Those techniques usually provide faster lookup times, at the cost of slower updates.

One of the first compression techniques was the path compression. The path compression stands for the removal of one-way branch nodes of a trie since no decision is made in those nodes. In LC-tries, the level compression is used to minimize the number of the trie levels by using adaptive stride lengths and, thus, they get faster [5]. Also, redundancy in a trie can be explored and the compression could reduce the trie based on found redunandancies [9]. Leaf pushing technique is often used in multibit tries. Since a multibit trie contains only some levels of a binary trie, the levels that are not visible in the multibit trie might contain some nodes that have the next-hop information. So, it is neccesseary to push the next-hop information from those internal nodes that are not visible to their offspring nodes at the first visible level in the multibit trie. Sometime, the prefix transformation is used, and it is usually an extension of the prefix to have a specified length [10]. Also, in some algorithms, modifications of the classical trie structures can also be found [11].

In [12] basic goals and assumptions for efficient IP lookup were introduced. The main goal for a good IP lookup algorithm is that it should be fast and easily implementable. In particular, a good lookup algorithm should require minimal number of accesses to the external memory, and easy updates. A good overview of lookup algorithms is given in [13].

Our algorithm is based on a multibit trie. Such algorithms traverse through the trie using m-bit strides to decide which node in the trie is next. Lookups are faster for longer strides, but the memory requirements are higher. For example, if the stride is s=32 bits long then the lookup would be performed in one step, but 2^{32} memory locations would be needed. The multibit trie algorithms might require the excessive time to be completed since they require many accesses to the external memory. Our algorithm keeps the limited information about the trie structure in the FPGA internal memory, so that it can search the ranges of prefixes in parallel. Different, but also parallelized lookup algorithm was proposed in [14], but it was designed primarily for IPv4, and is not easily extended to support IPv6. The data structure that describes the lookup table (i.e. multibit trie) used by our algorithm is similar to the one described in [15]. But in [15], different trie levels are searched sequentially, and not in parallel, and the data defining the trie is stored in the external memory. Also in [15], the subtrees of different levels are connected via pointers

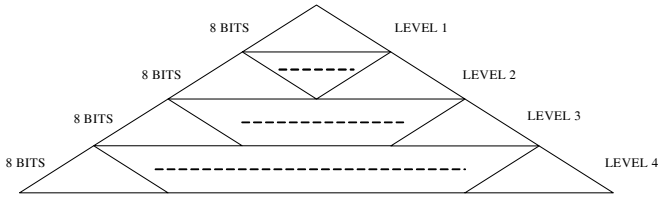

Fig. 1. Multibit trie

so that empty subtrees occupy the memory, and updates are slightly more complicated than in our algorithm. Algorithm implemented in [16] is similar to the one implemented in [15]. In [16], the subtree pointers to the next level subtrees are organized so that their access is faster. Also, the programmable FPGA chips are used for implementation of the algorithm in [16] instead of the ASIC chips used in [15]. In our algorithm, pointers are not used to determine the trie, and, no information about empty subtrees is stored which saves the memory and the registers. The output port addresses are stored in the external memory at the locations which are related to the locations of the corresponding trie entries in the FPGA memory. Reduced memory requirements allow the parallelized implementation of our algorithm on the FPGA device. Such parallelization achieves higher speeds with the minimal number of the external memories. In addition, the update procedure in our algorithm is much simpler than in other multibit trie algorithms and it is of the same simplicity as the lookup procedure itself. For these reasons, the proposed algorithm is suitable to support IPv6 as well.

The algorithm that we propose, and which we name the parallelized frugal lookup (PFL) algorithm, will be described in the next section. Its implementation will be analyzed in Section 3, and its performance will be compared with the performance of the previously proposed algorithms. The PFL algorithm will be implemented and analyzed for both IPv4 and IPv6 in Section 3. Section 4 concludes the paper.

II. Parallelized Frugal Lookup (PFL) Algorithm

The parallelized frugal lookup algorithm, PFL, that we propose is based on the multibit trie, as we have just mentioned. The multibit trie is split into the subtrees for the given stride length. Each stride reaches a new subtree as shown in Figure 1. In Figure 1, IPv4 with 32 bits long IP addresses, and the stride length of s=8 are assumed. Subtrees belong to different levels depending on the number of strides required to reach the particular subtree. The subtree prefix is the sequence of bits that leads to that subtree. The PFL processor searches a subtree at each level in a separate module. These modules operate in parallel. In this way, the speed of the multibit trie algorithm is multiplied by the number of levels. Concretely, the design is speed up L/s times where L is the IP address length and s is the stride length.

Architecture of the PFL processor is shown in Figure 2. The design consists of the central module, the level modules and the external SRAM. The level modules perform the search and update at the corresponding levels, and the central

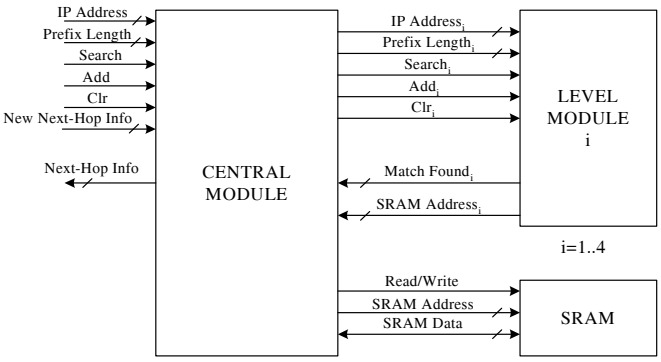

Fig. 2. The PFL processor

module selects the best solution found by the level modules. The SRAM memory holds the next-hop information for the given IP destination addresses of the incoming packets, most importantly the output ports to which these packets should be forwareded to. Using control signals *Search, Add,* and *Clr*, the PFL processor is given the instruction which procedure it should perform: search of the next-hop information for the given IP address, adding a new prefix to the lookup table and clearing the existing prefix from the lookup table. The central module defines the type of operation that the level modules should perform using these signals.

When they are commanded to search, each level module returns the signal *Match Found* that indicates if the particular level module found a match, and if so, it also returns the signal with the *SRAM address* corresponding to the longest prefix match of the given IP address. For the lookup update, the PFL processor needs the *Prefix Length* to determine which part of the IP address is the prefix, i.e. network address, that should be added to the lookup table, or deleted from the lookup table. Finally, when the central controller gets the *SRAM Address* i, $1 \leq i \leq L/s$, from the level modules for any of the cases, it determines the resulting *SRAM Address* and uses it appropriately, updates it or reads the *Next-Hop Info* from it.

A module that searches the subtree of a certain level is shown in Figure 3. It comprises registers, combinatorial logic and memory. Register k holds the prefix of the subtree whose memory address is k. Match logic compares the IP address prefix corresponding to the given level with the prefixes in the registers. It possibly finds a match between the IP address prefix corresponding to the level in question and the subtree prefix from one of the registers. The output of the match logic is the *Match Vector* which comprises zeros and one at the position that corresponds to the register that contains the prefix of the given IP address. One-hot decoder translates this *Match Vector* into the *Subtree Address* of the corresponding subtree. The subtree processor reads the *Subtree Vector* from the memory. The *Subtree Vector* comprises bits that correspond to the nodes in the subtree when they are numerated as shown in Figure 4. In Figure 4, node k corresponds to the kth bit of the *Subtree Vector*. Each node corresponds to one prefix. A bit is set to one if the corresponding prefix exists in the

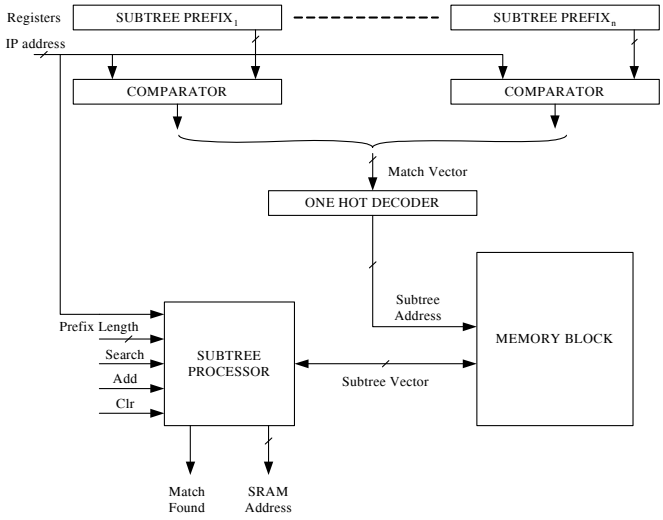

Fig. 3. Architecture of a level module

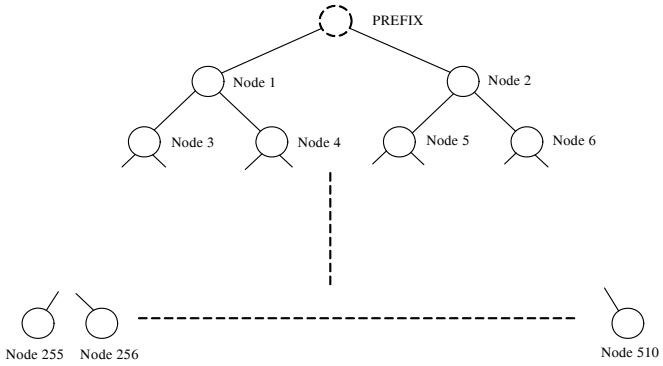

Fig. 4. Subtree structure

network. Based on the *Subtree Vector*, the subtree processor calculates the longest prefix of the IP address at the level in question, as well as the corresponding SRAM address. The number of subtrees at levels 1 and 2 are small enough (1 and 2^s respectively) to store all subtrees in the memories making the modules of these levels simpler in this way. In this case which we implemented, the modules of levels 1 and 2 have only the memory and the subtree processor. The subtree vector is read from the single memory location of level 1, or based on the first s bits from the memory of level 2.

The update process is very simple. It uses the same design blocks as the search process shown in Figure 3. When the lookup table is updated, the subtree address in the FPGA memory has to be determined, as well as the SRAM address of the next-hop information. The subtree address in the FPGA memory is calculated by the match logic based on the subtree prefix. Based on the prefix length and the IP address, the subtree processor calculates the corresponding bit of the subtree. This bit is set to 1 if a new prefix is to be added to the lookup table, or to 0 if an existing prefix is to be deleted from the subtree. The subtree processor calculates the SRAM address where the next-hop information for the new prefix will be written. When the SRAM address is calculated, the next-hop information is written to SRAM. Speed of the update process is as fast as the search process which is very important property of our approach.

The subtree prefix is stored in a register, and the subtree vector is stored in the memory of a module only if at least one prefix from the subtree exists in network. Otherwise, resources are not used for an empty subtree. In this way PFL saves significant memory resources, of both the internal FPGA memory and the external SRAM. Such a scheme is implemented by associating one counter to each register. The counters are initially set to 0. When a prefix belonging to a new subtree is to be added, one of the registers with the counter equal to 0 is chosen and the new subtree prefix is loaded to this register. Whenever a new prefix of this subtree is to be added, its counter is incremented by one. Whenever an existing prefix is deleted from the subtree, its counter is decremented by 1. When the counter of some subtree reaches 0, a new subtree can be stored in the FPGA memory and SRAM in the space that was occcupied by the subtree whose all prefixes were deleted. We use the register vector to keep the information about the free registers, in which 1 indicates that the corresponding register is free, and 0 that it is occupied. The register vector is updated whenever some counter reaches 0, or becomes positive. A selector is used to fastly determine which register to occupy by the new prefix to be added. The selector is realized as the combinatorial logic.

In both cases, search and update, access to the FPGA memory is needed for the logic to obtain or store the subtree. The subtree vector is collected in the second clock cycle after the FPGA memory address bus is set, therefore one empty clock cycle would exist in a straightforward implementation. To avoid this empty cycle, pipelining is introduced. In the empty cycle, instead of waiting for the data, a new FPGA memory address is set for the next update or the search procedure, so there is no loss of the cycles. While the *Subtree Vector* is read and processed, in that same cycle new FPGA memory address can be set. Therefore, the level modules are designed as the state machines with two states. In each state, the level module sets the FPGA memory address and processes the subtree vector which is located at the FPGA memory address set previously in the same state - two clock cycles earlier. Due to the pipelining, in each clock cycle, the level module provides a result that comprises the indication if there was a match or not, and if so, the SRAM address.

To summarize, the central module in each clock cycle gets the information from the level modules. Based on the information about matchings that have been found, it selects the SRAM address of the level module which found the match, and which processes the subtrees of the highest level compared to other modules that found the matches. When the lookup table is updated, only one level module is activated via the corresponding control signals because the level where the subtree is located is known from the prefix to be updated, which is determined based on the IP address and the prefix length. When the match is found, the appropriate action is

performed. In the case of search, the SRAM location is accessed for reading the next-hop information; or, in the case of adding a new prefix, the FPGA memory is updated and the SRAM location is accessed for writing the new next-hop information. In the case of deleting a prefix, only the FPGA memory is updated. Note that in the case of deleting the prefix which is the last in a subtree, only the register vector needs to be updated, and the corresponding FPGA memory is not accessed.

A slice of SRAM is associated to each level based on the expected number of prefixes at that level. Each subtree is allocated 2^{s+1} locations with the next-hop data which include the output port addresses. The relative address of a subtree within the SRAM slice allocated to its level, equals the subtree address in the FPGA memory of the associated level module (which also equals the index of the register with the corresponding subtree prefix in the same module). This relative address of the subtree within the memory allocated for its level is determined by the match logic in the highest level module that contains the prefix of the given IP address. This highest level determines the memory slice. Finally, the subtree processor of the highest level module finds the longest prefix for the given IP address, and it determines the exact address of the SRAM location within the memory slice dedicated to this subtree.

III. Performance Analysis

Our goal for the PFL implementation is to fit it onto the low-cost FPGAs. The PFL design was developed using Quartus II 8.1 software. Design was implemented and tested on the chip from the Altera Cyclone II and III families, for IPv4 and IPv6 addresses, respectively. The stride length was taken to be s=8. Besides the previously described design, its modification was also implemented and tested. In this modified design only the leaf nodes of the subtrees are memorized (like in the classic multibit trie algorithms), so the memory requirements, for both the FPGA memory and SRAM are 50% lower. However, downside of this modification is the slower update processing, since one update can affect more then one node. Namely, when one node is deleted or added, several leaves might need to be deleted or added as a result.

For the higher level modules, estimation of the required memory resources is needed. As shown in some papers [5], [15], for IPv4, the lengths of most prefixes fall in the range from 16 to 24, so the largest memory is needed for the level 3 module that contains the subtrees with prefixes whose lengths are from 17 to 24. We examined the implementation of the routing tables reported by IIT on their web page [17]. These routing tables contained from 32.5K entries (prefixes) to 143K entries. We also wrote a program that gave us the number of subtrees at each level for different routing tables, so that we could analyze the implementations of these routing tables on the FPGA chip. For the inspected set of the routing tables, the highest number of the level 3 subtrees was 911, and for the level 4 subtrees was 52. The memory will be allocated to different levels of the routing table according to these experimental results. The correct functioning of the PFL design was confirmed for these examples. Alternatively, the memory allocation to different levels can be made adaptive. Namely, a mechanism can be implemented to change the memory allocation whenever some slice of the memory starts to fill up, i.e. the number of prefixes in this slice exceeds a specified value.

Table I shows the performance of the design for the reported routing tables of various sizes. In particular, Table I shows the resource requirements and the maximum clock speed as the performance measures. Since the pipelining is used, results are acquired in every cycle, so the lookup completion time is $T_{MIN} = 1/f_{MAX}$, and is also given in the table. One can see that the worst case results lookup time is 34ns, which corresponds to, approximately, 29 millions lookups per second. Such fast lookup can be completed even for the shortest IP packets whose length is 64B and the speed is 10Gb/s, i.e. whose duration is 50ns. Since the update process is as fast as the search process, the lookup time isn't much affected by the lookup table updates because they are occuring much less frequently than the lookups themselves. Using the higher performance FPGAs would increase the clock speed, thus making the PFL implementation even faster. Further speeding up can be achieved by running multiple instances of the PFL algorithms in parallel. Table I gives also the required FPGA resources such as the number of logic elements, registers, and memory bits. This table also presents the number of SRAM locations, where the location width is $L_W = 8$ or 16, depending on how many bits are needed for the output port ID.

Table II shows the same results for the modified design, where only the leaf nodes are kept in subtrees. The modified design requires two times less memory than the original one, as was expected, and the clock speed improved very slightly.

The design in [16] is also implemented on the FPGA chip whose family is comparable with the family of the FPGA chip that we used. Comparing to the design in [16] which was shown to perform approximately 10 millions lookups per second for routing tables with up to 16.5K locations, our design provides two times faster speed for the routing tables on the order of magnitude larger (we tested lookup tables with up to 143K entries). The PFL performance also does not depend on the frequency of update processes as much as does the design in [16]. In [15], the results depend on the type of memory that is used, SRAM or DRAM, so the throughputs are around 80 and 20 millions of lookups per second, when SRAM and DRAM are used, respectively. Note that our design was tested with the SRAM memory, but it could work with the same speed when using the larger DRAM memory. The results in [15] were obtained for smaller routing tables with up to 41.8K entries. As we mentioned before, the memory requirements for SRAM or DRAM in [15] are larger than in our approach. Also, the design in [15] was not implemented on the FPGA chip, but on the specialized, ASIC, chip which is faster but more expensive and less flexible option for future modifications.

TABLE I
PERFORMANCE OF PFL FOR IPV4

Level 3 subtrees	Level 4 subtrees	Logic elements	Registers	FPGA memory	SRAM locations	f_{MAX}	T_{MIN}
512	128	31.7K	13.6K	463Kb	459K	38.1MHz	26ns
512	256	39K	16.8K	530Kb	525K	37.6MHz	27ns
1024	128	52.3K	22.3K	730Kb	721K	30.3MHz	33ns
1024	256	59.4K	25.5K	797Kb	787K	29.6 MHz	34ns

TABLE II
PERFORMANCE OF THE MODIFIED PFL FOR IPV4

Level 3 subtrees	Level 4 subtrees	Logic elements	Registers	FPGA memory	SRAM locations	f_{MAX}	T_{MIN}
512	128	27.3K	12.9K	236Kb	230K	38.5MHz	26ns
512	256	34K	16K	270Kb	262K	39.3MHz	25ns
1024	128	49.1K	21.6K	372Kb	361K	30.6MHz	33ns
1024	256	55.9K	24.8K	406Kb	393K	31.2 MHz	32ns

TABLE III
PERFORMANCE OF PFL FOR IPV6

Level 4 subtrees	Level 5 subtrees	Level 6 subtrees	Logic elements	Registers	FPGA memory	SRAM locations	f_{MAX}	T_{MIN}
256	64	256	52.3K	23.1K	315Kb	394K	60.6MHz	16.5ns
1024	256	512	117.6K	55K	882Kb	1115K	45.4MHz	22ns

Finally, we analyzed our design for IPv6 tables. Today, routing tables for IPv6 have small number of prefixes (up to around 1.6K), and most prefixes have length 32 or 48 [18], [19]. Other lengths are very rare, and most of them are between lengths 32 and 48. We used table from [18] that had 1.6K entries, where 94.9% were prefixes that have lengths of 32 or 48. Since IPv6 addresses are 128 bits long and the stride length is still assumed to be s =8, there are 16 levels in the PFL design for IPv6. Levels 4 and 6 have the largest number of subtrees. Level 4 has 145 subtrees with prefixes whose length is between 32 and 40 bits, and level 6 has 131 subtrees with prefixes whose length is between 48 and 56 bits. Level 5 contains 46 subtrees, and other levels have significantly less subtrees. In the PFL design, the levels 4 and 6 were allocated the resources for 256 subtrees, and the level 5 was allocated the resources for 64 subtrees, other levels received resources for 16 subtrees. Since the prefixes are longer then in IPv4 case, more logic elements are needed, and, therefore, the IPv6 processor was designed on the FPGA chip from the Cyclone III family. Table III shows the performance of the PFL design for IPv6. For the described lookup table with 1.6K entries, the design achieved 16.5ns as the maximum lookup time that corresponds to 60.6 millions of lookups per second. Chip resources can be also observed in the table. IPv6 routing tables will continue to grow, so we analyzed the design for larger routing table where the level 4 was allocated 1024 subtrees and the level 6 was allocated 512 subtrees, due to the fact that the number of 32-bit prefixes is currently twice as high as the number of 48-bit prefixes. The level 5 was allocated resources for 256 subtrees, and other levels resources for 32 subtrees. If the current trend of prefix distribution continues, the PFL design with these resources should be sufficient for at least ten times larger routing tables than the current ones. In this case, our design achieved the maximum lookup time of 22ns that corresponds to 45.37 millions of lookups per second, as shown in Table III.

By comparing Tables I and III, one can see that for similar number of subtrees, the PFL designs for IPv4 and IPv6 have the same memory requirements. This is because the PFL design does not use resources for storing the empty subtrees which can be numerous in the case of IPv6 addresses. This is very important because the FPGA memory is a scarce resource which needs to be utilized as efficiently as possible. In this way, the number of external memories is also minimized which simplifies the design.

IV. CONCLUSION

We presented a novel lookup algorithm, PFL, which can be effectively used for IPv6. It is based on a multibit trie which is split into levels based on the stride length. The parts of the lookup table belonging to different levels are searched in parallel. The information about the multibit trie structure is stored in the FPGA memory, so that it can be searched very fast because the FPGA memory blocks can be accessed in parallel. Because the FPGA memories are of a smaller size than the external memories, a special attention must be given to the memory usage. The PFL algorithm is designed so that only the information about the non-empty subtrees is stored. No information is stored about the empty subtrees. This feature decreases the required FPGA resources and the external memory size, and, consequently allows the support of the IPv6 protocol.

The PFL algorithm was implemented and analyzed for both IPv4 and IPv6. It was shown that it can be implemented on the low cost FPGAs and achieve the sufficiently fast lookups.

REFERENCES

[1] M. Petrović, A. Smiljanić, M. Blagojević, "Design of the Switching Controller for the High-Capacity Non-Blocking Internet Router," *IEEE Transactions on VLSI*, accepted for publication

[2] M. Blagojević, and A. Smiljanić, "Design of the Multicast Controller for the High-Capacity Internet Router," *IET Electronic Letters*, January 2008.

[3] M. Petrović, A. Smiljanić, "Optimization of the Scheduler for the Non-Blocking High-Capacity Router," *IEEE Communication Letters*, June 2007.

[4] M. Petrović and A. Smiljanić, "Design of the Scheduler for the High-Capacity Non-Blocking Packet Switch," *IEEE Workshop on High Performance Switching and Routing*, Poznan, Poland, June 2006.

[5] S. Nilsson and G. Karlsson "IP-Address Lookup Using LC-Tries," *IEEE JSAC*, vol. 17, no. 6, June 1999, pp. 108392.

[6] M. Degermark et al., "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM 97*, Sept. 1997, pp. 314.

[7] V. Srinivasan and G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion," *Proc. ACM Sigmetrics 98*, June 1998, pp. 111.

[8] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM97*, Cannes, France, pp. 2535.

[9] P. Crescenzi, L. Dardini, and R. Grossi, "IP Address Lookup Made Fast and Simple," *Proc. 7th Annual Euro. Symp. Algorithms*, 2001.

[10] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, June 1999.

[11] T. Kijkanjanarat, H.J. Chao, "Fast IP Routing Lookups Using a Two-trie structure," *IEEE GLOBECOM 1999*, vol.2, pp. 1570-1575.

[12] P. Gupta, S. Lin, N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE INFOCOM 1998*.

[13] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, no. 2, 2001, pp. 823.

[14] R. Rojas-Cessa, L. Ramesh, Z. Dong, L. Cai, and N. Ansari, "Parallel-Search Trie-based Scheme for Fast IP Lookup," *IEE Proceedings on Computers and Digital Techniques*, vol. 150, iss. 1, Jan. 2003, pp. 43 52.

[15] D. Pao , C. Liu, A. Wu, L. Yeung, K.S. Chan, "Efficient Hardware Architecture for Fast IP Address Lookup," *IEE Proceedings on Computers and Digital Techniques*, vol. 150, issue 1, Jan. 2003, pp. 43 52.

[16] D. Taylor, J. Lockwood, T. Sproull, J. Turner and D. Parlour, "Scalable IP Lookup for Programmable Routers," *Proc. IEEE INFOCOM 2002*, vol. 21, no. 1, June 2002, pp. 562-571.

[17] http://psp1.iit.cnr.it/~mcsoft/ast/ast_data.html

[18] http://bgp.potaroo.net

[19] http://www.caida.org