Packet header matching in Combo6 IPv6 router

David Antoš, Jan Kořenek, Kateřina Minaříková, Vojtěch Řehák[1]

9/1/2003

1   Abstract

The report describes current proposal and arising implementation of header matching engine in the *Combo6 IPv6 router* accelerator card.

2   Introduction

Combo6 is a PCI acceleration card for IPv6 (and IPv4) packet filtering and forwarding. This report covers *header matching* performed by the card. We divide the problem into two separate parts; the hardware processor that performs fast header matching, and the software support, preparing contents of processor's tables. From another point of view, the software's task is to compute the look-up program performed by the hardware processor.

The paper also shows the main principle we decided to follow, interconnecting the accelerator into the Un*x environment in the most standard way as possible. It means that the card should behave as usual network adapters, it only increases the performance in case of packets that can be forwarded by it. It allows incremental development and adding features in the future. It also does not force us to change standard tools for routing and network configuration, making us independent on the development of those tools.

The report is organized as follows. Section 3 describes hardware aspects of the look-up machine. Part 3.1 shows the blocks surrounding the look-up machine and part 3.2 describes the processor itself. Special interest should be given to part 3.3 where the group of formal verifications colaborating with the hardware team has shown their power. Finally, section 3.4 describes the reasons and methods why we chose to simulate the machine in software first.

Section 4 describes interconnection of the accelerator card into the operating system environment. We start describing system interfaces. Part 4.2 shows the concept of routing/firewalling table that should allow routing and packet filtering in the only look-up structure traversal. Part 4.3 discusses problems of the computation of effective version of routing/firewalling tables for the hardware look-up processor.

3   Hardware

Look-up operations (together with nearly all processes in the router) are performed by programmable hardware, *Field Programmable Gate Arrays (FPGA)*. We use two types of memories to store the look-up program. We have a Content Addressable Memory[2] (CAM from now on). We use Micron MT75W16Y136HBB. This type allows configuring as 4K words having 272 bits (it is one of the widest available CAMs). Moreover, it lets us specify "don't care" bits. Typical access time is about 80 ns. The other type of memory is an ordinary static RAM (SRAM) with typical access time about 10 ns.

3.1   Inputs and outputs

The input of the header matching engine comes in a structure called Unified-header, physically stored as a set of registers. The Unified-header is a fixed structure keeping the relevant information extracted from the packet's header. The content of Unified-header was proposed by Ivo Hažmuk. For detailed description, see the report of his group (to appear soon). For illustration, we give a short summary of main fields:

- L2 and L3 status flags

- Source and Destination MAC

- VLAN id

- Source and Destination IP address and port

- . . .

The structure comes over 440 bits in length.

There are four header extracting machines, one for each interface. Each machine has (at least) four sets of registers to store Unified-headers. Whole packets are

---

[2]CAM gets a (wide) word of bits and answers with the address, where the input data is kept in it.

stored into the main dynamic RAM, therefore the only part of the packet the header extractor must pass through is an identification of the packet (memory allocation block number). Each engine reads the Unified-header and computes a pointer to an editing program. The editing program denotes how the packet should be changed and which interfaces the packet should be sent out. Note that there may be more interfaces the packet must be replicated to. The Unified-header is destroyed after processing.

One of the output interfaces is software. This interface behaves like a normal network adapter from the operation system's point of view. This way we can

1. answer to erroneous packets,

2. receive packets sent to the host computer,

3. trace the network traffic using usual system tools,

4. process unusual packets we do not understand.

Of course, packets coming through the operating system are slowed down, therefore we plan to move some error handling to the hardware in future versions.

3.1.1 Hardware realization

Unified-header is physically stored in a buffer. The buffer is filled by the header field extractor and read by the look-up processor. The read and write operations have to be fully independent. Therefore the buffer is stored in dual port Block SelectRAM in the FPGA chip.

Each buffer consists of four sets of registers, each containing a Unified-header. The buffer is controlled by a status register; one bit for one set of registers. The bit signalizes whether the set of registers is free or contains a Unified-header. Header field extractor can write a Unified-header to the set only if the bit is zero. When the Unified-header is fully written to the buffer, the bit is set. The buffer is read by the look-up processor. It can start handling the Unified-header only if the bit is set. When it finishes the work it resets the bit. This way we guarantee that the header field extractor never rewrites any stored Unified-header and the look-up processor never processes the Unified-header that has not been fully stored.

3.2 Look–up processor

The processor performs a traversal through a tree structure (or finite automaton from other point of view), the program is started from the beginning for each incoming header. Instruction set was proposed by Pavel Zemčík.

There are three kinds of instructions:

- *CAM Step, List* instruction matches a subset of registers against CAM memory. We can choose part of the Unified-header with a bit mask *List*. The resulting address returned by CAM serves as the pointer to the continuation of the look-up program.

- Comparison instructions contain a set of instructions for comparing registers from Unified-header (or even upper and lower parts of them) to constants. The tests include equality, bigger, lower and so on.

- *EXE Queue* stops processing of the header and places the identification of the packet together with the *Queue* parameter into the output queue. The parameter is a pointer to the editing program.

Instructions of the first two sorts are relative conditional jumps. The length of the jump is driven by their parameter *Step*. For implementation reasons, we must restrict ourselves to a discrete set of possible steps, partly because of dearth of space in the binary encoding of instructions and also because of time complexity of general adder in hardware.

Instructions are encoded into 36-bit words. This is not a typo, we need to keep up to 32 bits of arguments in the instruction code, therefore we shall use parity bits of SRAM, too.

It is worth noting that *erroneous packets* are detected by the header extractor. It sets a flag in the L2 or L3 register. We must match the bit (most probably with *CAM* instruction) and mark the packet with the editing program that sends the packet to the operating system. In future versions, we plan generating ICMP messages for common errors in the editing engine. We also have a "trash" interface that only frees the dynamic memory, the memory is physically deallocated by the editing engine.

Once more, the look-up program acts as a traversal of a tree (trie) structure. *CAM* represents first levels, other instructions the rest. Important observation is that various levels of the tree have "distinct abilities," for example *CAM* is able to specify "don't care" bits but cannot do a less-or-equal comparison. The opposite is true for the rest of the tree, for other instructions.

3.2.1 Hardware level

Look-up processor is a hardware entity. Figure 1 shows supposed interfaces of the entity. The interface consists of signals connecting the look-up processor with neighboring components–buffer, CAM, SRAM, and replicator.
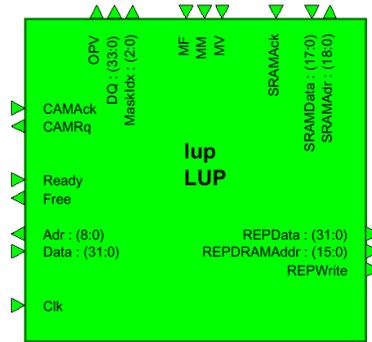
Figure 1: Look-up processor interface

Execution of an instruction starts by loading it from SRAM. Look-up processor decodes the instruction and parameters from its operating code. This step is the same for all instructions. The rest of processing differs according to the classification of instructions given above.

CAM instruction, the only instruction belonging to the first kind, is performed by the content addressable memory (CAM) connected to the FPGA chip. The look-up processor only has to load subset of registers from a buffer to CAM. Registers are specified by *List* parameter obtained from the instruction word. The parameter is organized as bit map and the look-up processor has to convert the bit map to a sequence of register addresses. The registers are loaded into CAM. After the look-up operation (consuming latency about 80 ns) CAM fetches the next instruction from SRAM.

Branching instructions compare registers to constants. This operation is done fully by the look-up processor. In the first step the parameters from the instruction word we obtained are *Address*, *Step*, *Mask*, and *Constant*. The register specified by *Address* is loaded from the buffer, masked with *Mask* and compared to the *Constant*. The type of the comparison depends on the instruction code. When the instruction succeeds we jump relatively the current address incremented by *Step*. Otherwise we go to the following instruction. It is worth noting that both the addresses are computed concurrently with the comparison and the desired address is copied to program counter.

Finally, *EXE* instruction finishes the processor's run and contains information for the packet replicator in its operation code. When processing reaches *EXE* the look-up machine copies the instruction's argument together with an identification of the packet to the output queue. Then it checks whether there is another Unified-header in the buffer and if so, it starts processing over.

3.2.2 Sharing CAM and SRAM

Let us recall that each interface has its own look-up processor. The processors share one CAM and one SRAM. Therefore it is necessary to ensure fair mutual exclusion of accesses to those resources among the processors. The mutual exclusion is an obvious must, moreover we want the accesses to be fair, protecting the interfaces from starving and blocking.

Each look-up machine has its time-slot to access SRAM. Sharing CAM gets more difficult. *CAM* instruction processing ends with accessing SRAM, so the begins of *CAM* instructions are driven by the finishing SRAM time-slot.

Figure 2: Time diagram of sharing CAM and SRAM

Figure 2 shows an example of processor's work. Small horizontal lines divide the graph into 10 ns slots. LUP1 to LUP4 are look-up processors and horizontal bars show what they do. Load data section means that registers are loaded into CAM, this time CAM cannot do anything else. CAM performs the match in the "latency time," in this phase it can be loaded another data to match. A small "SR" slot at the very end of CAM processing is getting the result of the match out of SRAM. This time-slot must get just to the SRAM time-slot for that machine. The machine gets an instruction to process, the instruction is decoded and executed during COMP phase. Following SR slots are SRAM accesses, fetching the following instruction.

At the figure bottom, there is an overview of SRAM slots with numbers of machines that are allowed to access SRAM. Finally, "CAM slot" is a place where particular machine may access CAM *if and only if* CAM is not in use by another machine. More precisely, if no machine uses CAM in the beginning of the

CAM slot, the machine owning the slot is allowed to start load data phase in the beginning of the slot. For example, when LUP1 accesses CAM in the beginning of the figure, it blocks LUP2, LUP3, and LUP4 from accessing CAM for the first four time-slots. Even if LUP4 asked for CAM at the very beginning of the picture, it would get it in the eighth time-slot (exactly as shown).

It is not obvious that this mechanism protects from blocking and provides fair scheduling. It can be shown that there is an upper bound of waiting time. Section 3.3 deeply discusses this topic.

3.3   Verification of Sharing CAM and SRAM

Decision on correctness of the arranged schedule is a feasible challenge to verification section of our team. Demanded properties were verified by the symbolic model checker NuSMV. Model checking[3] is a formal method which allows one to automatically prove whether a model of the system at the suitable level of abstraction satisfies given specification.

Our model consists of five synchronous modules, `timer` and four look-up processors (lups from now on). The `timer` counts 10 ns slots modulo 4 in variable `time`. Lups `lup0`, `lup1`, `lup2`, and `lup3` simulate four look-up processors sharing CAM and SRAM. Each lup can be in one of six states, changing only when the variable `time` is equal to its rank. Meaning and behaviour of each state is as follows:

The `sleep` state simulates behaviour of processor that has empty input buffer. Next state depends on whether any packet comes and whether CAM is not in use by another machine. If no packet comes then the look-up processor remains in `sleep` else look-up 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 loading data into CAM–the critical section of CAM sharing. 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.

---

[3]For more information see [Bar02].

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. 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 accesses to SRAM
-- specification
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
-- specification
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)
-- LTL specification
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
```

3.4 Software simulation of the look–up engine

We have decided to do a prototype of the header matching engine in C before we start coding in VHDL[4]. VHDL brings completely different approach in programming compared to usual languages. VHDL is designed to describe hardware, therefore it covers parallelism of processes and signals.

The main reason for the software simulation is that we want to test behavior of the look-up machine and related software before we fix the structure of hardware. Changes in the simulator are much cheaper and easier, moreover even a small change in the instruction set can cause a complete hardware redesign.

We will simulate real behavior of the header matching engine by passing real packet headers to its input. Collecting experience from those experiments, we will be ready to fix the structure of the hardware.

3.4.1 Look–up processor simulator

The simulator simply interprets the look-up program (described above). First, it reads contents of CAM and SRAM memories, parses them, and checks it syntactically. We decided to store the memories as text files, it is less efficient (it does not matter in the simulator), but it allows us to check (and even create) them by hand.

The simulator is especially simple, it reads the content of the Unified-header on the input and starts interpreting the look-up program. It processes the instructions until it reaches *EXE Queue*. Then it writes its value to the output queue.

3.4.2 Interfaces

The input of the simulator is generated by L2 striper and header field extractor; those blocks prepare the Unified-header out of the real packet. There are particular simulators of the blocks, too. Look-up processor receives Unified-header stored in 32 16-bit registers from those engines. We also want to test the behavior of the interfaces among the blocks. We have been working on so called "parallel simulator" that performs simultaneous run of all blocks. The

---

[4]VHDL is a high level language for hardware programming; its syntax is similar to Ada.

simulators communicate using shared memory. The parallel simulator executes four instances of header parsers and one look-up processor. (It is to be changed according to the last changes in the proposal.) The main result of the simulation is an analysis of time dependencies in the engine.

3.4.3 Future plans

We are about to finish interconnection of the header parser block and the look-up processor. Simulation is very limited at this first stage–it is because of the fact that today's header parser simulator works just at the L2 level. This means that Unified-header is not filled properly. L2 level related registers are the Source and the Destination MAC addresses. But later on–when L3 level will be ready–simulation may be very useful. We will be able to observe used instructions according to the type of packet on the input by means of instruction counters.

4   Software

Let us recall that the final goal of the software proposal is to make it independent on the tools used for routing and network setting in the operating system. Therefore we will use standard operating system interfaces as widely as possible and all exceptions must be discussed and judged well.
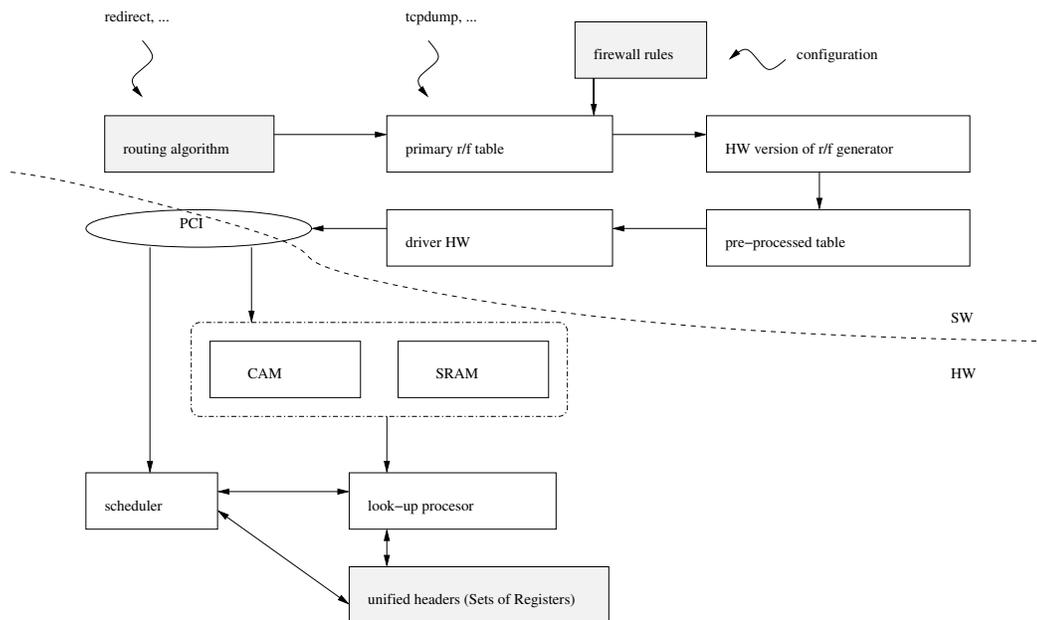
Figure 3: Block structure of header matching engine and supporting software

We describe current proposal of software support. Let us start with interfaces of the software support module.

4.1   Interfaces

In the lowest level, the module should produce programs for the look-up machine. It feeds the program to the memories through a driver. The driver exports a usual Un*x device (*/dev/combo*) and has an API on that. The API covers low level operations altering the contents of the memories. Typical operations of the API are reading/writing 36 bit words from/into SRAM, the same for blocks (for SRAM burst mode), reading/writing a line of CAM (it is 272 bits), the same for blocks of lines. The driver serves to the needs of all supporting software; it provides uploading firmware, reading status information, transferring packets, etc. This kind of communication is beyond scope of this report. Probably, the */dev/combo* device will be split into several devices, according to the structure of inner memories of the card.

Before we turn to the opposite interface, let us sum up what the operating system does to enable packet forwarding and filtering. Kernel keeps its routing table. The table is either configured statically or is maintained by a routing daemon (this will be surely our case). Firewall rules are configured using some kind of firewall description language, such as ipchains. When a packet arrives, kernel compares its headers with the tables and decides what to do with that packet.

Simply, we want to override this mechanism and let the hardware accelerator make the decision in case it knows how to treat the packet; for the most common packets first, adding other types (in accordance with a priority list) in later stages of development. Therefore we have to know the configuration of the interfaces, and current routing and firewalling tables.

There are two ways to obtain the tables. We can either modify the routing daemon (that tells the kernel what to do normally) to tell us the routing table, too. This approach has the main disadvantage that it requires modifying code out of our control, and maintaining changes in the process of development of the daemon. It also restricts the users of the *Combo6* router to the set of daemons we support. The other way (we decided to follow) is to create a daemon that listens the changes in the kernel's routing table and lets us know about them.

The separate problem arises with packet filtering. Tools for firewalling provide many different features and many mechanisms. There is no common standard in one operating system kernel, and the situation gets much worse when thinking about portability. This problem is being discussed and we plan to make a study about abilities of firewalling mechanisms compared to the possibilities and capabilities of the look-up program described above.

As we have said above, we have a daemon that watches changes in the kernel routing table and announces the differences. The hardware table computation mechanism is connected to it through so called *RT-callback interface*. The daemon itself should be divided into a system independent part and a small system

dependent layer converting the data to a format suitable for our processing. Something similar should be done with firewalling rules.

To enable network diagnostics, tools like *tcpdump* modify kernel tables to insert filters that check for the desired information. The modification of this kind must be propagated to the RT-callback interface. We suppose that packets that should be checked by *tcpdump* will be routed by software. In order not to degrade the performance too much, we nevertheless should use operating system to treat as small superset of the affected packets as possible. This is left to thorough discussion in future versions.

4.2   Routing/firewalling table

The RT-callback interface announces the changes of routing and firewalling tables together with the settings of network diagnostics tools. Today we have a very first draft of the interface suggested by Zdeněk Salvet. We shall probably keep our internal copies of the tables for efficiency reasons.

As we have the only machine for routing and firewalling in hardware, we must combine the information from the tables and we must perform the decision where to send the packet and whether to send it in the only look-up operation.

We created a concept of *routing/firewalling* table (r/f table from now on). The r/f table is the routing table with firewalling rules applied on each of its lines. In general, a routing rule may be spread into a number of r/f table entries. This structure must be analysed well, especially compared to the expression power of other "firewall describing" languages like ipchains.

When a change of input tables occurs, in the first development iteration we plan only recomputing the whole r/f table. Later we want to study how to make local changes only. Nevertheless, as we will see later, recomputation on this stage is not the critical part of the system.

Note that the r/f table also provides a connection to the editing program[5]. We expect the set of editing programs to be fixed, therefore the actions coming from the tables together with type of the packet denote the editing program.

It is worth saying that the r/f table is hardware independent. We also want to keep it as system independent as possible, but we may sacrifice some small part of this quality for efficiency reasons.

---

[5]Editing program describes what should be done with the packet, where to send it and how to change it.

4.3   Look–up program computation

Look-up program is physically stored as the contents of CAM and SRAM. It is computed out of the r/f table. The reader should keep in mind that the hardware form of the look-up structure does not have to allow reconstruction of the original r/f table for various reasons, e.g., it may contain expanded strings. We suppose that the hardware look-up program will be strongly optimised.

In the first development iteration, we plan that the update of CAM and SRAM is possible for the whole structure at once. The generator takes the whole r/f table, "thinks up" the optimization, and computes the look-up program in terms of the instructions described above. Then it asks the driver to upload the table into the memories. An easy way to update CAM is to divide it into halves, one is used for production work, the other serves for uploading new strings. When the upload is finished, the parts are atomically switched. It reduces the problem of updates only to timing of the CAM accesses. Note that SRAM is "magically" switched too, the uploadable part is not accessible from the production part of CAM and vice versa. The software driver must also keep an allocation information about the memories.

For future versions we want to enable updating only parts of the structure. Then, the look-up program generator would get an instruction "changed from-to, upload it." To demonstrate problems of this approach, we show the following. CAM, when more rows match the query, returns the very first matching row[6]. It means that more special cases must be located closer to the address zero in CAM. How do we add a more special rule, when CAM happens to be full in the location we have to use?

Overwriting of data brings also problems with validity of records. A change of look-up program in the memories must be atomic in the sense that we cannot use new data for production work until it is fully uploaded. (It is better to use obsolete routing table for a second or two than to make a complete mess.)

We suppose that a good response time for propagating changes through the whole structure should be in order of seconds. The time complexity will probably depend on the optimization technique used. In fact, the problem could be formulated as minimization of a special finite automaton with strong bounding conditions. This problem is one of the (several tens of) biggest challenges in the whole project.

---

[6]It is possible not only because of duplicated strings, but due to don't care bits.

5 Conclusion

As the reader surely noticed, "level of abstraction" increases with the chapter number in this report. This tracks the state of the project well.

We have created an implementation plan. Now (December 2002) we have a simulation of the look-up processor. Hardware part is specified in blocks and the blocks are being refined. Currently, the most of the effort is routed to the specification of interfaces. Then we plan to take care of r/f table and look-up program computation. In this state it can be tested on the card prototype where the headers are completely treated by the software simulator. Concurrently, hardware programming will take place.

The stages described above run more or less concurrently. The main point is that we want to hold up the hardware programming until we are really sure about the detailed structure of the instructions.

We have described proposed structure of the *Combo6* IPv6 card. We covered both hardware and software aspects of the design. We have shown the power of model-checking methods to make hardware design easier. We have found common language with the group of formal verification, reaching the desired point where theory meets practice.

References

[Ant02]   Antoš D.: *Overview of Data Structures in IP Lookups*
           CESNET technical report 9/2002

[Bar02]   Barnat J., Brázdil T., Krčál P., Řehák V., and Šafránek D.: *Model checking in IPv6 Hardware Router Design*
           CESNET technical report 8/2002

[Nov02]   Novotný J., Fučík O., Kokotek R.: *Schematics and PCB of COMBO6 card*
           CESNET technical report 14/2002

[WWW]   WWW pages `http://www.openrouter.net/`