Hard- and Software Modularity of the NOVA MPSoC Platform

Christian Sauer, Matthias Gries, Sebastian Dirk
Infineon Technologies, Communications Solutions, Munich, Germany
{Christian.Sauer, Matthias.Gries}@infineon.com

Abstract

The Network-Optimized Versatile Architecture Platform (NOVA) encapsulates embedded cores, tightly and loosely coupled coprocessors, on-chip memories, and I/O interfaces by special sockets that provide a common packet passing and communication infrastructure. To ease the programming of the heterogeneous multiprocessor target for the application developer, a component based framework is used for describing packet processing applications in a natural and productive way. Leveraging identical application and hardware communication semantics, code generators and off-the-shelf tool chains can automate the software implementation process. Using a prototype with four processing cores we quantify the overhead of modularity and programmability for the platform.

1. Introduction

Platforms have been introduced for coping with the increased pressure on time-to-market, as well as design and manufacturing costs [5]. They particularly make sense in price-sensitive application domains, such as access networks, where application-specific optimizations are required to reach defined objectives on costs and performance. In addition, sufficient flexibility of the platform is required to deal with emerging trends. The main design goal is therefore to integrate as much functionality as possible while preserving as much flexibility as needed, keeping constraints on performance and costs. We have developed the heterogeneous NOVA platform for comprehensive exploration of design alternatives in hardware and software, where the refinement process is driven by network access applications.

The systematic development of a flexible platform aims at the following design principles:

Seamless trade-off analysis: The developer must be able to isolate the impact of different design decisions on the overall design quality. The platform concept must not restrict the designer on predefined design choices.

Enabling reuse: The platform must be general enough so that existing design knowledge can be incorporated to avoid starting from scratch. At the same time, reuse must not constrain the designer to after-effects of reusing one particular design block.

Customization path from standard components: If standard components are employed, there must be clear ways for improving design criteria by following defined customization paths if design targets are not met.

For achieving these goals with our NOVA platform, we apply the following design rules:

Modularity: Generalized sockets are used to encapsulate hardware building blocks so that interfacing to processing resources can be decoupled from interfacing to the interconnect infrastructure. Similarly, our software framework is organized in exchangeable components that represent computational kernels.

Keep it simple: We advocate this discipline for all design decisions. Consequently, we prefer to, e.g., replace an existing bus interface with another one rather than writing a wrapper for the existing interface. It also means that we carefully reduce the complexity of standard processing elements as needed, e.g., by removing MMU, caches, and certain functional units.

Scalability: The platform must be able to accommodate a wide range of I/O interfaces and processing elements to handle different processing and communication requirements. The NOVA socket and message format allow the integration of traditional shared buses, as well as better scalable Network-on-Chip (NoC) techniques. A processing subsystem in NOVA is core-agnostic since we do not rely on proprietary interfaces for, e.g., coprocessor coupling.

Customization where needed: If the most flexible solution does not meet design criteria, NOVA provides design points for customization. Some of the PEs allow application-specific instruction set extensions. Coprocessors, either tightly or loosely coupled, can be used without breaking the representation of the application.

This paper describes the NOVA platform that leverages commodity blocks by unified sockets and enables the disciplined exploration of all design space aspects. Besides platform concepts, this paper quantifies feasible tradeoffs between the costs of modularity and performance. In the

978-3-9810801-2-4/DATE07 © 2007 EDAA

next section, we introduce the hardware aspect of NOVA and describe its modularity. Section 3 discusses the associated programming model. The FPGA implementation of a prototype running a real world application is explained in Sec. 4. The results section (5) evaluates design tradeoffs due to modularity and programmability. Related work is surveyed in Section 6. We conclude in Section 7.

2. NOVA Platform

The NOVA platform provides concepts, templates, and various building blocks for the systematic application-driven design space exploration of network processors. NOVA eases the use of commodity IP modules. They are encapsulated by the NOVA socket, which provides a unified interface to the on-chip network. In this way, all platform elements can be connected to each other and form arbitrary on-chip communication topologies.

2.1. On-chip communication

NOVA supports two types of on-chip communication in hardware: message passing and memory accesses. Messages are primarily used to transfer packet descriptors between processing nodes and are akin to the packet streaming semantics of the application domain. In addition, processing nodes can exchange system messages, e.g. for OS-like functions. Messages use on-chip routing headers and are between 12 and 64 bytes long. Message passing usually is non-blocking. A backpressure scheme implemented by the interconnect network, however, provides means to block a producer if desired.

Memory accesses may be split transactions, as long as the sequential order is maintained. Depending on the type of processing element, memory accesses can be implemented as blocking or non-blocking.

2.2. NOVA socket

The socket decouples the on-chip communication network from the processing nodes and provides unified interfaces between them. This is, for instance, helpful for the exploration of different communication schemes. Figure 1 shows the concept. The socket encapsulates an IP module and provides external interfaces. The figure shows three interfaces as they are implemented by our prototype (cf. Sec. 4): to the packet descriptor (PD), the system message (SM), and the memory access networks. The internal interfaces are specialized to the particular IP module.

Usually, *N*x*M* on-chip interfaces are required to explore *M* communication schemes for *N* different IP modules. By defining a handshake protocol between IP and NoC interfaces, the socket reduces this to an *N*+*M* complexity.

A welcome side effect of this approach is the option to insert FIFOs for globally asynchronous communication schemes. In Figure 1, the message passing networks are asynchronous, whereas the memory access network is connected synchronously. Optionally, DMA engines can be included in the sockets. These units can be customized to convert streaming interfaces into memory accesses and vice versa. The IO interface (cf. Figure 2), for instance, uses them to transfer packets to/from the memory.

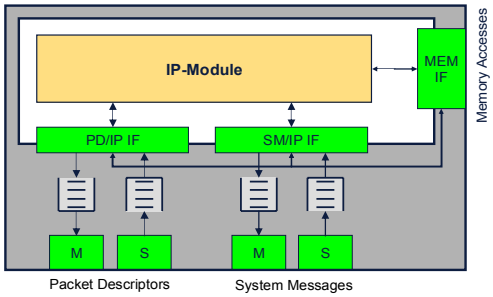

Figure 1: NOVA socket concept.

2.3. Platform building blocks

Deploying the socket and communication concepts, NOVA defines different types of building blocks.

Processing elements (PE): A PE is an embedded 'standard' processor that can be programmed in a high-level language. This processor and its subsystem, e.g. code and data memories, are encapsulated to form the PE. The PE in this paper uses a 32b MIPS 4K with Harvard architecture.

Coprocessors: These are specialized engines and accelerators, which cannot be programmed in a high-level language. They are deployed either tightly coupled in a processing elements' subsystem or loosely coupled as specialized processing node. NOVA uses coprocessors, e.g., for security functions and memory management.

On-chip memories: NOVA supports arbitrary on-chip memories. Currently the memory interface defines 32 bit wide addresses and data words and assumes pipelined synchronous memories. If encapsulated in sockets, memories can form co-processors accessed via system messages.

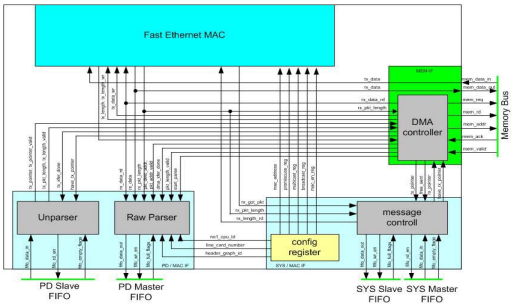

Figure 2: NOVA Ethernet IO module.

Off-chip interfaces: Off-chip interfaces are mostly off-the-shelf units encapsulated by NOVA sockets. Current emphasis is on network and memory IO. Figure 2 shows a Fast Ethernet MAC with its interface to the socket. Apart the MAC with memories, the module contains DMAs that autonomously store and forward packets, and parser /unparser units for the handling of packet descriptors.

2.4. Memory layout and hierarchy

NOVA does not impose any memory layout or hierarchy. PEs may use transparent cache hierarchies or deploy memories that are visible to and managed by the programmer. Memories shared between PEs require a unique resource manager either in hardware or software. Memories and all resources accessed by a PE are mapped into the individual PE's data memory map.

3. Programming Model & Deployment

In this section we show how concurrency and modularity is supported in our software description and generation process. We start from modular Click descriptions that we use for modeling functionality hardware independently. We use this input for code generation on embedded processors. Wrapper elements in Click and a thin OS layer used by the generated code take care of the specifics of the underlying multiprocessor system.

We use Click [3] for modeling the functionality of packet processing systems. Click descriptions are modular, executable, independent of a particular hardware architecture, and capture inherent parallelism in packet flows and dependencies among elements (Click components). Click elements describe computational network kernels, whereas connections specify the flow of packets (i.e. data) between elements. Application state is kept local within elements. Due to defined element interfaces, a Click description can quickly be customized to new protocols and environments by exchanging individual elements. Finally, all processing activity is initiated by the transportation of packets.

Our CRACC [9] single processor code generator takes a Click description and instantiates elements from a library written in C. CRACC elements are connected and configured at compile-time. Also, CRACC's memory footprint is much more compact than Click's C++ description. These techniques are necessary since our code generation targets are embedded processors, where optimization for code size and performance is most important.

3.1. Wrappers for heterogeneous platforms

A heterogeneous platform such as NOVA may contain many different building blocks. To incorporate their behavior into Click representations and the code generator, we distinguish between functionality that is made explicit in Click and functions that should be hidden from the application developer. In this subsection, we look at Click-conforming representations by using wrapper elements that encapsulate interfacing with hardware-specifics. Other functions are addressed in the following subsection.

Packet descriptor passing: If two Click or CRACC elements communicate with each other, pointers to context information are normally handed from element to element on the same processor. If these elements are mapped onto different processors, the message passing interface must be used, i.e. the context data must be copied into the interface buffers and routing information must be added, such as the on-chip destination address. *FromIO* and *ToIO* elements have been implemented for encapsulation of receive and send functionality of message passing hardware, respectively. Several *FromIO* and *ToIO* elements can be associated with the same message passing interface in hardware. The different software instances are distinguished by a unique graph ID that is also contained in the routing information of the message.

Hardwired coprocessors and network I/O: For modeling the function of coprocessors and off-chip communication interfaces, Click elements are needed that emulate the behavior of the module, e.g., for verification purposes with artificial traffic sources. For these elements code generation might not be necessary at all, but the full model is executable in Click. Click wrapper elements can also be used for configuring hardware blocks, i.e. code generation takes care of initializing the hardware block accordingly.

Mapping annotations: On a multiprocessor platform, the designer has the choice to partition the application onto several processing elements. For CRACC together with NOVA, this is a manual process where the designer annotates certain Click elements with mapping targets. A mapping target is defined by a node ID in the system. The part of a Click graph subject to mapping is specified by a unique graph ID so that the specification of mapping for individual elements can be avoided. *From-* and *ToIO* elements described earlier inherently are start and end-points of partial Click graphs. It is sufficient to specify a mapping for the *FromIO* element. This information is propagated during the CRACC code generation phase.

3.2. Multi-core and OS extensions for CRACC

Apart from Click wrappers we need additional services for messages, timers, task scheduling, and resource sharing among several processing elements. Since such mechanisms are not part of the Click syntax, these features are hidden from the Click representation and only partly visible for a library programmer.

System messages: Apart from the message passing mechanism that is visible in Click, we use message passing for exchanging information used by the OS, such as status messages, hardware module configuration data and requests for a certain shared resource. These system messages are shorter than packet descriptor messages but use the same routing specification (message header).

Visibility of memory hierarchy: In CRACC, a library programmer can explicitly address different memory areas, e.g. private local and shared slow memories. Every shared memory is associated with a unique memory manager that can be mapped to any PE, e.g. a coprocessor or a programmable core. Requests for memory access, allocation, and deletion are sent by system messages to the associated manager, which replies accordingly.

Timers: CRACC provides an API for timers that can be used, for instance, by timed Click elements. Timed elements register themselves for wakeup at a certain expiration date. Timers encapsulate the specifics of a target's implementation, e.g. a hardware timer that is register mapped, memory mapped, or a co-processor.

Split transactions: A direct consequence of using system messages in a GALS platform is the support of split transactions for latency hiding. If the sender of a system message is waiting for a response, it registers itself for wakeup by the scheduler on the respective processing core when the corresponding reply message arrives. Context switches caused by split transactions are explicit and require only minimal state embedded in the registration at the scheduler. The register file does not need to be saved.

3.3. Deployment

The software partitioning and development process that we follow can be summarized as follows:

1. Evaluation of Click model: The full system function is modeled in Click. In this way, the required packet processing can be determined and simulated with real or artificial network traffic on any Linux computer.

2. Profiling on single-core: The Click graph can be used by our code generator CRACC [9] to map the system function on a single core target, where it can be profiled in terms of per-packet processing requirements.

3. Partitioning of Click graph: Based on the profiling results of the preceding step, hot spots can be identified and feasible partitions of the graph onto several processing elements can be determined manually, as described earlier.

4. Multiprocessor code generation using CRACC: The mapping annotation is used to individually generate code for different processor targets.

5. Determine performance and reiterate: The properties of the full implementation can now be determined (e.g., speed, code size) by simulation or on the actual platform hardware. Reiterate starting at step 3 by repartitioning the Click graph until objectives are met.

Following these steps, a functionally correct implementation of the application can be derived quickly using Click. The subsequent performance optimization can then focus on individual elements and the partitioning of elements onto processing cores. This systematic approach leads to improved design productivity and simplifies reuse.

4. Platform Prototype

We have implemented a 4PE prototype of the NOVA platform that realizes a Digital Subscriber Line Access Multiplexer (DSLAM). Figure 3 shows the block diagram of the 4PE NOVA prototype. The device is dimensioned for the use as DSLAM line card processor and employs four of the MIPS processing elements (shown in Figure 3). Leveraging the FPGA-based Raptor2000[1] prototyping environment the system implements four Fast Ethernet IOs (Figure 2) that are connected to external PHYs and shared off-chip SRAM memory. The on-chip communication is based on three OCP busses for system messages, packet descriptors, and memories. In this way, high priority delivery of system messages is assured. The prototype also integrates a statistics and profiling module to derive run-time performance information.

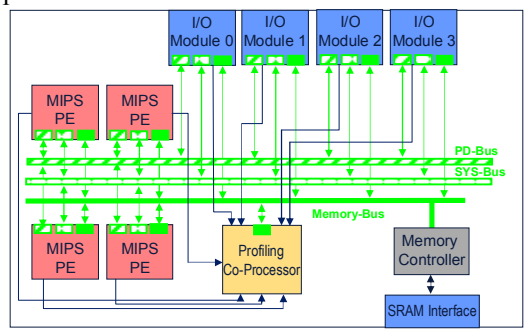

Figure 3: 4PE NOVA prototype.

4.1. DSLAM application

Our NOVA prototype runs the DSLAM application as described in [8]. Most of the functionality is mapped on the four PEs, only the Ethernet functions are absorbed in hardware by the NOVA IO modules. Using the NOVA realization on the Raptor2000 system, we connect the prototype to a set of traffic sources and sinks that represent DSL customers' voice, video, and best effort traffic with different Quality-of-Service requirements (cf. Figure 4).

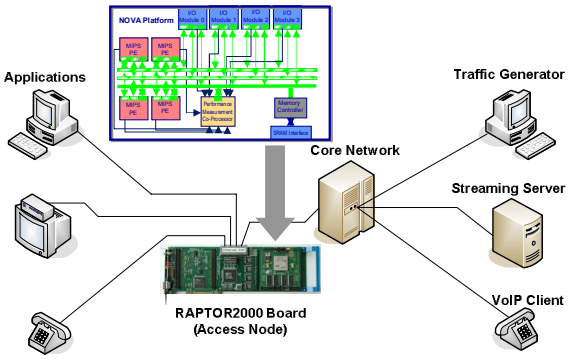

Figure 4: 4PE NOVA application setup.

4.2. Profiling support

The statistics and profiling coprocessor is connected to all resources and collects data from nodes and the on-chip network. It derives information about the packet throughput and loss, the processor load and software profile (by tracing the instruction address stream), and the utilization of the on-chip communication system at run-time.

[1] www.raptor2000.de

5. Performance Results

To evaluate the overhead of programmability and modularity we synthesize the 4PE prototype for the FPGA based Raptor2000 prototyping environment and a 90nm ASIC design flow. On a Xilinx XC2V6000-4 device the system runs at a convenient clock frequency of 25MHz.

5.1. Hardware modularity

The NOVA socket interfaces to three on-chip communication networks. Its area is dominated by the transfer queues for packet descriptors and system messages.

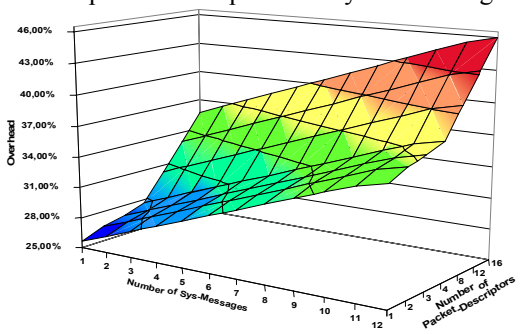

Figure 5: Relative area overhead of the NOVA Fast Ethernet IO socket.

Looking at NOVA's Fast Ethernet IO module in Figure 5, we determine the area of its socket compared to the embedded Ethernet MAC for the ASIC version. Depending on the number of queue entries and assuming equally sized receive and transmit queues the socket overhead is between 25% and 46%. Simple SoC bus interfaces without buffering and NoC capabilities require less area. A single PLB bus interface without memories, e.g., is only 1% of the MAC area. Buffers included, notably more area is required. Using a single Wishbone interface the overhead is more than 60% for Opencore's MAC. This indicates that the area for traditional SoC bus interfaces is similarly dominated by buffering. The required area for the socket is within the range of common bus interfaces.

5.2. Software modularity

To determine the runtime overhead of our modular programming environment, we use the CRACC code generator [9] and run the IP-DSLAM benchmark. This "out-of-box" version strictly preserves Click's modularity and object-oriented runtime features, such as virtual functions. In a second step, we de-virtualize functions and resolve push and pull chains statically (CRACC optimized).

In Figure 6, this is compared to a simple ANSI-C program (straight calls) that calls all functions directly from a central loop, without Click's function call semantics. The figure reveals that CRACC with static optimizations does not impose more overhead than the straight-function-call approach (for a given granularity). There is still a penalty of 30% for the structured and modular approach compared to a program that inlines all functions into a single packet processing loop (all-in-one).

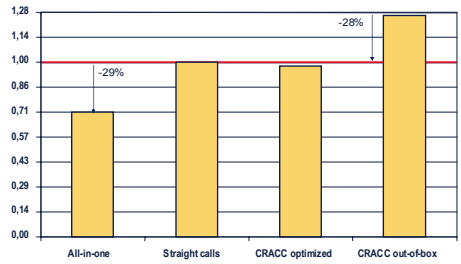

Figure 6: Overhead of modular software.

5.3. Runtime performance

After discussing hard- and software design trade offs we now look at the impact of modularity and programmability on the runtime performance. We analyze two aspects of the 4PE NOVA prototype: the packet latency through the system and the maximum packet throughput.

Packet latency. The latency of packets through a system can be significantly larger than their actual processing time due to receive and transmit related overhead. For the analysis, we therefore focus on the essential functionality that is required to set up the packet descriptor and move the packet content from network interface to network interface. For this measurement, buffer management is implemented in software on one PE. The path through the system, as displayed in Figure 7 in clock cycles, starts at the ingress Ethernet interface. The I/O module requests shared memory space by sending a system message to the buffer manager (BM). The BM replies with an address of a free segment. The BM accesses the shared memory for updating segment context information (not shown as separate entity). The packet contents can now be transferred to shared memory and the reception signaled to the downstream processing node, in our case the egress I/O module. This module finally reads out the packet (finished after 375 cycles) and releases the memory segment.

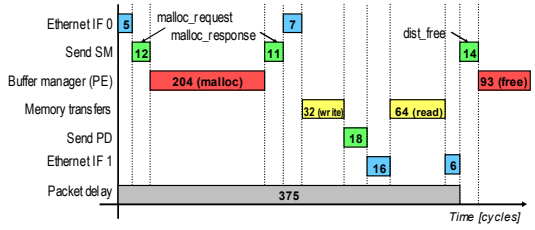

Figure 7: Latency of a 64 Byte packet.

We can recognize the benefit of implementing message passing in hardware that supports the data-driven activation of processing elements. In this way, much of the latency due to signaling packet events can be avoided. In [4], it is shown that the Intel IXP 1200 needs about 1500 cycles for handing 64B packets from and to the network interfaces alone. From Figure 7 we learn that these parts

need less than 200 cycles in NOVA. Even if we consider that we use SRAM whereas the IXP employs SDRAM, we can recognize a clear acceleration of the interaction between interfaces and PEs.

Maximum packet throughput. We are interested in the bounds on the packet throughput due to sockets and the interaction with network interfaces. In Figure 8, we compare wire speed on Fast Ethernet (100 Mbps) with measurements on the 4PE prototype for the same setup used in Figure 7, i.e. buffer management is implemented in software. We recognize that we achieve line speed for Ethernet frame lengths larger than 200 Byte, which corresponds to a usable cycle budget of about 350 cycles per core and 1000 cycles for three PEs, respectively.

Since the ASIC implementation is one order of magnitude faster, already the 4PE prototype is powerful enough to support common DSLAM linecard configurations.

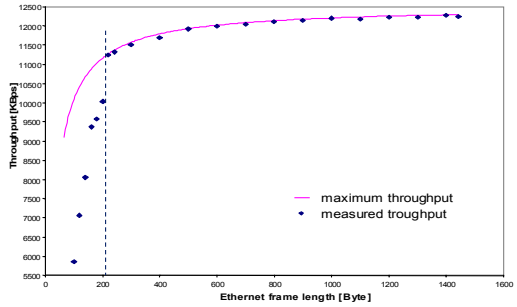

Figure 8: 4PE NOVA packet throughput.

6. Related Work

We find related work on network platforms with respect to software development, hardware architectures, and integrated platform approaches.

Platform approach: StepNP [7] is a framework for network processors that uses Click as application description. The software is based on C++ and provides message passing programming abstractions. Examples of commercial platform solutions for wireless and multimedia domains are Nomadik from ST Microelectronics, and Philips Nexperia. The VSI alliance has a broad scope and aims at virtual IP sockets to enable reuse and integration.

Network processor software: Click [6] is implemented for Linux using C++. SMP-Click [2] is a multi-threaded Linux variant. NP-Click [10] uses Click as a programming model for the Intel IXP network processor. Shangri-La [1] allows the automatic merging and partitioning of packet processing kernels onto several processing engines based on heuristics and profiling results.

Network processor hardware: A survey of the broad variety of network processor architectures can, e.g., be found in [11]. Commercial tools for customizing the micro-architecture of processing elements include CoWare's LISATek and Tensilica's Xtensa [3]. Networks on Chip (NoC) are, for instance, offered and used by Arteris SA, ST Microelectronics, and Sonics Inc.

7. Conclusions

NOVA is a modular and programmable hardware platform for packet-processing systems. It is based on unifying sockets and common packet passing and communication infrastructure for integrating various building blocks. Heterogeneous NOVA multiprocessors can be programmed intuitively and productively in a component-based framework. Due to matching communication semantics of application and architecture, a thin OS layer and code generation framework ease the application to architecture mapping significantly. Our results show that the overhead of hardware and software modularity is reasonable for NOVA compared to state-of-art techniques; and that NOVA is usable for the systematic application-driven design space exploration of network processors.

Acknowledgements

This project is founded by the German grant 01AK065A (PlaNetS), and the Bavarian grant IuK178/001 (SmartFlow). The groups of Prof. U. Rückert, University of Paderborn, and Prof. A. Herkersdorf, TU Munich, are contributing to this project.

References

[1] M.K. Chen, X.F. Li, et al.: Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming, PLDI, June 2005

[2] B. Chen, R. Morris: Flexible Control of Parallelism in a Multiprocessor PC Router, USENIX, June 2001

[3] R.E. Gonzalez: *Xtensa: a configurable and extensible processor*, IEEE Micro, 20(2), Mar./Apr. 2000

[4] M. Gries, C. Kulkarni, et al.: *Comparing Analytical Modeling with Simulation for Network Processors*, DATE, March 2003

[5] K. Keutzer, S. Malik, et al.: *System Level Design: Orthogonalization of Concerns and Platform-Based Design*. IEEE Trans. on CAD of Int. Circuits and Systems, 19(12), Dec. 2000

[6] E. Kohler, R. Morris, et al.: *The Click modular router*, ACM Transactions on Computer Systems, 18(3), August 2000

[7] P. Paulin, C. Pilkington, E. Bensoudane: *StepNP: A System-Level Exploration Platform for Network Processors*, IEEE Design and Test of Computers,19(6), Nov./Dec. 2002

[8] C. Sauer, M. Gries, S. Sonntag: *Modular Reference Implementation of an IP-DSLAM*, ISCC, Spain, June 2005

[9] C. Sauer, M. Gries, S. Sonntag: *Modular Domain-Specific Implementation and Exploration Framework for Embedded Software Platforms*, DAC, June 2005

[10] N. Shah, W. Plishker, K. Keutzer: NP-Click: *A Programming Model for the Intel IXP1200*, Network Processor Design, vol. 2, Morgan Kaufmann, Nov. 2003

[11] B. Wheeler, L. Gwennap: *A Guide to Network Processors*, 7th Edition, The Linley Group, Dec. 2005

[12] Agilent Technologies: JTC 003: *Mixed packet size throughput*, Journal of Internet Test Methodologies, Sept. 2004