Hardware Router's Lookup Machine and its Formal Verification

David Antoš
Faculty of Informatics,
Masaryk University Brno,
Botanická 68a, Brno 602 00
Czech Republic
Email: antos@liberouter.org

Vojtěch Řehák
Faculty of Informatics,
Masaryk University Brno,
Botanická 68a, Brno 602 00
Czech Republic
Email: rehak@liberouter.org

Jan Kořenek
Faculty of Information Technology,
Brno University of Technology,
Božetěchova 2, Brno 612 66
Czech Republic
Email: korenek@liberouter.org

***Abstract*— This article describes the design of the lookup machine implemented in hardware accelerator COMBO6 for IPv6 and IPv4 packet routing. The lookup machine is a single instruction machine using Content Addressable and Static Memories and the operations are performed by Field Programmable Gate Arrays.**

The design of the lookup machine is difficult to be proven correct by conventional methods, therefore model checking as a method of formal verification was employed and this case is explained in detail. In the last part, the article sums up software support needed to make behavior of the accelerator equivalent to the host computer.

***Index Terms*— IPv6 routing, FPGA, formal verification, Liberouter.**

I. Introduction

A personal computer acting as an IP router serves to two main basic tasks. It performs routing and packet switching. By routing we mean maintaining routing tables, configuration, and packet filter setting. Packet switching means that the operating system scans packet headers, compares it with the tables, and decides how to handle the packet.

The main limiting factor of a personal computer router is the throughput of the PCI system bus and interrupt latency. A natural solution is to overcome pushing packets through the bus and to build an accelerator that performs packet switching in hardware. The *COMBO6* accelerator developed in the scope of the *Liberouter* project [1], [2] serves just for this purpose. On the opposite, routing and filtering table maintenance is left to the host computer. This kind of functionality is quite difficult to implement in hardware; it is memory consuming compared to packet switching. Routing is not a time-critical operation and does not have to be accelerated—running a routing protocol on a computer causes only a negligible load.

In any case, we must keep in mind that the host computer implements the whole router functionality, it can both route and switch packets. Purely software routers are quite popular in the academic network of the Czech Republic for their reliability, configurability, and inexpensiveness [3]. Their throughput is comparable with middle-class commercial routers. We are also convinced that todays easiest solution for IPv6 routing is to use a personal computer with a *BSD family operating system installed.

We move part of the packet switching functionality into the hardware accelerator, step by step. This allows keeping the complete functionality all the time, only increasing the overall speed of the system during the whole development process. We are free to decide even not to implement some features at all if the operations required in hardware were too complicated. Such packets would not be accelerated, nevertheless it would not be a problem if their number in the traffic was small.

This approach is known as *hardware/software co-design*. The division of the functionality between hardware and software is unusually clear in our case—from the discussion above, packet switching is an obvious candidate to move to hardware.

One of the key parts of COMBO6 packet switching is the lookup operation that finds out how to handle a packet. The input of this process is the packet header, the output is a record denoting where to send the packet and how it must be edited before sending out. Software support is needed to ensure that packet switching is equivalent to the operating system one, in other words, the accelerator must treat pack-

This research is supported by the FP5 project "6NET" (IST-2001-32603) and CESNET project "IPv6 implementation in the CESNET2 network" (02/2003). Work of Vojtěch Řehák is partially supported by GAČR grant 201/03/0509.

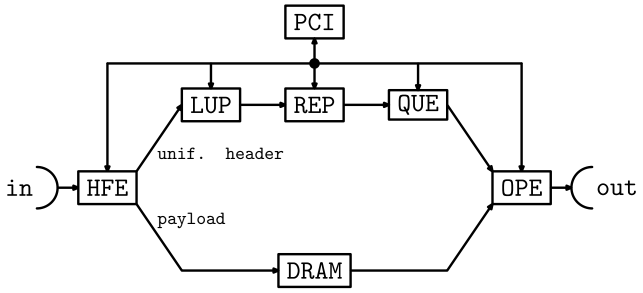

Fig. 1. Architecture of COMBO6 microcode

ets in the same way as the host computer would. Hence, software support must take operating system's knowledge about current routing and propagate it to the accelerator. Lookup operations in hardware differ substantially from software packet processing, so sophisticated conversion is needed.

This article describes proposed hardware implementation of the lookup machine and formal verification we have used to ensure that the design of the hardware unit is correct. In the other part, we briefly describe the architecture of accelerator's software support, routing/firewalling table concept, and lookup program computations.

II. Hardware implementation

COMBO6 accelerator is a PCI card mounted in a personal computer. The card contains a Field Programmable Gate Array (FPGA) [4], memories (static, dynamic, and content addressable), power sources, and other necessary logic. FPGAs are programmable, their microcode is uploaded at boot time and can even be partially changed at run time.

Microcode of the FPGA is designed as a sequence of *nanoprocessors*, small special-purpose processors. Instructions of nanoprocessors are interpreted by FPGA's microcode. Nanoprocessors perform dedicated operations with the packet.

A. Inputs and Outputs

Fig. 1 shows the block structure of COMBO6 card. Incoming packet enters a Header Field Extractor nanoprocessor. It parses headers of the packet and creates a structure called called *Unified-header*, physically kept as a set of registers. Unified-header is a 596 bit long structure containing relevant information extracted out of packet headers on fixed positions. It is intended to abstract away from the real structure of the header in order to simplify lookup operations. (Another data set denoting real structure of headers is created for output editors.)

Main fields of a Unified-header include

- L2 and L3 status registers,
- source and destination MAC addresses,
- source and destination IP addresses,
- VLAN id,
- source and destination ports,
- ...

Creating the Unified-header, the HFE stores the whole packet (headers and payload) into the dynamic memory (DRAM). When the Unified-header is finished, it is pushed to the lookup processor LUP.

COMBO6 is equipped with four input interfaces. Each of them has its own HFE and lookup machine.

LUP reads the Unified-header and computes pointers to editing programs. An editing program denotes interfaces where packet should be sent out and how packets shell be edited there. REP block replicates the packet identification to the set of output queues (QUE). Finally, output packet editor (OPE) modifies the packet before sending out of the router.

One of the output interfaces is also the operating system, card interfaces behave as ordinary network interfaces. This allows sending packets from the host computer, delivering packets to the host computer, and solving exceptions.

B. Lookup Nanoprocessor

Ideally, the whole problem of header lookup would be solved using appropriately wide Content Associative Memory (CAM)[1] containing preprocessed routing and firewalling rules. Unfortunately, no sufficiently wide CAM is available on the market. As only around 270 bits out of 596 needed can be matched in CAM, several routing/firewall rules may be mapped into a single CAM entry. Such situation can be solved by a consequent "tree lookup." This part of the lookup operation can be expressed with a set of "instructions."

From hardware point of view, we use two types of memories, namely associative and static RAM. CAM Micron MT75W16Y136HBB allows configuration for 4K words of 272 bit length. It is also capable to use "don't care" bits. Typical access time is about 80 ns. The other type of memory is an ordinary static RAM with typical access time 10 ns.

Instructions of the lookup machine are executed by the microcode of the FPGA. Instructions can be categorized into three groups:

1) `CAM Step, List` uses CAM memory. It takes a subset of registers of the Unified-header and finds them in CAM. `List` parameter denotes the subset to be matched. The value returned by CAM serves as a pointer to the next instruction in the program. The execution time depends on the speed of CAM; in the case of used memory is 120 ns.
2) Comparison instructions compare individual registers of Unified-header with constants. Tests include equality, less, less-or-equal and other relations. It also supports bit comparisons. It can use a mask to select a part of the register that should be compared. Instruction execution takes 40 ns.

   For example, `LTH Step, Reg, Const` compares the content of register `Reg` with constant `Const`.

[1] CAM takes a word of several tens or hundreds of bits as the input and answers with the address where the word is kept.

3) `EXE Queue` is the terminal instruction finishing the lookup run. It sends packet identification and its parameter to the output queue where the packet is replicated, edited, and sent out. `Queue` parameter contains information how to handle packet in the output queue. Instruction execution takes 40 ns.

All the instructions except `EXE` are conditional relative jumps. Length of the jump is denoted by the `Step` parameter. If the instruction succeeds, it jumps to address increased by `Step`, otherwise it goes to the instruction on the next memory position.

The lookup program starts for each Unified-header with `CAM` instruction, then it continues with a sequence of comparison instructions and finishes with `EXE` instruction. Having `CAM` as the first instruction is just a practical limitation. It is quite time consuming to collect a subset of registers to be fed into CAM. Doing so in the very beginning of the computation only increases latency and this operation can be pipelined. On the opposite, in the middle of the computation, such operation would cause delay.

If we understand the lookup program as a tree structure, CAM represents first levels and the remaining instructions the rest. Important property is that the levels have "distinct abilities." For example, CAM is able to specify "don't care" bits but cannot do a less-or-equal comparison efficiently. Don't care bits in CAM allow to perform best matching prefix search in this device if the entries are sorted from longer to shorter prefix lengths. If multiple entries of CAM match, the memory chooses the entry with lowest address.

C. Hardware Implementation

In the design, there are four lookup machines that share SRAM and CAM memories. The machines must access memories exclusively. Each lookup machine has its own time slot to access SRAM. CAM instruction ends with a SRAM access, therefore CAM instruction must start so that it ends in particular machine's SRAM time slot. Figure 2 shows an example of lookup machine runs. Horizontal bars divide the graph into 10 ns slots. LUP1 to LUP3 are lookup machines and the rectangles denote the activity of the machine. In *load data* phase, registers are copied into CAM. In this moment, CAM must be exclusively used by the machine. During *latency time*, CAM computes the match. Meanwhile, other data load may occur. The small *SR* slot takes the results accessing the static RAM. *COMP* phase is instruction decoding and computing. It ends with a SRAM access that fetches next instruction to process.

In the lower part of the figure possible SRAM accesses are shown. Finally, *CAM slot* is the time when lookup machine may access CAM *if and only if* CAM is not used by another machine in the beginning of the slot. In that case, the machine can start CAM operation load data.

D. CAM and SRAM Sharing Verification

The hardware design brings several important questions. Is the memory access mutually exclusive? Can a lookup

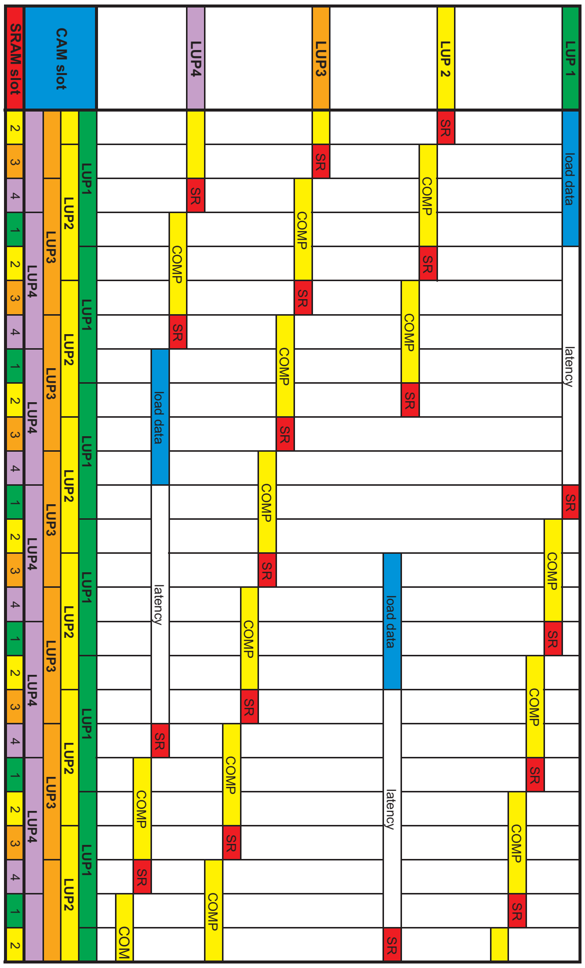

Fig. 2. Timing of CAM and SRAM sharing

machine get blocked? Is there an upper bound of waiting to get an access to CAM? We tried to solve those questions using conventional methods (i.e., ad-hoc), nevertheless this approach was rejected as too difficult.

Decision on correctness of the arranged schedule is a feasible challenge to verification section of our team. Demanded properties were verified with the symbolic model checker NuSMV. Model checking[2] is a formal method which allows one to automatically prove whether a model of a finite-state system at the suitable level of abstraction satisfies given specification.

Our model consists of five synchronous modules—four lookup processors and a `timer`. The `timer` counts time slots (each is 10 ns long in reality) modulo 4 in variable `time`. Lookup processors `lup0`, `lup1`, `lup2`, and `lup3` simulate four lookup processors sharing CAM and SRAM. Each lookup processor can be in one of six states; a processor can change state only when the variable `time` is equal to its rank. Meaning and behavior of each state is as follows:

The `sleep` state simulates behavior of a processor with empty input buffer. Subsequent state depends on whether

[2] For more information about the method see [5].

any packet comes and whether CAM is not in use by another machine. If no packet comes then the lookup processor remains in `sleep` else lookup processor changes to `wait` or `load_data` state. The choice of `wait` or `load_data` depends on whether CAM is in use by another machine or not.

The processor is in `wait` state if it wants to start processing of a packet but CAM is busy. Remaining in `wait` depends on availability of CAM. If CAM is free, the next state is `load_data`.

The `load_data` state represents the critical section of CAM sharing—loading data into CAM. The next state is `latency1`.

States `latency1` and `latency2` represent only waiting for result. In addition, `latency2` includes the finishing SRAM time slot. `Latency2` is followed by a sequence of `comp` states.

The `comp` states simulate computation using SRAM. A `comp` state corresponds to performing one instruction; in the end of the processing SRAM is accessed. The number of instructions is not limited. It is obvious that the next states are `comp`, `sleep`, `wait`, and `load_data`.

We abstract away from the emptiness of the input buffer and the number of instructions in computation using SRAM. It means that each decision based on that features is replaced by a non-deterministic choice. There are no restrictions on the choice. The described (and verified) model is therefore a bit more general than reality. It allows the input buffer to be empty forever or even to stay in `comp` states infinitely.

Good choice of the level of abstraction, invariant to the property we check, is the most powerful way to simplify the verification process. The verification gets easier, faster, or even possible.

The code below shows the real implementation of described CAM and SRAM sharing algorithm.

```
MODULE timer_type(time)
ASSIGN
  next(time) := (time+1) mod 4;

MODULE lup (me, time, CAM_busy)
VAR
  state : {sleep, wait, load_data,
        latency1, latency2, comp};
ASSIGN
  init(state) := sleep;
  next(state) :=
    case
      !(time=me)                 : state;
      state=sleep & ! CAM_busy :
                   {sleep, load_data};
      state=sleep                :
                         {sleep, wait};
      state=wait & ! CAM_busy  :
                             load_data;
      state=wait                 :  wait;
      state=load_data            :
                                latency1;
      state=latency1             :
                                latency2;
      state=latency2             :  comp;
      state=comp & ! CAM_busy  :
              {comp, sleep, load_data};
      state=comp                 :
                   {comp, sleep, wait};
    esac;
DEFINE
  SRAM_used := time=me &
        (state=latency2 | state=comp);

MODULE main
VAR
  time : 0..3;
  lup0 : lup(0,time,CAM_busy);
  lup1 : lup(1,time,CAM_busy);
  lup2 : lup(2,time,CAM_busy);
  lup3 : lup(3,time,CAM_busy);
  timer: timer_type(time);
ASSIGN
  init(time) := 0;
DEFINE
  CAM_busy := lup0.state = load_data |
              lup1.state = load_data |
              lup2.state = load_data |
              lup3.state = load_data;
```

We checked all interesting properties of this model such as mutual exclusion of access to SRAM, mutual exclusion of access to CAM, and fairness of using CAM (no starving, no blocking). Additionally we computed that the maximal length of waiting for CAM access is 120 ns. The verified formulas and results of verification are presented below.

```
-- Mutual exclusion of SRAM accesses:
-- Assert globally that no pair of lups
-- accesses SRAM at the same time
AG (!(lup0.SRAM_used & lup1.SRAM_used |
      lup0.SRAM_used & lup2.SRAM_used |
      lup0.SRAM_used & lup3.SRAM_used |
      lup1.SRAM_used & lup2.SRAM_used |
      lup1.SRAM_used & lup3.SRAM_used |
      lup2.SRAM_used & lup3.SRAM_used))
is true

-- Mutual exclusion of accesses to CAM:
-- Assert globally that no pair of lups
-- accesses CAM at the same time
AG (!(lup0.state = load_data &
    lup1.state = load_data |
      lup0.state = load_data &
    lup2.state = load_data |
      lup0.state = load_data &
    lup3.state = load_data |
      lup1.state = load_data &
    lup2.state = load_data |
```

```
      lup1.state = load_data &
    lup3.state = load_data |
      lup2.state = load_data &
    lup3.state = load_data))
is true

-- Fairness of using CAM
-- (no starving, no blocking):
-- Assert globally for each lup that
-- if the machine is in wait state then
-- it will reach load_data sometimes
G ((lup0.state = wait ->
      F lup0.state = load_data) &
   (lup1.state = wait ->
      F lup1.state = load_data) &
   (lup2.state = wait ->
      F lup2.state = load_data) &
   (lup3.state = wait ->
      F lup3.state = load_data))
is true

-- The maximal length of waiting
-- for the CAM access
-- MAX(lup0.state = wait,
   lup0.state = load_data) is 12
-- MAX(lup1.state = wait,
   lup1.state = load_data) is 12
-- MAX(lup2.state = wait,
   lup2.state = load_data) is 12
-- MAX(lup3.state = wait,
   lup3.state = load_data) is 12
```

III. Software Support

System software is developed in NetBSD and immediately ported to Linux to get wider user base. The goal of the design is to make COMBO6 independent on operating system tools used to set up and trace network traffic so as no changes to those tools are needed. Then, COMBO6 will be independent on routing daemon used in the host computer and, say, `ifconfig` may be used to set up interfaces of COMBO6.

System support can be divided into two main parts: drivers and lookup table computation.

Drivers provide standard operations like FPGA booting, accessing memories on the card (RAMs and 272 bits wide CAM words, including DMA access), status information reading, statistics collecting, and packet transfer. The driver provides a usual Un*x device (`/dev/combo`) with an application interface. Higher level lookup table computation support uses the driver to access the memories.

A. Interfaces

Let us sum how an operating system kernel treats delivered packets. The kernel maintains a routing table. It is either set up statically or maintained by a routing daemon. Packet filtering rules are described in a packet filtering language, e.g., ipfw or iptables, and kept in internal kernel structures. Processing a packet, kernel scans packet headers, consults them with the tables, and decides what to do with the packet.

Simply, we have to overcome this mechanism and push the lookup operation into the hardware lookup processor. First, we must get the configuration, routing, and filtering setting from the system. One possibility is to modify routing daemon and other tools to announce the configuration to the COMBO6 support. Main disadvantage of this approach is that we would have to modify and maintain third-party code. Moreover, COMBO6 user would be limited to a set of tools we support. Therefore, the information must be taken directly from the kernel. All changes in tables of interest are announced through so called RT-callback interface.

Firewall setting is a topic on its own. Filtering tools have an incredible amount of possibilities and mechanisms. There is no common standard in any supported operating system, to say nothing about portability. This problem is open nowadays and we study it intensively.

Traffic diagnostics is another challenge. Diagnostics tools like `tcpdump` modify kernel tables and insert filters searching for desired information there. Their settings must also be propagated into the hardware lookup machine. We suppose that in the first version, packets touched by `tcpdump` will be completely sent to software.[3] It is nevertheless desired to feed software filters with as small superset of desired packets as possible.

To sum up, the task is to take routing and firewalling table and compute the lookup structure we have described in Section II-B. A daemon watches the RT-callback interface announces and computes the hardware lookup structures. As the software and hardware way to perform lookups is quite distinct, we use a concept of routing/firewalling table.

B. Routing/firewalling table

An operating system kernel usually decides what to do with the packet in several steps. On the opposite, we have the only run of the lookup machine to handle a packet. Therefore we have to combine routing and filtering into just one lookup operation. We have created a concept of *routing/firewalling table*. Routing/firewalling table is a routing table with a-priori applied filtering rules. In general, a routing table row can be split into several routing/firewalling table entries. When any of the source tables change, routing/firewalling table must be recomputed.

This concept will have to be carefully analyzed and studied. Its expression power must be compared to standard tools for packet filtering, like ipfw or iptables.

C. Lookup structure computation

The routing/firewalling table serves as a source for lookup structure computation. The structure is computed out of

[3] Note using hardware/software co-design principles.

routing/firewalling table and is physically stored in CAM and SRAM memories.

We suppose that the structure will be strongly optimized [6]. The structure must also fit into the limited resources, mainly CAM. The optimization must take distinct abilities of parts of the search structure into account. For example, testing port numbers to be less than a constant is feasible in CAM by means of string expansion, wasting valuable CAM space significantly. Much better solution is to defer such a test to the comparison instructions.

As routing table may change, the changes must be propagated into the lookup program. For the first version we plan only recomputation of the whole structure from scratch, later parts of the structure should be modifiable. Propagating changes into hardware brings issues with timing and record validity. Two separate memories (CAM and SRAM) must be updated at the same time and the lookup structure must be switched atomically.

In the ideal case, a system monitoring network traffic would affect the optimizations according to actual conditions. In general, the optimization task can be formulated as optimization of special automaton under strong bounding conditions.

IV. Conclusion

We have described the proposed architecture of a block of a hardware design. The lookup machine is a central part of the COMBO6 hardware routing accelerator, therefore efficiency of the device depends on it crucially. Although the idea to move packet switching into hardware seems really straightforward, it leads to a non-trivial hardware design. Moreover, software support (currently in progress) is needed to propagate routing and filtering table setting into the lookup machine.

The hardware design got quite complicated and its correctness turned out to be very difficult to prove. Model checking as a method of formal verification showed to be extremely useful for this task.

This was the first case (after a long period of hunting for a common language) when cooperation with the project's formal verification group brought a practical and directly applicable result.

References

[1] Jiří Novotný, Otto Fučík, and David Antoš, "Project of IPv6 Router with FPGA Hardware Accelerator," in *Field-Programmable Logic and Applications, 13th International Conference FPL 2003*, Peter Y.K. Cheung, George A. Constantinides, and Jose T. de Sousa, Eds. September 2003, vol. 2778, pp. 964–967, Springer Verlag.

[2] Liberouter, "Liberouter Project WWW Page," `http://www.liberouter.org`, 2003.

[3] Ladislav Lhotka, "Software tools for router performance testing," Tech. Rep. 10/2001, CESNET, October 2001.

[4] Xilinx, Inc., "DS031-1 Virtex-II 1.5V Field Programable Gate Arrays," October 2001.

[5] Jiří Barnat, Tomáš Brázdil, Pavel Krčál, Vojtěch Řehák, and David Šafránek, "Model Checking in IPv6 Hardware Router Design," Tech. Rep. 8/2002, CESNET, 2002.

[6] David Antoš, "Associative Memories for IP Packet Routing, PhD Thesis Proposal," `http://www.fi.muni.cz/~xantos/pgs`, February 2003, Masaryk University Brno.

[7] David Antoš, "Overview of Data Structures in IP Lookups," Tech. Rep. 9/2002, CESNET, September 2002.

[8] David Antoš, Jan Kořenek, Kateřina Minaříková, and Vojtěch Řehák, "Packet header matching in Combo6 IPv6 router," Tech. Rep. 1/2003, CESNET, 2003.

[9] Gene Cheung and Steven McCanne, "Optimal Routing Table Design for IP Address Lookups Under Memory Constraints," in *INFOCOM (3)*, 1999, pp. 1437–1444.

[10] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi, "IP Address Lookup Made Fast and Simple," in *European Symposium on Algorithms*, 1999, pp. 65–73.

[11] Butler Lampson, Venkatachary Srinivasan, and George Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 324–334, 1999.

[12] John W. Lockwood, Naji Naufel, David E. Taylor, and Jon S. Turner, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *FPGA 2001: Ninth ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, February 2001.

[13] Anthony J. McAuley and Paul Francis, "Fast Routing Table Lookup Using CAMs," in *INFOCOM (3)*, 1993, pp. 1382–1391.

[14] Jiří Novotný, Otto Fučík, and Radomír Kokotek, "Schematics and PCB of COMBO6 card," Tech. Rep. 14/2002, CESNET, 2002.

[15] Jiří Novotný, Otto Fučík, and David Antoš, "Liberouter—New Way in IPv6 Routers," in *ICETA 2003 2nd International Conference Proceedings*. elfa, Košice, 2003, pp. 153–158, elfa, Košice.

[16] Sartaj Sahni and Kun Suk Kim, "Efficient Construction of Multibit Tries for IP Lookup," *IEEE/ACM Transactions on Networking*, June 2001.

[17] Pavel Satrapa, *IPv6*, Neokortex, 2002, In Czech.

[18] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, "Scalable High-Speed Prefix Matching," *ACM Transactions on Computer Systems*, vol. 19, no. 4, pp. 440–482, November 2001.