A Structural Architecture for HW Packet Processing

Harald Widiger, Stephan Kubisch, and Dirk Timmermann
Institute of Applied Microelectronics and Computer Engineering
University of Rostock, 18051 Rostock, Germany
{harald.widiger, stephan.kubisch, dirk.timmermann}@uni-rostock.de

Abstract— **In modern network applications and especially in Access Networks, the demands towards functionality and throughput are rising permanently. Furthermore, telecommunication carriers have different and changing requirements towards Access Network equipment. They are thus demanding a great deal of flexibility in IP-DSLAMS. To satisfy these various needs, highly flexible and particularly high performing packet processors are required. We propose an architecture for hardware modules, which joins the advantages of software and hardware solutions targeting packet processing. Our architecture provides a powerful and fast solution due to hardware implementation. Furthermore, it enables flexible and adaptive packet processors for different needs and configurations comparable to a software solution based on a network processor. This is accomplished without any overhead of unnecessary functionality and without the difficulties, which occur when it comes to adjusting pure ASIC packet processors to different tasks. Our architectural approach thus provides a good solution for packet processing.**

I. Introduction

In modern Access Networks, the demands towards the capabilities of network equipment such as IP Digital Subscriber Line Access Multiplexers (DSLAMs) and Broadband Remote Access Servers (BRAS) are rising permanently. Not only ADSL over ATM must be supported but also pure Ethernet and VDSL over Ethernet [11]. Internet carriers request more and more functionality already within the Access Network in order to relieve their core networks of extensive work load. The requirements on DSLAMs constantly evolve over time [1], [5]. There is functionality that must be implemented in the Access Network! Port information, for example, is required for Quality-of-Service (QoS) [4]. It is only available at the customer edge of the Access Network. Thus, functions depending on that port information must be executed within the DSLAM. Many of those functionalities like protocol recognition and classification, traffic shaping, packet modification, or error detection and correction are packet oriented. These tasks are usually performed by a packet processor. The purpose of a packet processor is to manipulate and to manage the transport of Ethernet frames or IP packets respectively. There are three ways of implementing packet processing tasks: Utilize a network processor (NP), a pure hardware solution, i.e. a specialized ASIC, or use a flexible hardware platform, which can be provided by Field Programmable Gate Arrays (FPGAs).

Basically, any computation and manipulation of a frame or packet can be performed with a network processor. NPs have the advantage of being highly flexible due to software programming. However, NPs show critical disadvantages if utilized in modern Access Network environments — relatively high monetary costs and technical limitations in throughput [7]. Even if a selected NP might have sufficient capabilities for today's applications, increasing its functional spectrum or adding extra tasks leads to insufficiency. If more bandwidth is required, the capacities of an NP are quickly exhausted. Adaptation can then only be achieved by implementing a more powerful NP, provided that there is one available.

Another approach is to choose a pure hardware solution, a specialized ASIC, which performs its tasks in hardware. Here, we experience extraordinary performance due to the hardware implementation of the processing tasks. But the mayor drawback of an ASIC solution is the lack of flexibility. If the ASIC solution cannot perform a special task sufficiently, we cannot make a trade-off between speed and functionality. To perform new tasks, a second processor is required to perform the computation the ASIC is not capable of. On the processor side again, we lack performance for those tasks.

To overcome the need to decide between performance and flexibility, FPGAs appear to be a good choice. [9] shows that the gap between modern FPGAs and standard cell implementations is less than factor 5. A customized FPGA design can provide a solution tailored for a current problem. Furthermore, changing circumstances and demands towards functionality or throughput can be answered in a short time and, in most cases, be implemented into the same FPGA hardware. This can even be done in the field. Thus, many manufacturers of network equipment implement FPGAs for special purposes besides standard NPs or ASICs within their solutions, which can be used for packet processing without increasing a systems' costs significantly. However, a main drawback of any hardware solution is the effort for development and adaptation. Today, it is almost always simpler to reprogram an NP than to design a new hardware platform.

Therefore, it is desirable to combine the advantages of software- and hardware-based solutions to meet today's requirements towards packet processing equipment. At the same time, the disadvantages of either solution should be eliminated. Ideally, this leads to a fast and powerful hardware-based packet processor, which is furthermore both flexible and easily customizable and adaptable. We propose an architecture for hardware modules, which fulfills these requirements. Section II describes the structure of the functional modules a customizable packet processor can be built of. Implementation and simulation results are presented in Section III. Before concluding the paper, an exemplary architecture of a packet processor is briefly presented in Section IV.

II. Architecture of Functional Modules

Conceptually, all operations, which require packet processing, can be partitioned into two major classes: Fast path operations and slow path operations [13]. Slow path operations usually require less computational power because they occur relatively seldom or have little relations to data path operations. General purpose processors can perform these operations with sufficient performance. Fast Path operations on the other hand require much more computational power in order to be performed at wire-speed and with highest throughputs. They are predestined to be implemented in specialized hardware. To achieve a flexible and adaptable solution, the approach we present proposes a modular architecture to build and connect functional modules (FMs). Different functionalities regarding packet processing can be implemented in these FMs. The FMs consist of a well defined data path interface for in- and egress of fast path data and a control interface. The control interface enables module control, status information gathering, and access to a CPU for control plane (slow path) operations.

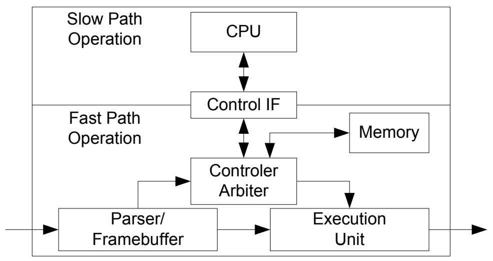

Fig. 1. Simple FM architecture comprising of a frame buffer, a memory, a control interface, and a submodule implementing the desired functionality

Each single FM must have the same interfaces to the slow and to the fast path. It is thus possible to create a very simple application specific packet processor by simply combining the FMs fulfilling all the desired functionalities. As the FMs are connected to each other by the defined interfaces, the internal architecture of an FM is in principle irrelevant for interconnecting many modules in the packet processor. However, to avoid a bottleneck in the data path, all modules shall have the same minimum data rate, because the FM with the narrowest bandwidth limits the whole system. The developed architecture is capable of managing high throughputs without inserting much latency. In this architecture (Figure 1), FMs basically consist of three main elements: A frame buffer (FB) with a parser, a controller and arbiter (CA) connected to a lookup memory and the control plane interface, and an execution unit (EU). The system works as follows: Incoming frames are buffered in the FB, an identifier (key) is extracted and sent to the CA, which coordinates the memory look up. The look up result is then presented to the EU. Depending on the result, the EU performs its task. An example: In a router, a single rule of a memory entry can be formulated like: if (packet.dest_IP equals 130.60.48.8) then route-to-port_5. Thus, the FB presents the extracted destination IP address of each packet as key to the CA, which organizes the memory look up. The memory module stores both the rules and the keys. It thus delivers the rule ”route-to-port_5”. Together with the frame, that rule is presented to the EU. The EU sorts the frame into a queue that serves the router’s output number.

A. Frame Buffer

Regarding latency, the FB is a decisive element of an FM. It has two main tasks: buffering frames and key parsing. All incoming frames are buffered in an internal Block RAM, which is implemented as FIFO of a variable size. It is thus not required to store incoming frames in an external memory as done in usual network processor architectures [7]. As the buffering of complete frames is done within the data path and exclusively for the actual FM, no unnecessary additional delay is inserted, as would be when storing frames in an off-chip memory. Furthermore, it is not required to implement mechanisms for off-chip memory operations. This way, the problem of repetitive read-write operations on an external memory, which is a known software problem [6], is avoided by the use of integrated FBs. However, the disadvantage is that on-chip RAM within an FPGA is rare and expensive compared to off-chip RAM. As a modern Xilinx FPGA, the XC4VFX20 offers no more than $1.2\ \mathrm{Mbit}$ of on-chip memory. The only way of increasing the internal memory resources is choosing a larger FPGA, with the limit being at $10\ \mathrm{Mbit}$ (Xilinx Virtex4 XC4VFX140 [14]).

As mentioned before, besides buffering frames, the FBs perform a second task. They parse all incoming frames and extract parts of the frame’s headers — the keys, which are used for packet classification. The classification result is a rule that determines how to proceed with a packet within the EU of the FM. Therefore, the parser can be configured to extract different parts of an Ethernet frame independently from each other. Every parsing block can be instantiated at compilation time. Thus, only the necessary parts of the frame’s header are extracted and the logic requirements are minimized. Furthermore, it is extremely simple to add a logic block to extract another part of the frame, e.g., the layer four port information. Depending on the result of the key look up, the frame is either forwarded to the execution unit or to the control interface.

B. Controller and Arbiter

The CA submodule organizes any communication within the FM. It acts mainly as an arbiter for any access to the memory. As can be seen in Figure 1 this task appears to be pretty simple. When concurrent requests occur, the only decision to make is which element should get access to the memory, the FB or the control interface? To assure data integrity, accesses from the control interface have always the highest priority. However, in order to adapt the possible throughput of an FM to desired rates, the data paths (FBs and EUs) can be multiplied. When running at $125\ \mathrm{MHz}$, the capacity of one data path is $1\ \mathrm{Gbit/s}$. Thus, for an $n\ \mathrm{Gbit/s}$ data path parallel and independent FBs and EUs are required.

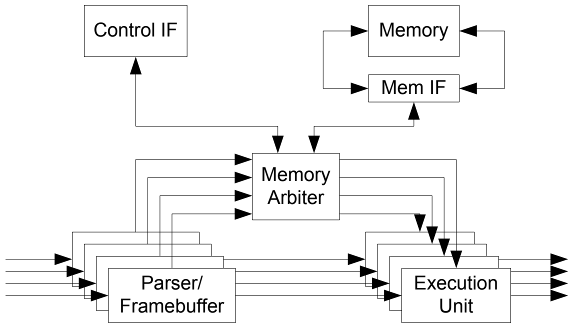

Fig. 2. FM with four independent data paths and dual port memory

Nevertheless, the memory remains a single one. Lookup requests from all data paths compete for memory accesses for classification. Thus, the necessity to implement an arbitration element arises. The arbiter has to efficiently and fairly schedule all competing memory access requests. To do so, with least laxity first (LLF), a fair scheduling algorithm is implemented. LLF, usually used for process scheduling in operating systems, assigns the memory access to the key with the smallest slack. In process scheduling, the smallest slack is computed as difference between two parameters of the involved processes—deadline and computation time. In case of two equal slack values, the process with the smallest deadline is scheduled. In the FM the deadline derives from the space left in the FIFO buffers. The computation time derives from the number of frames that are already stored in that FIFO. In case the control plane requires memory access, it has always the highest priority to ensure that the memory content is up to date.

C. Execution Unit and Memory

The EU represents the functional core of the FM. Different functions can be implemented. As part of our research, we implemented functionalities for MAC address translation [8], Traffic Management [8], and Multi Protocol Label Switching (MPLS) [12]. The architecture of all three implementations is alike. An incoming frame is modified with respect to the rule found in the memory. The MPLS-FM for example, inserts an MPLS label stack into each frame. The content of the stack is stored in the memory and presented to the EU together with the frame.

Special attention has to be paid to the memory implementation as it is the main bottleneck of the system. For our implementation, we used FPGA internal BRAMs as true dual port RAM (Figure 2). The memory itself is organized as a sorted list. This has the advantage of relatively fast memory lookups when searching an entry $(O(\log 2(n)))$. The disadvantage is that the costs for inserting and deleting entries rises to $O(\log 2(n) + n)$. But as the databases within DSLAMs do change relatively seldom, the additional costs for inserting and deleting can be spent. The implemented search algorithm takes two clock cycles to inspect an entry. For a memory consisting of 2048 entries, 5 million frames can be computed in one second. Considering the minimal frame size of an Ethernet frame of 64 byte, and an inter frame gap of 12 byte, the module is capable to guarantee a data throughput of $3\,\mathrm{Gbit/s}$ without any loss. However, when transferring only minimal sized frames, a GbE channel is not expected to reach a throughput of $1\,\mathrm{Gbit/s}$. Thus, the realistic data throughput exceeds $3\,\mathrm{Gbit/s}$. Assuming an average frame size of 400 byte, as suggested by the research reported in [2], $15\,\mathrm{Gbit/s}$ of data can be computed.

III. Implementation Results and Performance

The previously mentioned MPLS module was implemented and simulated as an example FM. It was implemented as drawn in Figure 2 with $4\,\mathrm{Gbit/s}$ throughput. The implementation required around 5000 slices on a Xilinx XC4VFX20. The simulations were performed according to [3], which defines benchmark methodologies and theoretically maximum througputs. The graphs in Figure 3 show the module's performance with that configuration. The graphs reveal throughput and frame loss rate of the FM. With the theoretically maximum input data rate, the throughput differs for 64 byte frames between 65% and 71% of the maximum throughput (depending on the size of the look up memory). As stated before, the memory lookups are the limiting factor of the system. When increasing the number of entries in the memory, or decreasing the size of the induced frames, the maximum throughput decreases. However, when the induced frames exceed a size of 256 bytes and hold 8k entries in the memory, a throughput of 98% of the theoretical maximum and more is reached. The same is true for the drop rate of frames. The more memory entries and the higher the injection rate, the more frames are dropped. Only with frames sized between 64 and 512 byte frame drops occur, while the drop rate differs from 16% to 1%. In real network traffic, the distribution of frame sizes is as follows: Frames with sizes of 64, 594 and 1518 Bytes are dominating the traffic [2], [10]. 64 Byte (minimal) frames represent 35% of the frames, 594 Byte frames 11%, and maximum sized frames around 10% of the traffic. Other frame sizes can be considered as being uniformly distributed. The average size of an IP-Packet is 402 Byte [2], leading to Ethernet frames with a size of 416 Byte. Considering such an average frame size, only minimal frames losses must be expected when computing 4 or even $8\,\mathrm{Gbit}$ channels in an FM. Furthermore, the average delay, inserted by an FM is about $130\,\mu\mathrm{s}$.

IV. Packet Processor System

With the FMs described, it is possible to construct a packet processing system (PPS). Figure 4 shows an example of a straight forward system. It is implemented as a simple pipeline structure of different FMs within an FPGA. At ingress and egress of the device, Medium Access Controllers (MACs) connect to the physical medium. All FMs are serially connected. As a result, all implemented functionalities can be applied to each frame traversing the PPS. Such system has two very important features. It provides both high performance and flexibility. On the one hand, because it is a pure hardware solution, the PPS reaches a great throughput and low latency. On the other hand, as it is a reconfigurable hardware, it is very

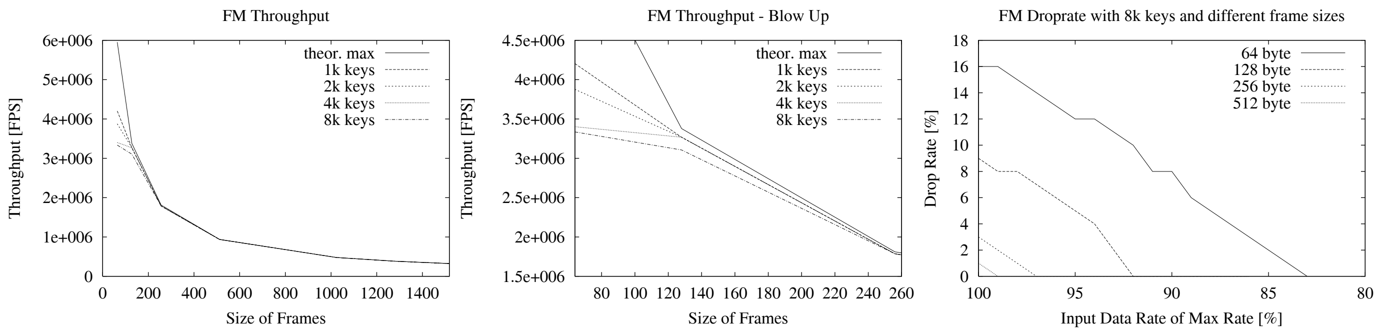

Fig. 3. Behaviour of an FM with a $4\ \mathrm{Gbit/s}$ data path

flexible. Comparable to software updates or new applications of common NPs, a new configuration file with new, changed, or improved FMs can simply be downloaded to the FPGA the packet processor is implemented on. Thus, when implemented on an FPGA, the packet processor is adaptable to changing requirements.

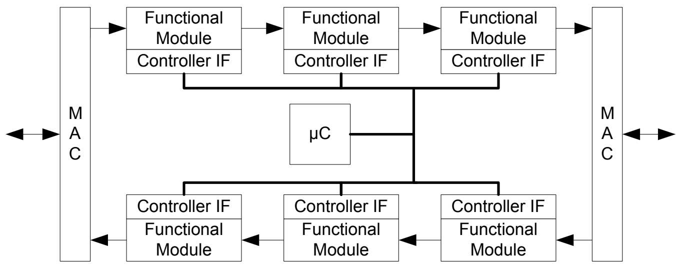

Fig. 4. System architecture of a configurable packet processor

Even the slow path can be integrated into today's FPGA platforms. Both Xilinx and Altera, the two major vendors of FPGAs offer micro processors as sources to be integrated on the FPGA platform. In some devices, Xilinx even offers hardwired PowerPC-Cores. To improve flexibility, different controller interfaces, both for Altera and Xilinx platforms, have been developed. At costs of around 300 logic cells each platform can be interfaced by the FMs in order to serve as CPU for slow path operations. Both interfaces are available as IP-cores to be connected to the FM's controller interface.

To further simplify and accelerate the generation of a packet processor, a software tool is under development supporting engineers in the process of generation and configuration of a complex packet processor. In such a software environment, an FM can be defined by setting parameters like buffer sizes, memory widths, desired throughput, and the number of desired database entries. To define a packet processor system, it is further required to compose and arrange desired FMs, preferably with a graphical user interface.

V. Conclusion

This paper has proposed an architecture for packet processing elements, which combines almost the speed and performance of dedicated hardware solutions and nearly the flexibility and configurability of network processor based software solutions. Even the integration of various processors is possible. Interfaces to connect the FMs to different μCs (NiosII, PowerPC) have been developed. Packet processing functionalities have been implemented and simulated with the proposed FM architecture and where capable of a throughput of $4\ \mathrm{Gbit/s}$. We shortly introduced a simple packet processor system, as can be built from the developed components.

Further research will be dedicated to the ideal arrangement of FMs to form a powerful packet processor system. The questions here are: How can the best throughput be reached for a PPS? Which type of PPS is required to reach the highest functional flexibility? Where is the optimum between theses goals? Finally, the development of the user friendly software system supporting the generation of a PPS, which was briefly introduced in Section IV, is also part of our future work.

References

[1] Agere Systems. Implementing Demanding Traffic Processing Requirements for Next-Generation DSLAM Network Architectures. 2005.

[2] Agilent Technologies. JTC 003 Mixed Packet Size Throughput.

[3] S. Brandner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, March 1999.

[4] DSL-Forum. DSL Evolution - Architecture Requirements for the Support of QoS-Enabled IP Services. In *Technical Report TR-059*, Sept. 2003.

[5] DSL-Forum. Multi-Service Architecture & Framework Requirements. In *Technical Report TR-058*, Sept. 2003.

[6] T. Hruby, W. de Brujn, H. Bos, M. L. Cristea, and L. Xu. Lessons learned in developing a flexible packet processor for high speed links. In *Technical Report IR-CS-016, Vrije Universiteit Amsterdam*, 2005.

[7] Intel. Intel®IXP2400 Network Processor. In *Product brief*, 2003.

[8] S. Kubisch, H. Widiger, D. Duchow, and D. Timmermann. Wirespeed MAC Address Translation and Traffic Management in Access Networks. In *Proc. The World Telecommunications Congress 2006 on CD-ROM*, May 2006.

[9] I. Kuon and J. Rose. Measuring the Gap between FPGAs and ASICs. In *Proc. The ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, pages 21–30, Jan. 2006.

[10] NLANR Measurement & Network Analysis.

[11] S. Park. Implementation of Next Generation VDSL Networks in Metro Ethernet Backbone Environments. In *Proc. Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/E-Learning on Telecommunications Workshop (AICT/SAPIR/ ELETE'05)*, pages 21–26, 2005.

[12] H. Widiger, S. Kubisch, D. Duchow, and D. Timmermann. A Simplified, Cost-Effective MPLS Labeling Architecture for Access Networks. In *Proc. The World Telecommunications Congress 2006 on CD-ROM*, May 2006.

[13] J. Williams. Architectures for network processing. In *International Symposium on VLSI Technology, Systems, and Applications*, 2001.

[14] Xilinx. Xilinx Virtex$^{\mathrm{TM}}$-4 Series FPGAs.